Db2 11.1 for Linux, UNIX, and Windows

Performance Tuning

Db2 11.1 for Linux, UNIX, and Windows

Performance Tuning

Notice regarding this document

This document in PDF form is provided as a courtesy to customers who have requested documentation in this format. It is provided As-Is without warranty or maintenance commitment.

Contents

Figures

Tables

Performance overview

Performance refers to the way that a computer system behaves in response to a particular workload. Performance is measured in terms of system response time, throughput, and resource utilization.

Performance is also affected by:

- The resources that are available on the system
- How well those resources are used and shared

In general, you will want to tune your system to improve its cost-benefit ratio. Specific goals could include:

- Processing larger, or more demanding, workloads without increasing processing costs
- Obtaining faster system response times, or higher throughput, without increasing processing costs
- Reducing processing costs without degrading service to users

Some benefits of performance tuning, such as a more efficient use of resources and the ability to add more users to the system, are tangible. Other benefits, such as greater user satisfaction because of quicker response times, are intangible.

Performance tuning guidelines

Keep the following guidelines in mind when developing an overall approach to performance tuning.

- **Remember the law of diminishing returns**: The greatest performance benefits usually come from your initial efforts.
- **Do not tune just for the sake of tuning**: Tune to relieve identified constraints. Tuning resources that are not the primary cause of performance problems can actually make subsequent tuning work more difficult.
- **Consider the whole system**: You cannot tune one parameter or resource in isolation. Before you make an adjustment, consider how the change will affect the system as a whole. Performance tuning requires trade-offs among various system resources. For example, you might increase buffer pool sizes to achieve improved I/O performance, but larger buffer pools require more memory, and that might degrade other aspects of performance.
- **Change one parameter at a time**: Do not change more than one factor at a time. Even if you are sure that all the changes will be beneficial, you will have no way of assessing the contribution of each change.
- **Measure and configure by levels**: Tune one level of your system at a time. System levels include:
  - Hardware
  - Operating system
  - Application server and requester
  - Database manager
  - SQL and XQuery statements
  - Application programs

- **Check for hardware as well as software problems**: Some performance problems can be corrected by applying service to your hardware, your software, or both. Do not spend excessive time monitoring and tuning your system before applying service to the hardware or software.
- **Understand the problem before you upgrade your hardware**: Even if it seems that additional storage or processor power could immediately improve performance, take the time to understand where your bottlenecks are. You might spend money on additional disk storage, only to find that you do not have the processing power or the channels to exploit it.
- **Put fallback procedures in place before you start tuning**: If tuning efforts result in unexpected performance degradation, the changes made should be reversed before you attempt an alternative approach. Save your original settings so that you can easily undo changes that you do not want to keep.

Developing a performance improvement process

The performance improvement process is an iterative approach to monitoring and tuning aspects of performance. Depending on the results of this performance monitoring, you will adjust the configuration of the database server and make changes to the applications that use the database server.

Base your performance monitoring and tuning decisions on your knowledge of the kinds of applications that use the data and on your understanding of patterns of data access. Different kinds of applications have different performance requirements.

Any performance improvement process includes the following fundamental steps:

1. Define the performance objectives.
2. Establish performance indicators for the major constraints in the system.
3. Develop and execute a performance monitoring plan.
4. Continually analyze monitoring results to determine which resources require tuning.
5. Make one adjustment at a time.

If, at some point, you can no longer improve performance by tuning the database server or applications, it might be time to upgrade the hardware.

Performance information that users can provide

The first sign that your system requires tuning might be complaints from users. If you do not have enough time to set performance objectives and to monitor and tune in a comprehensive manner, you can address performance issues by listening to your users. Start by asking a few simple questions, such as the following:

- What do you mean by “slow response”? Is it 10% slower than you expect it to be, or tens of times slower?
- When did you notice the problem? Is it recent, or has it always been there?
- Do other users have the same problem? Are these users one or two individuals or a whole group?
- If a group of users is experiencing the same problem, are these users connected to the same local area network?
- Does the problem seem to be related to a specific type of transaction or application program?

- Do you notice any pattern of occurrence? For example, does the problem occur at a specific time of day, or is it continuous?

Performance tuning limits

The benefits of performance tuning are limited. When considering how much time and money should be spent on improving system performance, be sure to assess the degree to which the investment of additional time and money will help the users of the system.

Tuning can often improve performance if the system is encountering response time or throughput problems. However, there is a point beyond which additional tuning cannot help. At this point, consider revising your goals and expectations. For more significant performance improvements, you might need to add more disk storage, faster CPUs, additional CPUs, more main memory, faster communication links, or a combination of these.

Related information:

Best practices: Tuning and Monitoring Database System Performance

Performance tuning tools and methodology

Benchmark testing

Benchmark testing is a normal part of the application development life cycle. It is a team effort that involves both application developers and database administrators (DBAs).

Benchmark testing is performed against a system to determine current performance and can be used to improve application performance. If the application code has been written as efficiently as possible, additional performance gains might be realized by tuning database and database manager configuration parameters.

Different types of benchmark tests are used to discover specific kinds of information. For example:

- An *infrastructure benchmark* determines the throughput capabilities of the database manager under certain limited laboratory conditions.
- An *application benchmark* determines the throughput capabilities of the database manager under conditions that more closely reflect a production environment.

Benchmark testing to tune configuration parameters is based upon controlled conditions. Such testing involves repeatedly running SQL from your application and changing the system configuration (and perhaps the SQL) until the application runs as efficiently as possible.

The same approach can be used to tune other factors that affect performance, such as indexes, table space configuration, and hardware configuration, to name a few.

Benchmark testing helps you to understand how the database manager responds to different conditions. You can create scenarios that test deadlock handling, utility performance, different methods of loading data, transaction rate characteristics as more users are added, and even the effect on the application of using a new release of the database product.

Benchmark tests are based on a repeatable environment so that the same test run under the same conditions will yield results that you can legitimately compare.

You might begin by running the test application in a normal environment. As you narrow down a performance problem, you can develop specialized test cases that limit the scope of the function that you are testing. The specialized test cases need not emulate an entire application to obtain valuable information. Start with simple measurements, and increase the complexity only if necessary.

Characteristics of good benchmarks include:

- The tests are repeatable
- Each iteration of a test starts in the same system state
- No other functions or applications are unintentionally active in the system
- The hardware and software used for benchmark testing match your production environment

Note that started applications use memory even when they are idle. This increases the probability that paging will skew the results of the benchmark and violates the repeatability criterion.

Benchmark preparation

There are certain prerequisites that must be satisfied before performance benchmark testing can be initiated.

Before you start performance benchmark testing:

- Complete both the logical and physical design of the database against which your application will run
- Create tables, views, and indexes
- Normalize tables, bind application packages, and populate tables with realistic data; ensure that appropriate statistics are available
- Plan to run against a production-size database, so that the application can test representative memory requirements; if this is not possible, try to ensure that the proportions of available system resources to data in the test and production systems are the same (for example, if the test system has 10% of the data, use 10% of the processor time and 10% of the memory that is available to the production system)
- Place database objects in their final disk locations, size log files, determine the location of work files and backup images, and test backup procedures
- Check packages to ensure that performance options, such as row blocking, are enabled when possible

Although the practical limits of an application might be revealed during benchmark testing, the purpose of the benchmark is to measure performance, not to detect defects.

Your benchmark testing program should run in an accurate representation of the final production environment. Ideally, it should run on the same server model with the same memory and disk configurations. This is especially important if the application will ultimately serve large numbers of users and process large amounts of data. The operating system and any communications or storage facilities used directly by the benchmark testing program should also have been tuned previously.

SQL statements to be benchmark tested should be either representative SQL or worst-case SQL, as described in the following list.

Representative SQL

Representative SQL includes those statements that are executed during typical operations of the application that is being benchmark tested. Which statements are selected depends on the nature of the application. For example, a data-entry application might test an INSERT statement, whereas a banking transaction might test a FETCH, an UPDATE, and several INSERT statements.

Worst-case SQL

Statements falling under this category include:

- Statements that are executed frequently
- Statements that are processing high volumes of data
- Statements that are time-critical. For example, statements in an application that runs to retrieve and update customer information while the customer is waiting on the telephone.
- Statements with a large number of joins, or the most complex statements in the application. For example, statements in a banking application that produces summaries of monthly activity for all of a customer's accounts. A common table might list the customer's address and account numbers; however, several other tables must be joined to process and integrate all of the necessary account transaction information.
- Statements that have a poor access path, such as one that is not supported by an available index
- Statements that have a long execution time
- Statements that are executed only at application initialization time, but that have disproportionately large resource requirements. For example, statements in an application that generates a list of account work that must be processed during the day. When the application starts, the first major SQL statement causes a seven-way join, which creates a very large list of all the accounts for which this application user is responsible. This statement might only run a few times each day, but it takes several minutes to run if it has not been tuned properly.

Benchmark test creation

You will need to consider a variety of factors when designing and implementing a benchmark testing program.

Because the main purpose of the testing program is to simulate a user application, the overall structure of the program will vary. You might use the entire application as the benchmark and simply introduce a means for timing the SQL statements that are to be analyzed. For large or complex applications, it might be more practical to include only blocks that contain the important statements. To test the performance of specific SQL statements, you can include only those statements in the benchmark testing program, along with the necessary CONNECT, PREPARE, OPEN, and other statements, as well as a timing mechanism.

Another factor to consider is the type of benchmark to use. One option is to run a set of SQL statements repeatedly over a certain time interval. The number of statements executed over this time interval is a measure of the throughput for the application. Another option is to simply determine the time required to execute individual SQL statements.

For all benchmark testing, you need a reliable and appropriate way to measure elapsed time. To simulate an application in which individual SQL statements execute in isolation, measuring the time to PREPARE, EXECUTE, or OPEN,

FETCH, or CLOSE for each statement might be best. For other applications, measuring the transaction time from the first SQL statement to the COMMIT statement might be more appropriate.

Although the elapsed time for each query is an important factor in performance analysis, it might not necessarily reveal bottlenecks. For example, information on CPU usage, locking, and buffer pool I/O might show that the application is I/O bound and not using the CPU at full capacity. A benchmark testing program should enable you to obtain this kind of data for a more detailed analysis, if needed.

Not all applications send the entire set of rows retrieved from a query to some output device. For example, the result set might be input for another application. Formatting data for screen output usually has a high CPU cost and might not reflect user needs. To provide an accurate simulation, a benchmark testing program should reflect the specific row handling activities of the application. If rows are sent to an output device, inefficient formatting could consume the majority of CPU time and misrepresent the actual performance of the SQL statement itself.

Although it is very easy to use, the Db2® command line processor (CLP) is not suited to benchmarking because of the processing overhead that it adds. A benchmark tool (**db2batch**) is provided in the `bin` subdirectory of your instance `sqllib` directory. This tool can read SQL statements from either a flat file or from standard input, dynamically prepare and execute the statements, and return a result set. It also enables you to control the number of rows that are returned to **db2batch** and the number of rows that are displayed. You can specify the level of performance-related information that is returned, including elapsed time, processor time, buffer pool usage, locking, and other statistics collected from the database monitor. If you are timing a set of SQL statements, **db2batch** also summarizes the performance results and provides both arithmetic and geometric means.

By wrapping **db2batch** invocations in a Perl or Korn shell script, you can easily simulate a multiuser environment. Ensure that connection attributes, such as the isolation level, are the same by selecting the appropriate **db2batch** options.

Note that in partitioned database environments, **db2batch** is suitable only for measuring elapsed time; other information that is returned pertains only to activity on the coordinator database partition.

You can write a driver program to help you with your benchmark testing. On Linux or UNIX systems, a driver program can be written using shell programs. A driver program can execute the benchmark program, pass the appropriate parameters, drive the test through multiple iterations, restore the environment to a consistent state, set up the next test with new parameter values, and collect and consolidate the test results. Driver programs can be flexible enough to run an entire set of benchmark tests, analyze the results, and provide a report of the best parameter values for a given test.

Benchmark test execution

In the most common type of benchmark testing, you choose a configuration parameter and run the test with different values for that parameter until the maximum benefit is achieved.

A single test should include repeated execution of the application (for example, five or ten iterations) with the same parameter value. This enables you to obtain a more reliable average performance value against which to compare the results from other parameter values.

The first run, called a warmup run, should be considered separately from subsequent runs, which are called normal runs. The warmup run includes some startup activities, such as initializing the buffer pool, and consequently, takes somewhat longer to complete than normal runs. The information from a warmup run is not statistically valid. When calculating averages for a specific set of parameter values, use only the results from normal runs. It is often a good idea to drop the high and low values before calculating averages.

For the greatest consistency between runs, ensure that the buffer pool returns to a known state before each new run. Testing can cause the buffer pool to become loaded with data, which can make subsequent runs faster because less disk I/O is required. The buffer pool contents can be forced out by reading other irrelevant data into the buffer pool, or by de-allocating the buffer pool when all database connections are temporarily removed.

After you complete testing with a single set of parameter values, you can change the value of one parameter. Between each iteration, perform the following tasks to restore the benchmark environment to its original state:

- If the catalog statistics were updated for the test, ensure that the same values for the statistics are used for every iteration.
- The test data must be consistent if it is being updated during testing. This can be done by:
  - Using the restore utility to restore the entire database. The backup copy of the database contains its previous state, ready for the next test.
  - Using the import or load utility to restore an exported copy of the data. This method enables you to restore only the data that has been affected. The reorg and runstats utilities should be run against the tables and indexes that contain this data.

In summary, follow these steps to benchmark test a database application:

Step 1 Leave the Db2 registry, database and database manager configuration parameters, and buffer pools at their standard recommended values, which can include:

- Values that are known to be required for proper and error-free application execution
- Values that provided performance improvements during prior tuning
- Values that were suggested by the **AUTOCONFIGURE** command
- Default values; however, these might not be appropriate:
  - For parameters that are significant to the workload and to the objectives of the test
  - For log sizes, which should be determined during unit and system testing of your application
  - For any parameters that must be changed to enable your application to run

Run your set of iterations for this initial case and calculate the average elapsed time, throughput, or processor time. The results should be as consistent as possible, ideally differing by no more than a few percentage

points from run to run. Performance measurements that vary significantly from run to run can make tuning very difficult.

Step 2 Select one and only one method or tuning parameter to be tested, and change its value.

Step 3 Run another set of iterations and calculate the average elapsed time or processor time.

Step 4 Depending on the results of the benchmark test, do one of the following:

- If performance improves, change the value of the same parameter and return to Step 3. Keep changing this parameter until the maximum benefit is shown.
- If performance degrades or remains unchanged, return the parameter to its previous value, return to Step 2, and select a new parameter. Repeat this procedure until all parameters have been tested.

Benchmark test analysis example

Output from a benchmark testing program should include an identifier for each test, iteration numbers, statement numbers, and the elapsed times for each execution.

Note that the data in these sample reports is shown for illustrative purposes only. It does not represent actual measured results.

A summary of benchmark testing results might look like the following:

```
Test     Iter.   Stmt    Timing         SQL Statement
Numbr    Numbr   Numbr   (hh:mm:ss.ss)
 002      05      01     00:00:01.34    CONNECT TO SAMPLE
 002      05      10     00:02:08.15    OPEN cursor_01
 002      05      15     00:00:00.24    FETCH cursor_01
 002      05      15     00:00:00.23    FETCH cursor_01
 002      05      15     00:00:00.28    FETCH cursor_01
 002      05      15     00:00:00.21    FETCH cursor_01
 002      05      15     00:00:00.20    FETCH cursor_01
 002      05      15     00:00:00.22    FETCH cursor_01
 002      05      15     00:00:00.22    FETCH cursor_01
 002      05      20     00:00:00.84    CLOSE cursor_01
 002      05      99     00:00:00.03    CONNECT RESET
```

Figure 1. Sample Benchmark Testing Results

Analysis shows that the CONNECT (statement 01) took 1.34 seconds to complete, the OPEN CURSOR (statement 10) took 2 minutes and 8.15 seconds, the FETCH (statement 15) returned seven rows, with the longest delay being 0.28 seconds, the CLOSE CURSOR (statement 20) took 0.84 seconds, and the CONNECT RESET (statement 99) took 0.03 seconds to complete.

If your program can output data in a delimited ASCII format, the data could later be imported into a database table or a spreadsheet for further statistical analysis.

A summary benchmark report might look like the following:

```
PARAMETER        VALUES FOR EACH BENCHMARK TEST
TEST NUMBER      001     002     003    004     005
locklist         63      63      63      63      63
maxappls         8       8       8       8       8
applheapsz       48      48      48      48      48
dbheap           128     128     128     128     128
sortheap         256     256     256     256     256
maxlocks         22      22      22      22      22
stmtheap         1024    1024    1024    1024    1024
SQL STMT         AVERAGE TIMINGS (seconds)
  01             01.34   01.34   01.35   01.35   01.36
  10             02.15   02.00   01.55   01.24   01.00
  15             00.22   00.22   00.22   00.22   00.22
  20             00.84   00.84   00.84   00.84   00.84
  99             00.03   00.03   00.03   00.03   00.03
```

Figure 2. Sample Benchmark Timings Report

Performance monitoring tools and methodology

Operational monitoring of system performance

Operational monitoring refers to collecting key system performance metrics at periodic intervals over time. This information gives you critical data to refine that initial configuration to be more tailored to your requirements, and also prepares you to address new problems that might appear on their own or following software upgrades, increases in data or user volumes, or new application deployments.

Operational monitoring considerations

An operational monitoring strategy needs to address several considerations.

Operational monitoring needs to be very light weight (not consuming much of the system it is measuring) and generic (keeping a broad "eye" out for potential problems that could appear anywhere in the system).

Because you plan regular collection of operational metrics throughout the life of the system, it is important to have a way to manage all that data. For many of the possible uses you have for your data, such as long-term trending of performance, you want to be able to do comparisons between arbitrary collections of data that are potentially many months apart. The Db2 product itself facilitates this kind of data management very well. Analysis and comparison of monitoring data becomes very straightforward, and you already have a robust infrastructure in place for long-term data storage and organization.

A Db2 database ("Db2") system provides some excellent sources of monitoring data. The primary ones are snapshot monitors and, in Db2 Version 9.5 and later, workload management (WLM) table functions for data aggregation. Both of these focus on summary data, where tools like counters, timers, and histograms maintain running totals of activity in the system. By sampling these monitor elements over time, you can derive the average activity that has taken place between the start and end times, which can be very informative.

There is no reason to limit yourself to just metrics that the Db2 product provides. In fact, data outside of the Db2 software is more than just a nice-to-have. Contextual information is key for performance problem determination. The users,

the application, the operating system, the storage subsystem, and the network - all of these can provide valuable information about system performance. Including metrics from outside of the Db2 database software is an important part of producing a complete overall picture of system performance.

The trend in recent releases of the Db2 database product has been to make more and more monitoring data available through SQL interfaces. This makes management of monitoring data with Db2 very straightforward, because you can easily redirect the data from the administration views, for example, right back into Db2 tables.

For deeper dives, activity event monitor data can also be written to Db2 tables, providing similar benefits. With the vast majority of our monitoring data so easy to store in Db2, a small investment to store system metrics (such as CPU utilization from **vmstat**) in Db2 is manageable as well.

Types of data to collect for operational monitoring

Several types of data are useful to collect for ongoing operational monitoring.

- A basic set of Db2 system performance monitoring metrics.
- Db2 configuration information

  Taking regular copies of database and database manager configuration, Db2 registry variables, and the schema definition helps provide a history of any changes that have been made, and can help to explain changes that arise in monitoring data.
- Overall system load

  If CPU or I/O utilization is allowed to approach saturation, this can create a system bottleneck that might be difficult to detect using just Db2 snapshots. As a result, the best practice is to regularly monitor system load with **vmstat** and **iostat** (and possibly **netstat** for network issues) on Linux and UNIX, and **perfmon** on Windows. You can also use the administrative views, such as ENV_GET_SYSTEM_RESOURCES, to retrieve operating system, CPU, memory, and other information related to the system. Typically you look for changes in what is normal for your system, rather than for specific one-size-fits-all values.
- Throughput and response time measured at the business logic level

  An application view of performance, measured over Db2, at the business logic level, has the advantage of being most relevant to the end user, plus it typically includes everything that could create a bottleneck, such as presentation logic, application servers, web servers, multiple network layers, and so on. This data can be vital to the process of setting or verifying a service level agreement (SLA).

The Db2 system performance monitoring elements and system load data are compact enough that even if they are collected every five to fifteen minutes, the total data volume over time is irrelevant in most systems. Likewise, the overhead of collecting this data is typically in the one to three percent range of additional CPU consumption, which is a small price to pay for a continuous history of important system metrics. Configuration information typically changes relatively rarely, so collecting this once a day is usually frequent enough to be useful without creating an excessive amount of data.

Basic set of system performance monitor elements

11 metrics of system performance provide a good basic set to use in an on-going operational monitoring effort.

There are hundreds of metrics to choose from, but collecting all of them can be counter-productive due to the sheer volume of data produced. You want metrics that are:

- Easy to collect - You do not want to use complex or expensive tools for everyday monitoring, and you do not want the act of monitoring to significantly burden the system.
- Easy to understand - You do not want to look up the meaning of the metric each time you see it.
- Relevant to your system - Not all metrics provide meaningful information in all environments.
- Sensitive, but not too sensitive - A change in the metric should indicate a real change in the system; the metric should not fluctuate on its own.

This starter set includes 11 metrics:

1. The number of transactions executed:
   `TOTAL_APP_COMMITS`

   This provides an excellent base level measurement of system activity.
2. Buffer pool hit ratios, measured separately for data, index, XML storage object, and temporary data:

   Note: The information that follows discusses buffer pools in environments other than Db2 pureScale® environments. Buffer pools work differently in Db2 pureScale environments. For more information, see "Buffer pool monitoring in a Db2 pureScale environment", in the *Database Monitoring Guide and Reference*.

   - Data pages: ((pool_data_lbp_pages_found - pool_async_data_lbp_pages_found) / (pool_data_l_reads + pool_temp_data_l_reads)) × 100
   - Index pages: ((pool_index_lbp_pages_found - pool_async_index_lbp_pages_found) / (pool_index_l_reads + pool_temp_index_l_reads)) × 100
   - XML storage object (XDA) pages: ((pool_xda_lbp_pages_found - pool_async_xda_lbp_pages_found) / (pool_xda_l_reads + pool_temp_xda_l_reads)) × 100

   Buffer pool hit ratios are one of the most fundamental metrics, and give an important overall measure of how effectively the system is exploiting memory to avoid disk I/O. Hit ratios of 80-85% or better for data and 90-95% or better for indexes are generally considered good for an OLTP environment, and of course these ratios can be calculated for individual buffer pools using data from the buffer pool snapshot.

   Note: The formulas shown for hit ratios for data and index pages exclude any read activity by prefetchers.

   Although these metrics are generally useful, for systems such as data warehouses that frequently perform large table scans, data hit ratios are often irretrievably low, because data is read into the buffer pool and then not used again before being evicted to make room for other data.
3. Buffer pool physical reads and writes per transaction:

   Note: The information that follows discusses buffer pools in environments other than Db2 pureScale environments. Buffer pools work differently in Db2 pureScale environments. For more information, see "Buffer pool monitoring in a Db2 pureScale environment", in the *Database Monitoring Guide and Reference*.

```
(POOL_DATA_P_READS + POOL_INDEX_P_READS + POOL_XDA_P_READS
  POOL_TEMP_DATA_P_READS + POOL_TEMP_INDEX_P_READS)
  / TOTAL_APP_COMMITS

(POOL_DATA_WRITES + POOL_INDEX_WRITES + POOL_XDA_WRITES)
  / TOTAL_APP_COMMITS
```

These metrics are closely related to buffer pool hit ratios, but have a slightly different purpose. Although you can consider target values for hit ratios, there are no possible targets for reads and writes per transaction. Why bother with these calculations? Because disk I/O is such a major factor in database performance, it is useful to have multiple ways of looking at it. As well, these calculations include writes, whereas hit ratios only deal with reads. Lastly, in isolation, it is difficult to know, for example, whether a 94% index hit ratio is worth trying to improve. If there are only 100 logical index reads per hour, and 94 of them are in the buffer pool, working to keep those last 6 from turning into physical reads is not a good use of time. However, if a 94% index hit ratio were accompanied by a statistic that each transaction did twenty physical reads (which can be further broken down by data and index, regular and temporary), the buffer pool hit ratios might well deserve some investigation.

The metrics are not just physical reads and writes, but are normalized per transaction. This trend is followed through many of the metrics. The purpose is to decouple metrics from the length of time data was collected, and from whether the system was very busy or less busy at that time. In general, this helps ensure that similar values for metrics are obtained, regardless of how and when monitoring data is collected. Some amount of consistency in the timing and duration of data collection is a good thing; however, normalization reduces it from being critical to being a good idea.

4. The ratio of database rows read to rows selected:

   ```
   ROWS_READ / ROWS_RETURNED
   ```

   This calculation gives an indication of the average number of rows that are read from database tables to find the rows that qualify. Low numbers are an indication of efficiency in locating data, and generally show that indexes are being used effectively. For example, this number can be very high in the case where the system does many table scans, and millions of rows have to be inspected to determine if they qualify for the result set. Alternatively, this statistic can be very low in the case of access to a table through a fully-qualified unique index. Index-only access plans (where no rows need to be read from the table) do not cause ROWS_READ to increase.

   In an OLTP environment, this metric is generally no higher than 2 or 3, indicating that most access is through indexes instead of table scans. This metric is a simple way to monitor plan stability over time - an unexpected increase is often an indication that an index is no longer being used and should be investigated.

5. The amount of time spent sorting per transaction:

   ```
   TOTAL_SORT_TIME / TOTAL_APP_COMMITS
   ```

   This is an efficient way to handle sort statistics, because any extra time due to spilled sorts automatically gets included here. That said, you might also want to collect TOTAL_SORTS and SORT_OVERFLOWS for ease of analysis, especially if your system has a history of sorting issues.

6. The amount of lock wait time accumulated per thousand transactions:

   ```
   1000 * LOCK_WAIT_TIME / TOTAL_APP_COMMITS
   ```

Excessive lock wait time often translates into poor response time, so it is important to monitor. The value is normalized to one thousand transactions because lock wait time on a single transaction is typically quite low. Scaling up to one thousand transactions provides measurements that are easier to handle.

7. The number of deadlocks and lock timeouts per thousand transactions:

   ```
   1000 * (DEADLOCKS + LOCK_TIMEOUTS) / TOTAL_APP_COMMITS
   ```

   Although deadlocks are comparatively rare in most production systems, lock timeouts can be more common. The application usually has to handle them in a similar way: re-executing the transaction from the beginning. Monitoring the rate at which this happens helps avoid the case where many deadlocks or lock timeouts drive significant extra load on the system without the DBA being aware.

8. The number of dirty steal triggers per thousand transactions:

   ```
   1000 * POOL_DRTY_PG_STEAL_CLNS / TOTAL_APP_COMMITS
   ```

   A “dirty steal” is the least preferred way to trigger buffer pool cleaning. Essentially, the processing of an SQL statement that is in need of a new buffer pool page is interrupted while updates on the victim page are written to disk. If dirty steals are allowed to happen frequently, they can have a significant affect on throughput and response time.

9. The number of package cache inserts per thousand transactions:

   ```
   1000 * PKG_CACHE_INSERTS / TOTAL_APP_COMMITS
   ```

   Package cache insertions are part of normal execution of the system; however, in large numbers, they can represent a significant consumer of CPU time. In many well-designed systems, after the system is running at steady-state, very few package cache inserts occur, because the system is using or reusing static SQL or previously prepared dynamic SQL statements. In systems with a high traffic of ad hoc dynamic SQL statements, SQL compilation and package cache inserts are unavoidable. However, this metric is intended to watch for a third type of situation, one in which applications unintentionally cause package cache churn by not reusing prepared statements, or by not using parameter markers in their frequently executed SQL.

10. The time an agent waits for log records to be flushed to disk:

    ```
    LOG_WRITE_TIME
      / TOTAL_APP_COMMITS
    ```

    The transaction log has significant potential to be a system bottleneck, whether due to high levels of activity, or to improper configuration, or other causes. By monitoring log activity, you can detect problems both from the Db2 side (meaning an increase in number of log requests driven by the application) and from the system side (often due to a decrease in log subsystem performance caused by hardware or configuration problems).

11. In partitioned database environments, the number of fast communication manager (FCM) buffers sent and received between partitions:

    ```
    FCM_SENDS_TOTAL, FCM_RECVS_TOTAL
    ```

    These give the rate of flow of data between different partitions in the cluster, and in particular, whether the flow is balanced. Significant differences in the numbers of buffers received from different partitions might indicate a skew in the amount of data that has been hashed to each partition.

Cross-partition monitoring in partitioned database environments

Almost all of the individual monitoring element values mentioned previously are reported on a per-partition basis.

In general, you expect most monitoring statistics to be fairly uniform across all partitions in the same Db2 partition group. Significant differences might indicate data skew. Sample cross-partition comparisons to track include:

- Logical and physical buffer pool reads for data, indexes, and temporary tables
- Rows read, at the partition level and for large tables
- Sort time and sort overflows
- FCM buffer sends and receives
- CPU and I/O utilization

Abnormal values in monitoring data

Being able to identify abnormal values is key to interpreting system performance monitoring data when troubleshooting performance problems.

A monitor element provides a clue to the nature of a performance problem when its value is worse than normal, that is, the value is abnormal. Generally, a worse value is one that is higher than expected, for example higher lock wait time. However, an abnormal value can also be lower than expected, such as lower buffer pool hit ratio. Depending on the situation, you can use one or more methods to determine if a value is worse than normal.

One approach is to rely on industry rules of thumb or best practices. For example, a rule of thumb is that buffer pool hit ratios of 80-85% or better for data are generally considered good for an OLTP environment. Note that this rule of thumb applies to OLTP environments and would not serve as a useful guide for data warehouses where data hit ratios are often much lower due to the nature of the system.

Another approach is to compare current values to baseline values collected previously. This approach is often most definitive and relies on having an adequate operational monitoring strategy to collect and store key performance metrics during normal conditions. For example, you might notice that your current buffer pool hit ratio is 85%. This would be considered normal according to industry norms but abnormal when compared to the 99% value recorded before the performance problem was reported.

A final approach is to compare current values with current values on a comparable system. For example, a current buffer pool hit ratio of 85% would be considered abnormal if comparable systems have a buffer pool ratio of 99%.

The governor utility

The governor monitors the behavior of applications that run against a database and can change that behavior, depending on the rules that you specify in the governor configuration file.

Important: With the new strategic Db2 workload manager features introduced in Db2 Version 9.5, the Db2 governor utility has been deprecated in Version 9.7 and might be removed in a future release. For more information about the deprecation of the governor utility, see “Db2 Governor and Query Patroller have been deprecated”. To learn more about Db2 workload manager and how it replaces the

governor utility, see “Introduction to Db2 workload manager concepts” and “Frequently asked questions about Db2 workload manager”.

A governor instance consists of a frontend utility and one or more daemons. Each instance of the governor is specific to an instance of the database manager. By default, when you start the governor, a governor daemon starts on each database partition of a partitioned database. However, you can specify that a daemon be started on a single database partition that you want to monitor.

The governor manages application transactions according to rules in the governor configuration file. For example, applying a rule might reveal that an application is using too much of a particular resource. The rule would also specify the action to take, such as changing the priority of the application, or forcing it to disconnect from the database.

If the action associated with a rule changes the priority of the application, the governor changes the priority of agents on the database partition where the resource violation occurred. In a partitioned database, if the application is forced to disconnect from the database, the action occurs even if the daemon that detected the violation is running on the coordinator node of the application.

The governor logs any actions that it takes.

Note: When the governor is active, its snapshot requests might affect database manager performance. To improve performance, increase the governor wake-up interval to reduce its CPU usage.

Starting the governor

The governor utility monitors applications that are connected to a database, and changes the behavior of those applications according to rules that you specify in a governor configuration file for that database.

Before you begin

Before you start the governor, you must create a governor configuration file.

To start the governor, you must have SYSADM or SYSCTRL authorization.

About this task

Important: With the workload management features introduced in Db2 Version 9.5, the Db2 governor utility was deprecated in Version 9.7 and might be removed in a future release. It is not supported in Db2 pureScale environments. For more information, see “Db2 Governor and Query Patroller have been deprecated” at http://www.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.wn.doc/doc/i0054901.html.

Procedure

- To start the governor, use the **db2gov** command, specifying the following required parameters:

  START *database-name*
  The database name that you specify must match the name of the database in the governor configuration file.

config-file
The name of the governor configuration file for this database. If the file is not in the default location, which is the `sqllib` directory, you must include the file path as well as the file name.

log-file
The base name of the log file for this governor. For a partitioned database, the database partition number is added for each database partition on which a daemon is running for this instance of the governor.

- To start the governor on a single database partition of a partitioned database, specify the **dbpartitionnum** option.

  For example, to start the governor on database partition 3 of a database named SALES, using a configuration file named `salescfg` and a log file called `saleslog`, enter the following command:

  ```
  db2gov start sales dbpartitionnum 3 salescfg saleslog
  ```

- To start the governor on all database partitions, enter the following command:

  ```
  db2gov start sales salescfg saleslog
  ```

The governor daemon

The governor daemon collects information about applications that run against a database.

The governor daemon runs the following task loop whenever it starts.

1. The daemon checks whether its governor configuration file has changed or has not yet been read. If either condition is true, the daemon reads the rules in the file. This allows you to change the behavior of the governor daemon while it is running.
2. The daemon requests snapshot information about resource use statistics for each application and agent that is working on the database.
3. The daemon checks the statistics for each application against the rules in the governor configuration file. If a rule applies, the governor performs the specified action. The governor compares accumulated information with values that are defined in the configuration file. This means that if the configuration file is updated with new values that an application might have already breached, the rules concerning that breach are applied to the application during the next governor interval.
4. The daemon writes a record in the governor log file for any action that it takes.

When the governor finishes its tasks, it sleeps for an interval that is specified in the configuration file. When that interval elapses, the governor wakes up and begins the task loop again.

If the governor encounters an error or stop signal, it performs cleanup processing before stopping. Using a list of applications whose priorities have been set, cleanup processing resets all application agent priorities. It then resets the priorities of any agents that are no longer working on an application. This ensures that agents do not remain running with non-default priorities after the governor ends. If an error occurs, the governor writes a message to the administration notification log, indicating that it ended abnormally.

The governor cannot be used to adjust agent priorities if the value of the **agentpri** database manager configuration parameter is not the system default.

Although the governor daemon is not a database application, and therefore does not maintain a connection to the database, it does have an instance attachment. Because it can issue snapshot requests, the governor daemon can detect when the database manager ends.

The governor configuration file

The governor configuration file contains rules governing applications that run against a database.

The governor evaluates each rule and takes specified actions when a rule evaluates to true.

The governor configuration file contains general clauses that identify the database to be monitored (required), the interval at which account records containing CPU usage statistics are written, and the sleep interval for governor daemons. The configuration file might also contain one or more optional application monitoring rule statements. The following guidelines apply to both general clauses and rule statements:

- Delimit general comments with braces ({ }).
- In most cases, specify values using uppercase, lowercase, or mixed case characters. The exception is application name (specified following the `applname` clause), which is case sensitive.
- Terminate each general clause or rule statement with a semicolon (;).

If a rule needs to be updated, edit the configuration file without stopping the governor. Each governor daemon detects that the file has changed, and rereads it.

In a partitioned database environment, the governor configuration file must be created in a directory that is mounted across all database partitions so that the governor daemon on each database partition can read the same configuration file.

General clauses

The following clauses cannot be specified more than once in a governor configuration file.

dbname
The name or alias of the database to be monitored. This clause is required.

account *n*
The interval, in minutes, after which account records containing CPU usage statistics for each connection are written. This option is not available on Windows operating systems. On some platforms, CPU statistics are not available from the snapshot monitor. If this is the case, the account clause is ignored.

If a short session occurs entirely within the account interval, no log record is written. When log records are written, they contain CPU statistics that reflect CPU usage since the previous log record for the connection. If the governor is stopped and then restarted, CPU usage might be reflected in two log records; these can be identified through the application IDs in the log records.

interval *n*
The interval, in seconds, after which the daemon wakes up. If you do not specify this clause, the default value of 120 seconds is used.

Rule clauses

Rule statements specify how applications are to be governed, and are assembled from smaller components called rule clauses. If used, rule clauses must appear in a specific order in the rule statement, as follows:

1. **desc**: A comment about the rule, enclosed by single or double quotation marks
2. **time**: The time at which the rule is evaluated
3. **authid**: One or more authorization IDs under which the application executes statements
4. **applname**: The name of the executable or object file that connects to the database. This name is case sensitive. If the application name contains spaces, the name must be enclosed by double quotation marks.
5. **setlimit**: Limits that the governor checks; for example, CPU time, number of rows returned, or idle time. On some platforms, CPU statistics are not available from the snapshot monitor. If this is the case, the setlimit clause is ignored.
6. **action**: The action that is to be taken when a limit is reached. If no action is specified, the governor reduces the priority of agents working for the application by 10 when a limit is reached. Actions that can be taken against an application include reducing its agent priority, forcing it to disconnect from the database, or setting scheduling options for its operations.

Combine the rule clauses to form a rule statement, using a specific clause no more than once in each rule statement.

```
desc "Allow no UOW to run for more than an hour"
setlimit uowtime 3600 action force;
```

If more than one rule applies to an application, all are applied. Usually, the action that is associated with the first limit encountered is the action that is applied first. An exception occurs if you specify a value of -1 for a rule clause: A subsequently specified value for the same clause can only override the previously specified value; other clauses in the previous rule statement are still operative.

For example, one rule statement uses the `rowssel 100000` and `uowtime 3600` clauses to specify that the priority of an application is decreased if its elapsed time is greater than 1 hour or if it selects more than 100 000 rows. A subsequent rule uses the `uowtime -1` clause to specify that the same application can have unlimited elapsed time. In this case, if the application runs for more than 1 hour, its priority is not changed. That is, `uowtime -1` overrides `uowtime 3600`. However, if it selects more than 100 000 rows, its priority is lowered because `rowssel 100000` still applies.

Order of rule application

The governor processes rules from the top of the configuration file to the bottom. However, if the `setlimit` clause in a particular rule statement is more relaxed than the same clause in a preceding rule statement, the more restrictive rule applies. In the following example, ADMIN continues to be limited to 5000 rows, because the first rule is more restrictive.

```
desc "Force anyone who selects 5000 or more rows."
setlimit rowssel 5000 action force;

desc "Allow user admin to select more rows."
authid admin setlimit rowssel 10000 action force;
```

To ensure that a less restrictive rule overrides a more restrictive previous rule, specify -1 to clear the previous rule before applying the new one. For example, in the following configuration file, the initial rule limits all users to 5000 rows. The second rule clears this limit for ADMIN, and the third rule resets the limit for ADMIN to 10000 rows.

```
desc "Force anyone who selects 5000 or more rows."
setlimit rowssel 5000 action force;

desc "Clear the rowssel limit for admin."
authid admin setlimit rowssel -1;

desc "Now set the higher rowssel limit for admin"
authid admin  setlimit rowssel 10000 action force;
```

Example of a governor configuration file

```
{ The database name is SAMPLE; do accounting every 30 minutes;
  wake up once a second. }
dbname sample; account 30; interval 1;

desc "CPU restrictions apply to everyone 24 hours a day."
setlimit cpu 600 rowssel 1000000 rowsread 5000000;

desc "Allow no UOW to run for more than an hour."
setlimit uowtime 3600 action force;

desc 'Slow down a subset of applications.'
applname jointA, jointB, jointC, quryA
setlimit cpu 3 locks 1000 rowssel 500 rowsread 5000;

desc "Have the governor prioritize these 6 long apps in 1 class."
applname longq1, longq2, longq3, longq4, longq5, longq6
setlimit cpu -1
action schedule class;

desc "Schedule all applications run by the planning department."
authid planid1, planid2, planid3, planid4, planid5
setlimit cpu -1
action schedule;

desc "Schedule all CPU hogs in one class, which will control consumption."
setlimit cpu 3600
action schedule class;

desc "Slow down the use of the Db2 CLP by the novice user."
authid novice
applname db2bp.exe
setlimit cpu 5 locks 100 rowssel 250;

desc "During the day, do not let anyone run for more than 10 seconds."
time 8:30 17:00 setlimit cpu 10 action force;

desc "Allow users doing performance tuning to run some of
     their applications during the lunch hour."
time 12:00 13:00 authid ming, geoffrey, john, bill
applname tpcc1, tpcc2, tpcA, tpvG
setlimit cpu 600 rowssel 120000 action force;

desc "Increase the priority of an important application so it always
      completes quickly."
applname V1app setlimit cpu 1 locks 1 rowssel 1 action priority -20;

desc "Some people, such as the database administrator (and others),
     should not be limited. Because this is the last specification
     in the file, it will override what came before."
authid gene, hershel, janet setlimit cpu -1 locks -1 rowssel -1 uowtime -1;
```

Governor rule clauses

Each rule in the governor configuration file is made up of clauses that specify the conditions for applying the rule and the action that results if the rule evaluates to true.

The rule clauses must be specified in the order shown.

Optional beginning clauses

desc Specifies a description for the rule. The description must be enclosed by either single or double quotation marks.

time Specifies the time period during which the rule is to be evaluated. The time period must be specified in the following format: `time hh:mm hh:mm`; for example, `time 8:00 18:00`. If this clause is not specified, the rule is evaluated 24 hours a day.

authid Specifies one or more authorization IDs under which the application is executing. Multiple authorization IDs must be separated by a comma (,); for example: `authid gene, michael, james`. If this clause is not specified, the rule applies to all authorization IDs.

applname
Specifies the name of the executable or object file that is connected to the database. Multiple application names must be separated by a comma (,); for example: `applname db2bp, batch, geneprog`. If this clause is not specified, the rule applies to all application names.

Note:

1. Application names are case sensitive.
2. The database manager truncates all application names to 20 characters. You should ensure that the application that you want to govern is uniquely identified by the first 20 characters of its application name. Application names specified in the governor configuration file are truncated to 20 characters to match their internal representation.

Limit clauses

setlimit
Specifies one or more limits for the governor to check. The limits must be -1 or greater than 0 (for example, `cpu -1 locks 1000 rowssel 10000`). At least one limit must be specified, and any limit that is not specified in a rule statement is not limited by that rule. The governor can check the following limits:

cpu *n* Specifies the number of CPU seconds that can be consumed by an application. If you specify -1, the application's CPU usage is not limited.

idle *n* Specifies the number of idle seconds that are allowed for a connection. If you specify -1, the connection's idle time is not limited.

Note: Some database utilities, such as backup and restore, establish a connection to the database and then perform work through engine dispatchable units (EDUs) that are not visible to the governor. These database connections appear to be idle and might exceed the idle time limit. To prevent the governor from taking action against these utilities, specify -1 for them through the

authorization ID that invoked them. For example, to prevent the governor from taking action against utilities that are running under authorization ID DB2SYS, specify `authid DB2SYS setlimit idle -1`.

locks *n*
Specifies the number of locks that an application can hold. If you specify -1, the number of locks held by the application is not limited.

rowsread *n*
Specifies the number of rows that an application can select. If you specify -1, the number of rows the application can select is not limited. The maximum value that can be specified is 4 294 967 298.

Note: This limit is not the same as `rowssel`. The difference is that `rowsread` is the number of rows that must be read to return the result set. This number includes engine reads of the catalog tables and can be reduced when indexes are used.

rowssel *n*
Specifies the number of rows that can be returned to an application. This value is non-zero only at the coordinator database partition. If you specify -1, the number of rows that can be returned is not limited. The maximum value that can be specified is 4 294 967 298.

uowtime *n*
Specifies the number of seconds that can elapse from the time that a unit of work (UOW) first becomes active. If you specify -1, the elapsed time is not limited.

Note: If you used the sqlmon API to deactivate the unit of work monitor switch or the timestamp monitor switch, this will affect the ability of the governor to govern applications based on the unit of work elapsed time. The governor uses the monitor to collect information about the system. If a unit of work (UOW) of the application has been started before the Governor starts, then the Governor will not govern that UOW.

Action clauses

action Specifies the action that is to be taken if one or more specified limits is exceeded. If a limit is exceeded and the `action` clause is not specified, the governor reduces the priority of agents working for the application by 10.

force Specifies that an application is forced if the `setlimit` is exceeded on any partition where the application is running.

Note: In partitioned database environments, a local application snapshot is used to collect information for `setlimit` evaluation, instead of a global snapshot. If the governor evaluates a `setlimit` on an application's remote partition and determines a limit is exceeded and the force action is performed, the application terminates on all database partitions. For example, if a rule has a "idle 30" `setlimit` and a remote subagent is idle for 40 seconds, the force action terminates the application on all partitions.

forcecoord

Specifies that an application is forced only if `setlimit` is exceeded on the application's coordinating partition.

In partitioned database environments, a local application snapshot is used to collect information for `setlimit` evaluation, instead of a global snapshot. The `forcecoord` action occurs only if `setlimit` values are exceeded on the application's coordinating partition. For example, if a rule has a `setlimit` of "idle 30" and a coordinating agent is idle for 40 seconds, then the `forcecoord` action terminates the application on all partitions. However, no action is performed when only a remote subagent is idle for 40 seconds.

nice *n* Specifies a change to the relative priority of agents working for the application. Valid values range from -20 to +20 on Linux and UNIX, and from -1 to 6 on Windows platforms.

- On Linux and UNIX, the **agentpri** database manager configuration parameter must be set to the default value; otherwise, it overrides the `nice` value.
- On Windows platforms, the **agentpri** database manager configuration parameter and the `nice` value can be used together.

You can use the governor to control the priority of applications that run in the default user service superclass, SYSDEFAULTUSERCLASS. If you use the governor to lower the priority of an application that runs in this service superclass, the agent disassociates itself from its outbound correlator (if it is associated with one) and sets its relative priority according to the agent priority specified by the governor. You cannot use the governor to alter the priority of agents in user-defined service superclasses and subclasses. Instead, you must use the agent priority setting for the service superclass or subclass to control applications that run in these service classes. You can, however, use the governor to force connections in any service class.

Note: On AIX® systems, the instance owner must have the CAP_NUMA_ATTACH capability to raise the relative priority of agents working for the application. To grant this capability, logon as root and run the following command:

```
chuser capabilities=CAP_NUMA_ATTACH,CAP_PROPAGATE <userid>
```

On Solaris 10 or higher, the instance owner must have the proc_priocntl privilege to be able to raise the relative priority of agents working for the application. To grant this privilege, logon as `root` and run the following command:

```
usermod -K defaultpriv=basic,proc_priocntl db2user
```

In this example, proc_priocntl is added to the default privilege set of user db2user.

Moreover, when Db2 is running in a non-global zone of Solaris, the proc_priocntl privilege must be added to the zone's limit privilege set. To grant this privilege to the zone, logon as `root` and run the following command:

```
global# zonecfg -z db2zone
zonecfg:db2zone> set limitpriv="default,proc_priocntl"
```

In this example, proc_priocntl is added to the limit privilege set of zone db2zone.

On Solaris 9, there is no facility for Db2 to raise the relative priority of agents. Upgrade to Solaris 10 or higher to use the ACTION NICE clause of Db2 governor.

schedule [class]

Scheduling improves the priorities of agents working on applications. The goal is to minimize the average response time while maintaining fairness across all applications.

The governor chooses the top applications for scheduling on the basis of the following criteria:

- The application holding the greatest number of locks (an attempt to reduce the number of lock waits)
- The oldest application
- The application with the shortest estimated remaining run time (an attempt to allow as many short-lived statements as possible to complete during the interval)

The top three applications in each criterion are given higher priorities than all other applications. That is, the top application in each criterion group is given the highest priority, the next highest application is given the second highest priority, and the third-highest application is given the third highest priority. If a single application is ranked in the top three for more than one criterion, it is given the appropriate priority for the criterion in which it ranked highest, and the next highest application is given the next highest priority for the other criteria. For example, if application A holds the most locks but has the third shortest estimated remaining run time, it is given the highest priority for the first criterion. The fourth ranked application with the shortest estimated remaining run time is given the third highest priority for that criterion.

The applications that are selected by this governor rule are divided up into three classes. For each class, the governor chooses nine applications, which are the top three applications from each class, based on the criteria described previously. If you specify the `class` option, all applications that are selected by this rule are considered to be a single class, and nine applications are chosen and given higher priorities as described previously.

If an application is selected in more than one governor rule, it is governed by the last rule in which is it selected.

Note: If you used the sqlmon API to deactivate the statement switch, this will affect the ability of the governor to govern applications based on the statement elapsed time. The governor uses the monitor to collect information about the system. If you turn off the switches in the database manager configuration file, they are turned off for the entire instance, and the governor no longer receives this information.
The schedule action can:

- Ensure that applications in different groups get time, without all applications splitting time evenly. For example, if 14 applications (three short, five medium, and six long) are running at the same

time, they might all have poor response times because they are splitting the CPU. The database administrator can set up two groups, medium-length applications and long-length applications. Using priorities, the governor permits all the short applications to run, and ensures that at most three medium and three long applications run simultaneously. To achieve this, the governor configuration file contains one rule for medium-length applications, and another rule for long applications.

The following example shows a portion of a governor configuration file that illustrates this point:

```
desc "Group together medium applications in 1 schedule class."
applname medq1, medq2, medq3, medq4, medq5
setlimit cpu -1
action schedule class;

desc "Group together long applications in 1 schedule class."
applname longq1, longq2, longq3, longq4, longq5, longq6
setlimit cpu -1
action schedule class;
```

- Ensure that each of several user groups (for example, organizational departments) gets equal prioritization. If one group is running a large number of applications, the administrator can ensure that other groups are still able to obtain reasonable response times for their applications. For example, in a case involving three departments (Finance, Inventory, and Planning), all the Finance users could be put into one group, all the Inventory users could be put into a second group, and all the Planning users could be put into a third group. The processing power would be split more or less evenly among the three departments.

  The following example shows a portion of a governor configuration file that illustrates this point:

  ```
  desc "Group together Finance department users."
  authid tom, dick, harry, mo, larry, curly
  setlimit cpu -1
  action schedule class;

  desc "Group together Inventory department users."
  authid pat, chris, jack, jill
  setlimit cpu -1
  action schedule class;

  desc "Group together Planning department users."
  authid tara, dianne, henrietta, maureen, linda, candy
  setlimit cpu -1
  action schedule class;
  ```

- Let the governor schedule all applications.

  If the `class` option is not specified, the governor creates its own classes based on how many active applications fall under the schedule action, and puts applications into different classes based on the query compiler's cost estimate for the query the application is running. The administrator can choose to have all applications scheduled by not qualifying which applications are chosen; that is, by not specifying `applname`, `authid`, or `setlimit` clauses.

Governor log files

Whenever a governor daemon performs an action, it writes a record to its log file.

Actions include the following:

- Starting or stopping the governor
- Reading the governor configuration file
- Changing an application's priority
- Forcing an application
- Encountering an error or warning

Each governor daemon has a separate log file, which prevents file-locking bottlenecks that might result when many governor daemons try to write to the same file simultaneously. To query the governor log files, use the **db2govlg** command.

The log files are stored in the `log` subdirectory of the `sqllib` directory, except on Windows operating systems, where the `log` subdirectory is located under the Common Application Data directory that Windows operating systems use to host application log files. You provide the base name for the log file when you start the governor with the **db2gov** command. Ensure that the log file name contains the database name to distinguish log files on each database partition that is governed. To ensure that the file name is unique for each governor in a partitioned database environment, the number of the database partition on which the governor daemon runs is automatically appended to the log file name.

Log file record format

Each record in the log file has the following format:

```
Date Time DBPartitionNum RecType Message
```

The format of the *Date* and *Time* fields is *yyyy-mm-dd-hh.mm.ss*. You can merge the log files for each database partition by sorting on this field. The *DBPartitionNum* field contains the number of the database partition on which the governor is running.

The *RecType* field contains different values, depending on the type of record being written to the log. The values that can be recorded are:

- ACCOUNT: the application accounting statistics
- ERROR: an error occurred
- FORCE: an application was forced
- NICE: the priority of an application was changed
- READCFG: the governor read the configuration file
- SCHEDGRP: a change in agent priorities occurred
- START: the governor was started
- STOP: the governor was stopped
- WARNING: a warning occurred

Some of these values are described in more detail in the following list.

ACCOUNT
An ACCOUNT record is written in the following situations:

- The value of the **agent_usr_cpu_time** or **agent_sys_cpu_time** monitor element for an application has changed since the last ACCOUNT record was written for this application.
- An application is no longer active.

The ACCOUNT record has the following format:

```
<auth_id> <appl_id> <applname> <connect_time> <agent_usr_cpu_delta>
  <agent_sys_cpu_delta>
```

ERROR
An ERROR record is written when the governor daemon needs to shut down.

FORCE
A FORCE record is written when the governor forces an application, based on rules in the governor configuration file. The FORCE record has the following format:

```
<appl_name> <auth_id> <appl_id> <coord_partition> <cfg_line>
  <restriction_exceeded>
```

where:

coord_partition
Specifies the number of the application's coordinator database partition.

cfg_line
Specifies the line number in the governor configuration file where the rule causing the application to be forced is located.

restriction_exceeded
Provides details about how the rule was violated. Valid values are:

- CPU: the total application USR CPU plus SYS CPU time, in seconds
- Locks: the total number of locks held by the application
- Rowssel: the total number of rows selected by the application
- Rowsread: the total number of rows read by the application
- Idle: the amount of time during which the application was idle
- ET: the elapsed time since the application's current unit of work started (the `uowtime setlimit` was exceeded)

NICE A NICE record is written when the governor changes the priority of an application, based on rules in the governor configuration file. The NICE record has the following format:

```
<appl_name> <auth_id> <appl_id> <nice_value> <cfg_line>
  <restriction_exceeded>
```

where:

nice_value
Specifies the increment or decrement that will be made to the priority value for the application's agent process.

cfg_line
Specifies the line number in the governor configuration file where the rule causing the application's priority to be changed is located.

restriction_exceeded
Provides details about how the rule was violated. Valid values are:

- CPU: the total application USR CPU plus SYS CPU time, in seconds
- Locks: the total number of locks held by the application
- Rowssel: the total number of rows selected by the application
- Rowsread: the total number of rows read by the application
- Idle: the amount of time during which the application was idle
- ET: the elapsed time since the application's current unit of work started (the `uowtime setlimit` was exceeded)

SCHEDGRP

A SCHEDGRP record is written when an application is added to a scheduling group or an application is moved from one scheduling group to another. The SCHEDGRP record has the following format:

```
<appl_name> <auth_id> <appl_id> <cfg_line> <restriction_exceeded>
```

where:

cfg_line
Specifies the line number in the governor configuration file where the rule causing the application to be scheduled is located.

restriction_exceeded
Provides details about how the rule was violated. Valid values are:

- CPU: the total application USR CPU plus SYS CPU time, in seconds
- Locks: the total number of locks held by the application
- Rowssel: the total number of rows selected by the application
- Rowsread: the total number of rows read by the application
- Idle: the amount of time during which the application was idle
- ET: the elapsed time since the application's current unit of work started (the `uowtime setlimit` was exceeded)

START

A START record is written when the governor starts. The START record has the following format:

```
Database = <database_name>
```

STOP A STOP record is written when the governor stops. It has the following format:

```
Database = <database_name>
```

WARNING

A WARNING record is written in the following situations:

- The sqlefrce API was called to force an application, but it returned a positive SQLCODE.
- A snapshot call returned a positive SQLCODE that was not 1611 (SQL1611W).
- A snapshot call returned a negative SQLCODE that was not -1224 (SQL1224N) or -1032 (SQL1032N). These return codes occur when a previously active instance has been stopped.
- On Linux and UNIX, an attempt to install a signal handler has failed.

Because standard values are written, you can query the log files for different types of actions. The *Message* field provides other nonstandard information that depends

on the type of record. For example, a FORCE or NICE record includes application information in the *Message* field, whereas an ERROR record includes an error message.

A governor log file might look like the following example:

```
2007-12-11-14.54.52    0 START      Database = TQTEST
2007-12-11-14.54.52    0 READCFG    Config = /u/db2instance/sqllib/tqtest.cfg
2007-12-11-14.54.53    0 ERROR      SQLMON Error: SQLCode = -1032
2007-12-11-14.54.54    0 ERROR      SQLMONSZ Error: SQLCode = -1032
```

Stopping the governor

The governor utility monitors applications that are connected to a database, and changes the behavior of those applications according to rules that you specify in a governor configuration file for that database.

Before you begin

To stop the governor, you must have SYSADM or SYSCTRL authorization.

About this task

Important: With the workload management features introduced in Db2 Version 9.5, the Db2 governor utility was deprecated in Version 9.7 and might be removed in a future release. It is not supported in Db2 pureScale environments. For more information, see "Db2 Governor and Query Patroller have been deprecated" at http://www.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.wn.doc/doc/i0054901.html.

Procedure

To stop the governor, use the **db2gov** command, specifying the **STOP** parameter.

Example

For example, to stop the governor on all database partitions of the SALES database, enter the following command:

```
db2gov STOP sales
```

To stop the governor on only database partition 3, enter the following command:

```
db2gov START sales nodenum 3
```

Factors affecting performance

System architecture

Db2 architecture and process overview

On the client side, local or remote applications are linked with the Db2 client library. Local clients communicate using shared memory and semaphores; remote clients use a protocol, such as named pipes (NPIPE) or TCP/IP. On the server side, activity is controlled by engine dispatchable units (EDUs).

Figure 3 on page 29 shows a general overview of the Db2 architecture and processes.

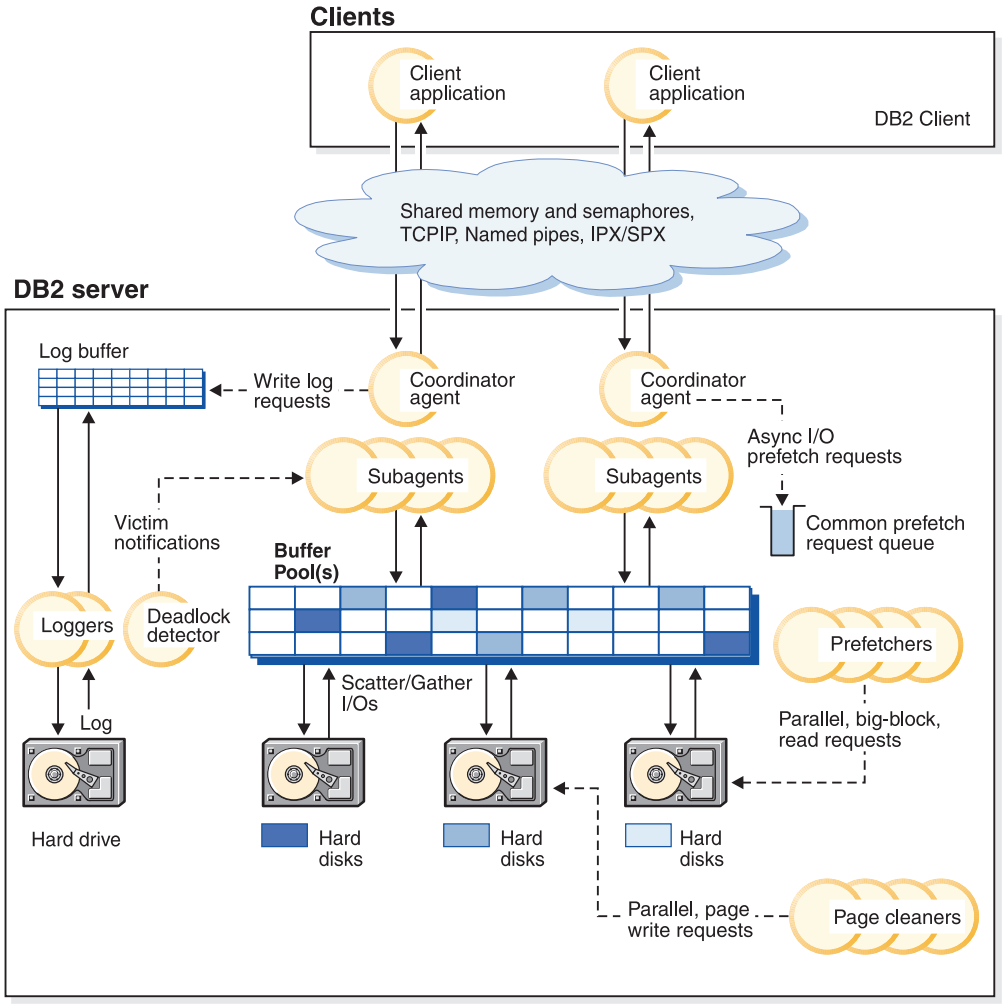


Figure 3. Client connections and database server components

EDUs are shown as circles or groups of circles.

EDUs are implemented as threads on all platforms. Db2 agents are the most common type of EDU. These agents perform most of the SQL and XQuery processing on behalf of applications. Prefetchers and page cleaners are other common EDUs.

A set of subagents might be assigned to process client application requests. Multiple subagents can be assigned if the machine on which the server resides has multiple processors or is part of a partitioned database environment. For example, in a symmetric multiprocessing (SMP) environment, multiple SMP subagents can exploit multiple processors.

All agents and subagents are managed by a pooling algorithm that minimizes the creation and destruction of EDUs.

Buffer pools are areas of database server memory where pages of user data, index data, and catalog data are temporarily moved and can be modified. Buffer pools are a key determinant of database performance, because data can be accessed much faster from memory than from disk.

The configuration of buffer pools, as well as prefetcher and page cleaner EDUs, controls how quickly data can be accessed by applications.

- *Prefetchers* retrieve data from disk and move it into a buffer pool before applications need the data. For example, applications that need to scan through large volumes of data would have to wait for data to be moved from disk into a buffer pool if there were no data prefetchers. Agents of the application send asynchronous read-ahead requests to a common prefetch queue. As prefetchers become available, they implement those requests by using big-block or scatter-read input operations to bring the requested pages from disk into the buffer pool. If you have multiple disks for data storage, the data can be striped across those disks. Striping enables the prefetchers to use multiple disks to retrieve data simultaneously.
- *Page cleaners* move data from a buffer pool back to disk. Page cleaners are background EDUs that are independent of the application agents. They look for pages that have been modified, and write those changed pages out to disk. Page cleaners ensure that there is room in the buffer pool for pages that are being retrieved by prefetchers.

Without the independent prefetchers and page cleaner EDUs, the application agents would have to do all of the reading and writing of data between a buffer pool and disk storage.

The Db2 process model

Knowledge of the Db2 process model helps you to understand how the database manager and its associated components interact. This knowledge can help you to troubleshoot problems that might arise.

The process model that is used by all Db2 database servers facilitates communication between database servers and clients. It also ensures that database applications are isolated from resources, such as database control blocks and critical database files.

The Db2 database server must perform many different tasks, such as processing database application requests or ensuring that log records are written out to disk. Each task is typically performed by a separate *engine dispatchable unit* (EDU).

There are many advantages to using a multithreaded architecture for the Db2 database server. A new thread requires less memory and fewer operating system resources than a process, because some operating system resources can be shared among all threads within the same process. Moreover, on some platforms, the context switch time for threads is less than that for processes, which can improve performance. Using a threaded model on all platforms makes the Db2 database server easier to configure, because it is simpler to allocate more EDUs when needed, and it is possible to dynamically allocate memory that must be shared by multiple EDUs.

For each database accessed, separate EDUs are started to deal with various database tasks such as prefetching, communication, and logging. Database agents are a special class of EDU that are created to handle application requests for a database.

In general, you can rely on the Db2 database server to manage the set of EDUs. However, there are Db2 tools that look at the EDUs. For example, you can use the **db2pd** command with the **-edus** option to list all EDU threads that are active.

Each client application connection has a single coordinator agent that operates on a database. A *coordinator agent* works on behalf of an application, and communicates to other agents by using private memory, interprocess communication (IPC), or remote communication protocols, as needed.

While in a Db2 pureScale instance, these processes are used to monitor the health of Db2 members and/or cluster caching facilities (CFs) running on the host, as well as to distribute the cluster state to all Db2 members and CFs in the instance.

The Db2 architecture provides a firewall so that applications run in a different address space than the Db2 database server (Figure 4). The firewall protects the database and the database manager from applications, stored procedures, and user-defined functions (UDFs). The firewall maintains the integrity of the data in the databases, because it prevents application programming errors from overwriting internal buffers or database manager files. The firewall also improves reliability, because application errors cannot crash the database manager.

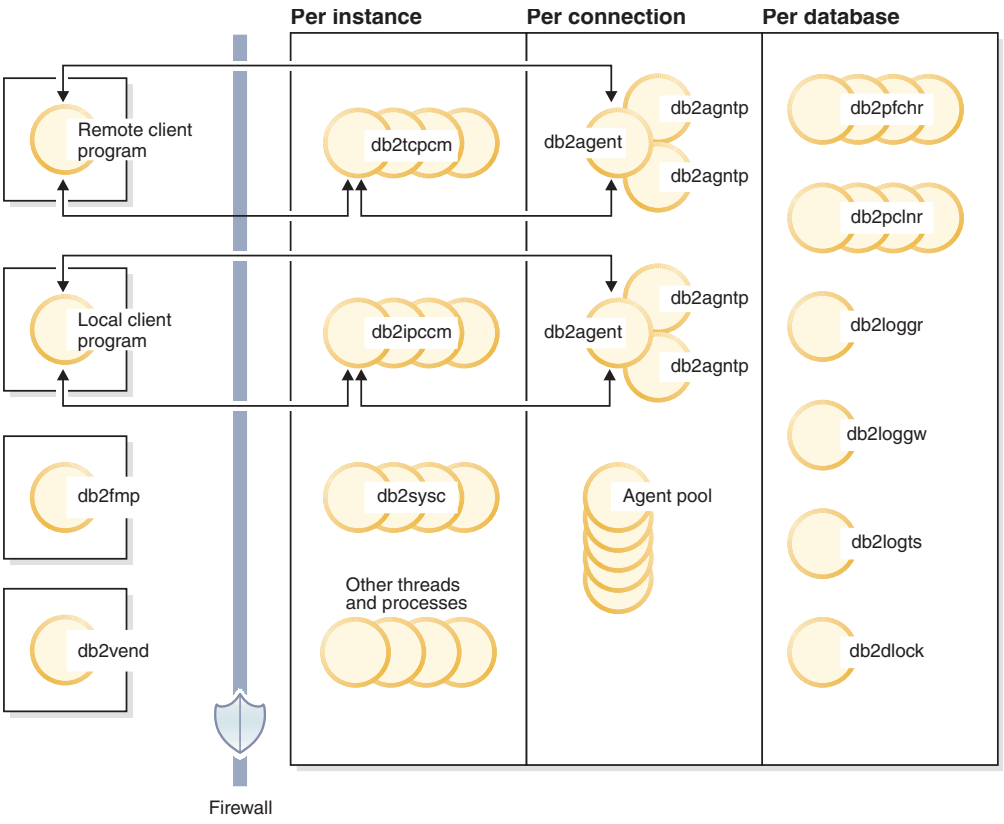


Figure 4. Process model for Db2 database systems

Client programs

Client programs can be remote or local, running on the same machine as the database server. Client programs make first contact with a database through a communication listener.

Listeners

Communication listeners start when the Db2 database server starts. There is a listener for each configured communications protocol, and an interprocess communications (IPC) listener (**db2ipccm**) for local client programs. Listeners include:

- **db2ipccm**, for local client connections
- **db2tcpcm**, for TCP/IP connections
- **db2tcpdm**, for TCP/IP discovery tool requests

Agents

All connection requests from local or remote client programs (applications) are allocated a corresponding coordinator agent (**db2agent**). When the coordinator agent is created, it performs all database requests on behalf of the application.

In partitioned database environments, or systems on which *intraquery parallelism* is enabled, the coordinator agent distributes database requests to subagents (**db2agntp** and **db2agnts**). Subagents that are associated with an application but that are currently idle are named **db2agnta**.

A coordinator agent might be:

- Connected to the database with an alias; for example, **db2agent (DATA1)** is connected to the database alias DATA1.
- Attached to an instance; for example, **db2agent (user1)** is attached to the instance user1.

The Db2 database server instantiates other types of agents, such as independent coordinator agents or subcoordinator agents, to execute specific operations. For example, the independent coordinator agent **db2agnti** is used to run event monitors, and the subcoordinator agent **db2agnsc** is used to parallelize database restart operations after an abnormal shutdown.

A gateway agent (**db2agentg**) is an agent associated to a remote database. It provides indirect connectivity that allows clients to access the host database.

Idle agents reside in an agent pool. These agents are available for requests from coordinator agents that operate on behalf of client programs, or from subagents that operate on behalf of existing coordinator agents. Having an appropriately sized idle agent pool can improve performance when there are significant application workloads. In this case, idle agents can be used as soon as they are required, and there is no need to allocate a new agent for each application connection, which involves creating a thread and allocating and initializing memory and other resources. The Db2 database server automatically manages the size of the idle agent pool.

A pooled agent can be associated to a remote database or a local database. An agent pooled on a remote database is referred to as a pooled gateway agent (**db2agntgp**). An agent pooled on a local database is referred to as a pooled database agent (**db2agntdp**).

db2fmp

The fenced mode process is responsible for executing fenced stored procedures and user-defined functions outside of the firewall. The **db2fmp** process is always a

separate process, but might be multithreaded, depending on the types of routines that it executes.

db2vend

The **db2vend** process is a process to execute vendor code on behalf of an EDU; for example, to execute a user exit program for log archiving (UNIX only).

Database EDUs

The following list includes some of the important EDUs that are used by each database:

- **db2c2c**, the Db2 thread responsible for connecting to cloud products such as IBM® Cloud Product Insights
- **db2dlock**, for deadlock detection. In a partitioned database environment, an additional thread (**db2glock**) is used to coordinate the information that is collected by the **db2dlock** EDU on each partition; **db2glock** runs only on the catalog partition. In a Db2 pureScale environment, a **db2glock** EDU is used to coordinate the information that is collected by the **db2dlock** EDU on each member. A **db2glock** EDU is started on each member, but only one is active.
- **db2fw**, the event monitor fast writer; which is used for high volume, parallel writing of event monitor data to tables, files, or pipes
  - **db2fwx**, an event monitor fast writer thread where "x" identifies the thread number. During database activation the Db2engine sets the number of **db2fwx** threads to a value that is optimal for the performance of event monitors and avoids potential performance problems when different types of workloads are run. The number of **db2fwx** threads equals the number of logical CPUs on the system (for multi-core CPUs, each core counts as one logical CPU). For DPF instances, the number of **db2fwx** threads spawned equals to the number of logical CPUs per member, per database divided by the number of local partitions on the host.
- **db2hadrp**, the high availability disaster recovery (HADR) primary server thread
- **db2hadrs**, the HADR standby server thread
- **db2lfr**, for log file readers that process individual log files
- **db2loggp**, for periodic log work such as determining the recovery window
- **db2loggr**, for manipulating log files to handle transaction processing and recovery
- **db2loggw**, for writing log records to the log files
- **db2logmgr**, for the log manager. Manages log files for a recoverable database.
- **db2logts**, for tracking which table spaces have log records in which log files. This information is recorded in the `DB2TSCHG.HIS` file in the database directory.
- **db2lused**, for updating object usage
- **db2pclnr**, for buffer pool page cleaners
- **db2pcsd**, for autonomic cleanup of the package cache
- **db2pfchr**, for buffer pool prefetchers
- **db2pkgrb**, for the autonomic rebind of invalid packages. During database catalog node startup, **db2pkgrb** makes a single attempt to rebind each invalid package. If auto revalidation is not disabled (ie. if the auto_reval database configuration parameter is not set to DISABLED), then **db2pkgrb** will also attempt to rebind each inoperative package. Afterwards, it shuts down until the next startup. Just before it terminates, a summary message is written at INF level to the `diag.log` when it has finished processing the list. If it is the first rebind attempt after an

upgrade, detailed messages for any failed rebind are written at INF level to the `diag.log`. For all other startups, only the summary message is generated.

- **db2redom**, for the redo master. During recovery, it processes redo log records and assigns log records to redo workers for processing.
- **db2redow**, for the redo workers. During recovery, it processes redo log records at the request of the redo master.
- **db2shred**, for processing individual log records within log pages
- **db2stmm**, for the self-tuning memory management feature
- **db2taskd**, for the distribution of background database tasks. These tasks are executed by threads called **db2taskp**.
- **db2wlmd**, for automatic collection of workload management statistics
- Event monitor threads are identified as follows:
  - db2evm*%1%2* (*%3*)

    where *%1* can be:

    - `g` - global file event monitor
    - `gp` - global piped event monitor
    - `l` - local file event monitor
    - `lp` - local piped event monitor
    - `t` - table event monitor

    and *%2* can be:

    - `i` - coordinator
    - `p` - not coordinator

    and *%3* is the event monitor name
- Backup and restore threads are identified as follows:
  - db2bm.*%1*.*%2* (backup and restore buffer manipulator) and db2med.*%1*.*%2* (backup and restore media controller), where:
    - *%1* is the EDU ID of the agent that controls the backup or restore session
    - *%2* is a sequential value that is used to distinguish among (possibly many) threads that belong to a particular backup or restore session

    For example: **db2bm.13579.2** identifies the second **db2bm** thread that is controlled by the **db2agent** thread with EDU ID 13579.
- The following database EDUs are used for locking in a Db2 pureScale environment:
  - **db2LLMn1**, to process information sent by the global lock manager; there are two of these EDUs on each member, one for the primary CF, and another for the secondary CF
  - **db2LLMn2**, to process information sent by the global lock manager for a special type of lock that is used during database activation and deactivation processing; there are two of these EDUs on each member, one for the primary CF, and another for the secondary CF
  - **db2LLMng**, to ensure that the locks held by this member are released in a timely manner when other members are waiting for these locks
  - **db2LLMrl**, to process the release of locks to the global lock manager
  - **db2LLMrc**, for processing that occurs during database recovery operations and during CF recovery

Database server threads and processes

The system controller (**db2sysc** on UNIX and **db2syscs.exe** on Windows operating systems) must exist if the database server is to function. The following threads and processes carry out various tasks:

- **db2acd**, an autonomic computing daemon that hosts the health monitor, automatic maintenance utilities, and the administrative task scheduler. This process was formerly known as **db2hmon**.
- **db2aiothr**, manages asynchronous I/O requests for a database partition (UNIX only)
- **db2alarm**, notifies EDUs when their requested timer expires (UNIX only)
- **db2disp**, the client connection concentrator dispatcher
- **db2fcms**, the fast communications manager sender daemon
- **db2fcmr**, the fast communications manager receiver daemon
- **db2fmd**, the fault monitor daemon
- **db2licc**, manages installed Db2 licenses
- **db2panic**, the panic agent, which handles urgent requests after agent limits are reached
- **db2pdbc**, the parallel system controller, which handles parallel requests from remote database partitions (used in both partitioned database environments and Db2 pureScale environments)
- **db2resync**, the resync agent that scans the global resync list
- **db2rocm** and **db2rocme**, while in a Db2 pureScale instance, these processes monitor the operational state of Db2 members and cluster caching facilities (CFs) running on each host, as well as to distribute cluster state information to all Db2 members and CFs in the instance.
- **db2sysc**, the main system controller EDU; it handles critical Db2 server events
- **db2sysc** (idle), the Db2 idle processes, which enable the restart light of a guest member on a host quickly and without competing with the resident member for resources.
- **db2thcln**, recycles resources when an EDU terminates (UNIX only)
- **db2wdog**, the watchdog on UNIX and Linux operating systems, which handles abnormal terminations.
- **db2wlmt**, the WLM dispatcher scheduling thread
- **db2wlmtm**, the WLM dispatcher timer thread

Database agents

When an application accesses a database, several processes or threads begin to perform the various application tasks. These tasks include logging, communication, and prefetching. Database agents are threads within the database manager that are used to service application requests. In Version 9.5, agents are run as threads on all platforms.

The maximum number of application connections is controlled by the **max_connections** database manager configuration parameter. The work of each application connection is coordinated by a single worker agent. A *worker agent* carries out application requests but has no permanent attachment to any particular application. *Coordinator agents* exhibit the longest association with an application, because they remain attached to it until the application disconnects. The only exception to this rule occurs when the engine concentrator is enabled, in which case a coordinator agent can terminate that association at transaction boundaries (COMMIT or ROLLBACK).

There are three types of worker agents:

- Idle agents

  This is the simplest form of worker agent. It does not have an outbound connection, and it does not have a local database connection or an instance attachment.

- Active coordinator agents

  Each database connection from a client application has a single active agent that coordinates its work on the database. After the coordinator agent is created, it performs all database requests on behalf of its application, and communicates to other agents using interprocess communication (IPC) or remote communication protocols. Each agent operates with its own private memory and shares database manager and database global resources, such as the buffer pool, with other agents. When a transaction completes, the active coordinator agent might become an inactive agent. When a client disconnects from a database or detaches from an instance, its coordinator agent will be:

  – An active coordinator agent if other connections are waiting
  – Freed and marked as idle if no connections are waiting, and the maximum number of pool agents is being automatically managed or has not been reached
  – Terminated and its storage freed if no connections are waiting, and the maximum number of pool agents has been reached

- Subagents

  The coordinator agent distributes database requests to subagents, and these subagents perform the requests for the application. After the coordinator agent is created, it handles all database requests on behalf of its application by coordinating the subagents that perform requests against the database. In Db2 Version 9.5, subagents can also exist in nonpartitioned environments and in environments where intraquery parallelism is not enabled.

Agents that are not performing work for any application and that are waiting to be assigned are considered to be idle agents and reside in an *agent pool*. These agents are available for requests from coordinator agents operating on behalf of client programs, or for subagents operating on behalf of existing coordinator agents. The number of available agents depends on the value of the **num_poolagents** database manager configuration parameter.

If no idle agents exist when an agent is required, a new agent is created dynamically. Because creating a new agent requires a certain amount of overhead, CONNECT and ATTACH performance is better if an idle agent can be activated for a client.

When a subagent is performing work for an application, it is associated with that application. After it completes the assigned work, it can be placed in the agent pool, but it remains associated with the original application. When the application requests additional work, the database manager first checks the idle pool for associated agents before it creates a new agent.

Database agent management:

Most applications establish a one-to-one relationship between the number of connected applications and the number of application requests that can be processed by the database server. Your environment, however, might require a many-to-one relationship between the number of connected applications and the number of application requests that can be processed.

Two database manager configuration parameters control these factors separately:

- The **max_connections** parameter specifies the maximum number of connected applications
- The **max_coordagents** parameter specifies the maximum number of application requests that can be processed concurrently

The connection concentrator is enabled when the value of **max_connections** is greater than the value of **max_coordagents**. Because each active coordinating agent requires global database resource overhead, the greater the number of these agents, the greater the chance that the upper limits of available global resources will be reached. To prevent this from occurring, set the value of **max_connections** to be higher than the value of **max_coordagents**, or set both parameters to AUTOMATIC.

There are two specific scenarios in which setting these parameters to AUTOMATIC is a good idea:

- If you are confident that your system can handle all of the connections that might be needed, but you want to limit the amount of global resources that are used (by limiting the number of coordinating agents), set **max_connections** to AUTOMATIC. When **max_connections** is greater than **max_coordagents**, the connection concentrator is enabled.
- If you do not want to limit the maximum number of connections or coordinating agents, but you know that your system requires or would benefit from a many-to-one relationship between the number of connected applications and the number of application requests that are processed, set both parameters to AUTOMATIC. When both parameters are set to AUTOMATIC, the database manager uses the values that you specify as an ideal ratio of connections to coordinating agents. Note that both of these parameters can be configured with a starting value and an AUTOMATIC setting. For example, the following command associates both a value of 200 and AUTOMATIC with the **max_coordagents** parameter:`update dbm config using max_coordagents 200 automatic`.

Example

Consider the following scenario:

- The **max_connections** parameter is set to AUTOMATIC and has a current value of 300
- The **max_coordagents** parameter is set to AUTOMATIC and has a current value of 100

The ratio of **max_connections** to **max_coordagents** is 300:100. The database manager creates new coordinating agents as connections come in, and connection concentration is applied only when needed. These settings result in the following actions:

- Connections 1 to 100 create new coordinating agents
- Connections 101 to 300 do not create new coordinating agents; they share the 100 agents that have been created already
- Connections 301 to 400 create new coordinating agents
- Connections 401 to 600 do not create new coordinating agents; they share the 200 agents that have been created already
- and so on...

In this example, it is assumed that the connected applications are driving enough work to warrant creation of new coordinating agents at each step. After some

period of time, if the connected applications are no longer driving sufficient amounts of work, coordinating agents will become inactive and might be terminated.

If the number of connections is reduced, but the amount of work being driven by the remaining connections is high, the number of coordinating agents might not be reduced right away. The **max_connections** and **max_coordagents** parameters do not directly affect agent pooling or agent termination. Normal agent termination rules still apply, meaning that the connections to coordinating agents ratio might not correspond exactly to the values that you specified. Agents might return to the agent pool to be reused before they are terminated.

If finer granularity of control is needed, specify a simpler ratio. For example, the ratio of 300:100 from the previous example can be expressed as 3:1. If **max_connections** is set to 3 (AUTOMATIC) and **max_coordagents** is set to 1 (AUTOMATIC), one coordinating agent can be created for every three connections.

Client-server processing model:

Both local and remote application processes can work with the same database. A remote application is one that initiates a database action from a machine that is remote from the machine on which the database server resides. Local applications are directly attached to the database at the server machine.

How client connections are managed depends on whether the connection concentrator is on or off. The connection concentrator is on whenever the value of the **max_connections** database manager configuration parameter is larger than the value of the **max_coordagents** configuration parameter.

- If the connection concentrator is off, each client application is assigned a unique engine dispatchable unit (EDU) called a *coordinator agent* that coordinates the processing for that application and communicates with it.
- If the connection concentrator is on, each coordinator agent can manage many client connections, one at a time, and might coordinate the other worker agents to do this work. For internet applications with many relatively transient connections, or applications with many relatively small transactions, the connection concentrator improves performance by allowing many more client applications to be connected concurrently. It also reduces system resource use for each connection.

In Figure 5 on page 39, each circle in the Db2 server represents an EDU that is implemented using operating system threads.

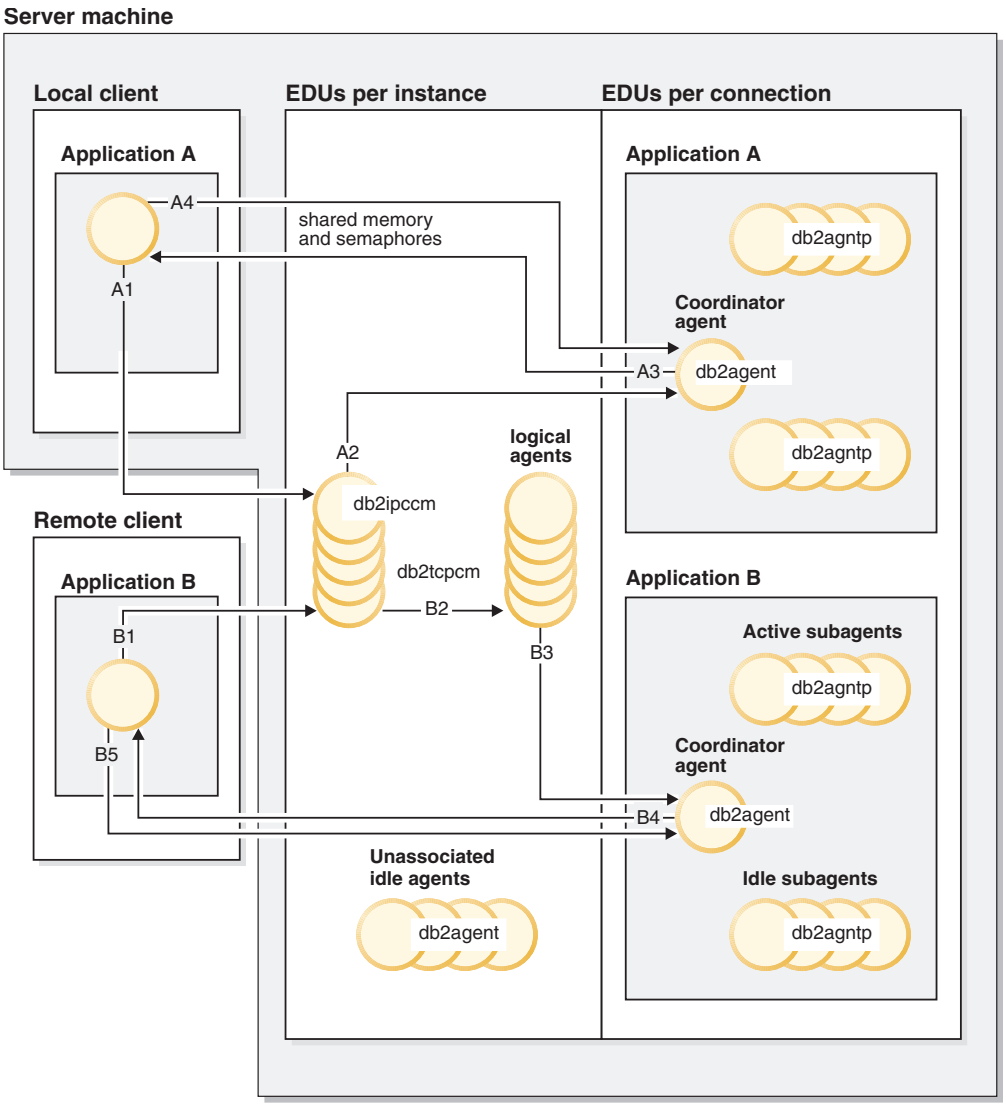


Figure 5. Client-server processing model overview

- At A1, a local client establishes communications through db2ipccm.
- At A2, db2ipccm works with a db2agent EDU, which becomes the coordinator agent for application requests from the local client.
- At A3, the coordinator agent contacts the client application to establish shared memory communications between the client application and the coordinator.
- At A4, the application at the local client connects to the database.
- At B1, a remote client establishes communications through db2tcpcm. If another communications protocol was chosen, the appropriate communications manager is used.
- At B2, db2tcpcm works with a db2agent EDU, which becomes the coordinator agent for the application and passes the connection to this agent.
- At B4, the coordinator agent contacts the remote client application.
- At B5, the remote client application connects to the database.

Note also that:

- Worker agents carry out application requests. There are four types of worker agents: active coordinator agents, active subagents, associated subagents, and idle agents.
- Each client connection is linked to an active coordinator agent.
- In a partitioned database environment, or an environment in which intrapartition parallelism is enabled, the coordinator agents distribute database requests to subagents (db2agntp).
- There is an agent pool (db2agent) where idle agents wait for new work.
- Other EDUs manage client connections, logs, two-phase commit operations, backup and restore operations, and other tasks.

Figure 6 shows additional EDUs that are part of the server machine environment. Each active database has its own shared pool of prefetchers (db2pfchr) and page cleaners (db2pclnr), and its own logger (db2loggr) and deadlock detector (db2dlock).

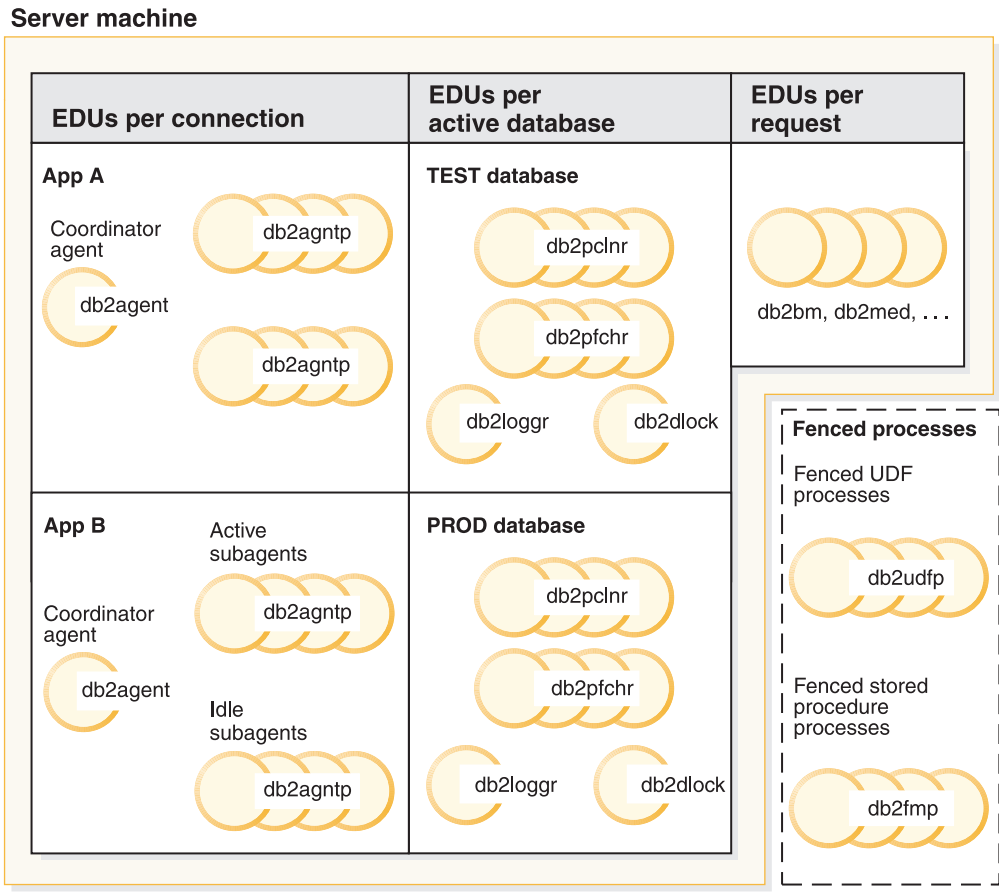


Figure 6. EDUs in the database server

Fenced user-defined functions (UDFs) and stored procedures, which are not shown in the figure, are managed to minimize costs that are associated with their creation and destruction. The default value of the **keepfenced** database manager configuration parameter is YES, which keeps the stored procedure process available for reuse at the next procedure call.

Note: Unfenced UDFs and stored procedures run directly in an agent's address space for better performance. However, because they have unrestricted access to the agent's address space, they must be rigorously tested before being used.

Figure 7 shows the similarities and differences between the single database partition processing model and the multiple database partition processing model.

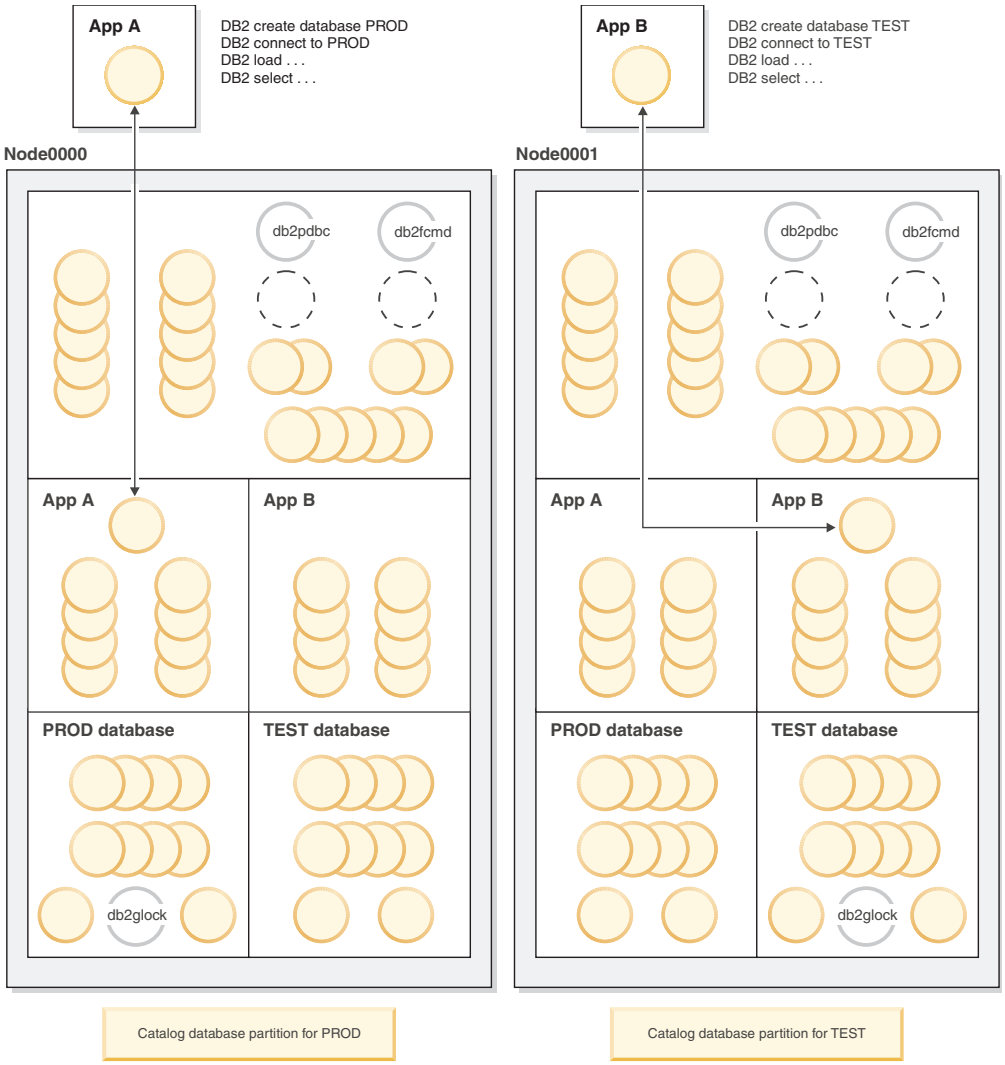


Figure 7. Process model for multiple database partitions

In a multiple database partition environment, the database partition on which the **CREATE DATABASE** command was issued is called the *catalog database partition*. It is on this database partition that the system catalog tables are stored. The system catalog is a repository of all of the information about objects in the database.

As shown in Figure 7, because Application A creates the PROD database on Node0000, the catalog for the PROD database is also created on this database partition. Similarly, because Application B creates the TEST database on Node0001, the catalog for the TEST database is created on this database partition. It is a good idea to create your databases on different database partitions to balance the extra activity that is associated with the catalog for each database across the database partitions in your environment.

There are additional EDUs (db2pdbc and db2fcmd) that are associated with the instance, and these are found on each database partition in a multiple database partition environment. These EDUs are needed to coordinate requests across database partitions and to enable the fast communication manager (FCM).

There is an additional EDU (db2glock) that is associated with the catalog database partition. This EDU controls global deadlocks across the database partitions on which the active database is located.

Each connect request from an application is represented by a connection that is associated with a coordinator agent. The *coordinator agent* is the agent that communicates with the application, receiving requests and sending replies. It can satisfy a request itself or coordinate multiple subagents to work on the request. The database partition on which the coordinator agent resides is called the *coordinator database partition* of that application.

Parts of the database requests from an application are sent by the coordinator database partition to subagents at the other database partitions. All of the results are consolidated at the coordinator database partition before being sent back to the application.

Any number of database partitions can be configured to run on the same machine. This is known as a *multiple logical partition* configuration. Such a configuration is very useful on large symmetric multiprocessor (SMP) machines with very large main memory. In this environment, communications between database partitions can be optimized to use shared memory and semaphores.

Connection-concentrator improvements for client connections:

The connection concentrator improves the performance of applications that have frequent but relatively transient connections by enabling many concurrent client connections to be processed efficiently. It also reduces memory use during each connection and decreases the number of context switches.

The connection concentrator is enabled when the value of the **max_connections** database manager configuration parameter is greater than the value of the **max_coordagents** configuration parameter.

In an environment that requires many simultaneous user connections, you can enable the connection concentrator for more efficient use of system resources. This feature incorporates advantages that were formerly found only in Db2 Connect connection pooling. After the first connection, the connection concentrator reduces the time that is required to connect to a host. When disconnection from a host is requested, the inbound connection is dropped, but the outbound connection to the host is kept within a pool. When a new connection request is received, the database manager attempts to reuse an existing outbound connection from the pool.

For best performance of applications that use connection pooling or the connection concentrator, tune the parameters that control the size of the block of data that is cached. For more information, see the Db2 Connect product documentation.

Examples

- Consider a single-partition database to which, on average, 1000 users are connected simultaneously. At times, the number of connected users might be

higher. The number of concurrent transactions can be as high as 200, but it is never higher than 250. Transactions are short.

For this workload, you could set the following database manager configuration parameters:

- Set **max_coordagents** to 250 to support the maximum number of concurrent transactions.
- Set **max_connections** to AUTOMATIC with a value of 1000 to ensure support for any number of connections; in this example, any value greater than 250 will ensure that the connection concentrator is enabled.
- Leave **num_poolagents** at the default value, which should ensure that database agents are available to service incoming client requests, and that little overhead will result from the creation of new agents.

- Consider a single-partition database to which, on average, 1000 users are connected simultaneously. At times, the number of connected users might reach 2000. An average of 500 users are expected to be executing work at any given time. The number of concurrent transactions is approximately 250. Five hundred coordinating agents would generally be too many; for 1000 connected users, 250 coordinating agents should suffice.

  For this workload, you could update the database manager configuration as follows:

  ```
  update dbm cfg using max_connections 1000 automatic
  update dbm cfg using max_coordagents 250 automatic
  ```

  This means that as the number of connections beyond 1000 increases, additional coordinating agents will be created as needed, with a maximum to be determined by the total number of connections. As the workload increases, the database manager attempts to maintain a relatively stable ratio of connections to coordinating agents.
- Suppose that you do not want to enable the connection concentrator, but you do want to limit the number of connected users. To limit the number of simultaneously connected users to 250, for example, you could set the following database manager configuration parameters:
  - Set **max_coordagents** to 250.
  - Set **max_connections** to 250.
- Suppose that you do not want to enable the connection concentrator, and you do not want to limit the number of connected users. You could update the database manager configuration as follows:

  ```
  update dbm cfg using max_connections automatic
  update dbm cfg using max_coordagents automatic
  ```

Agents in a partitioned database:

In a partitioned database environment, or an environment in which intrapartition parallelism has been enabled, each database partition has its own pool of agents from which subagents are drawn.

Because of this pool, subagents do not have to be created and destroyed each time one is needed or has finished its work. The subagents can remain as associated agents in the pool and can be used by the database manager for new requests from the application with which they are associated or from new applications.

The impact on both performance and memory consumption within the system is strongly related to how your agent pool is tuned. The database manager configuration parameter for agent pool size (**num_poolagents**) affects the total

number of agents and subagents that can be kept associated with applications on a database partition. If the pool size is too small and the pool is full, a subagent disassociates itself from the application it is working on and terminates. Because subagents must be constantly created and reassociated with applications, performance suffers.

By default, **num_poolagents** is set to AUTOMATIC with a value of 100, and the database manager automatically manages the number of idle agents to pool.

If the value of **num_poolagents** is manually set too low, one application could fill the pool with associated subagents. Then, when another application requires a new subagent and has no subagents in its agent pool, it will recycle inactive subagents from the agent pools of other applications. This behavior ensures that resources are fully utilized.

If the value of **num_poolagents** is manually set too high, associated subagents might sit unused in the pool for long periods of time, using database manager resources that are not available for other tasks.

When the connection concentrator is enabled, the value of **num_poolagents** does not necessarily reflect the exact number of agents that might be idle in the pool at any one time. Agents might be needed temporarily to handle higher workload activity.

In addition to database agents, other asynchronous database manager activities run as their own process or thread, including:

- Database I/O servers or I/O prefetchers
- Database asynchronous page cleaners
- Database loggers
- Database deadlock detectors
- Communication and IPC listeners
- Table space container rebalancers

Configuring for good performance

Some types of Db2 deployment, such as the InfoSphere® Balanced Warehouse® (BW), or those within SAP systems, have configurations that are highly specified.

In the BW case, hardware factors, such as the number of CPUs, the ratio of memory to CPU, the number and configuration of disks, and versions are pre-specified, based on thorough testing to determine the optimal configuration. In the SAP case, hardware configuration is not as precisely specified; however, there are a great many sample configurations available. In addition, SAP best practice provides Db2 configuration settings. If you are using a Db2 deployment for a system that provides well-tested configuration guidelines, you should generally take advantage of the guidelines in place of more general rules-of-thumb.

Consider a proposed system for which you do not already have a detailed hardware configuration. Your goal is to identify a few key configuration decisions that get the system well on its way to good performance. This step typically occurs before the system is up and running, so you might have limited knowledge of how it will actually behave. In a way, you have to make a "best guess," based on your knowledge of what the system will be doing.

Hardware configuration

CPU capacity is one of the main independent variables in configuring a system for performance. Because all other hardware configuration typically flows from it, it is not easy to predict how much CPU capacity is required for a given workload. In business intelligence (BI) environments, 200-300 GB of active raw data per processor core is a reasonable estimate. For other environments, a sound approach is to gauge the amount of CPU required, based on one or more existing Db2 systems. For example, if the new system needs to handle 50% more users, each running SQL that is at least as complex as that on an existing system, it would be reasonable to assume that 50% more CPU capacity is required. Likewise, other factors that predict a change in CPU usage, such as different throughput requirements or changes in the use of triggers or referential integrity, should be taken into account as well.

After you have the best idea of CPU requirements (derived from available information), other aspects of hardware configuration start to fall into place. Although you must consider the required system disk capacity in gigabytes or terabytes, the most important factors regarding performance are the capacity in I/Os per second (IOPS), or in megabytes per second of data transfer. In practical terms, this is determined by the number of individual disks involved.

Why is that the case? The evolution of CPUs over the past decade has seen incredible increases in speed, whereas the evolution of disks has been more in terms of their capacity and cost. There have been improvements in disk seek time and transfer rate, but they haven't kept pace with CPU speeds. So to achieve the aggregate performance needed with modern systems, using multiple disks is more important than ever, especially for systems that will drive a significant amount of random disk I/O. Often, the temptation is to use close to the minimum number of disks that can contain the total amount of data in the system, but this generally leads to very poor performance.

In the case of RAID storage, or for individually addressable drives, a rule-of-thumb is to configure at least ten to twenty disks per processor core. For storage servers, a similar number is required. However, in this case, a bit of extra caution is warranted. Allocation of space on storage servers is often done more with an eye to capacity rather than throughput. It is a very good idea to understand the physical layout of database storage, to ensure that the inadvertent overlap of logically separate storage does not occur. For example, a reasonable allocation for a 4-way system might be eight arrays of eight drives each. However, if all eight arrays share the same eight underlying physical drives, the throughput of the configuration would be drastically reduced, compared to eight arrays spread over 64 physical drives.

It is good practice to set aside some dedicated (unshared) disk for the Db2 transaction logs. This is because the I/O characteristics of the logs are very different from Db2 containers, for example, and the competition between log I/O and other types of I/O can result in a logging bottleneck, especially in systems with a high degree of write activity.

In general, a RAID-1 pair of disks can provide enough logging throughput for up to 400 reasonably write-intensive Db2 transactions per second. Greater throughput rates, or high-volume logging (for example, during bulk inserts), requires greater log throughput, which can be provided by additional disks in a RAID-10 configuration, connected to the system through a write-caching disk controller.

Because CPUs and disks effectively operate on different time scales - nanoseconds versus microseconds - you need to decouple them to enable reasonable processing performance. This is where memory comes into play. In a database system, the main purpose of memory is to avoid I/O, and so up to a point, the more memory a system has, the better it can perform. Fortunately, memory costs have dropped significantly over the last several years, and systems with tens to hundreds of gigabytes (GB) of RAM are not uncommon. In general, four to eight gigabytes per processor core should be adequate for most applications.

AIX configuration

There are relatively few AIX parameters that need to be changed to achieve good performance. Again, if there are specific settings already in place for your system (for example, a BW or SAP configuration), those should take precedence over the following general guidelines.

- The VMO parameter **LRU_FILE_REPAGE** should be set to 0. This parameter controls whether AIX victimizes computational pages or file system cache pages. In addition, **minperm** should be set to 3. These are both default values in AIX 6.1.
- The AIO parameter **maxservers** can be initially left at the default value of ten per CPU. After the system is active, **maxservers** is tuned as follows:
  1. Collect the output of the `ps -elfk | grep aio` command and determine if all asynchronous I/O (AIO) kernel processes (aioservers) are consuming the same amount of CPU time.
  2. If they are, **maxservers** might be set too low. Increase **maxservers** by 10%, and repeat step 1.
  3. If some aioservers are using less CPU time than others, the system has at least as many of them as it needs. If more than 10% of aioservers are using less CPU, reduce **maxservers** by 10% and repeat step 1.
- The AIO parameter **maxreqs** should be set to `MAX( NUM_IOCLEANERS x 256, 4096 )`. This parameter controls the maximum number of outstanding AIO requests.
- The hdisk parameter **queue_depth** should be based on the number of physical disks in the array. For example, for IBM disks, the default value for **queue_depth** is 3, and the suggested value would be 3 x *number-of-devices*. This parameter controls the number of queuable disk requests.
- The disk adapter parameter **num_cmd_elems** should be set to the sum of **queue_depth** for all devices connected to the adapter. This parameter controls the number of requests that can be queued to the adapter.

Solaris and HP-UX configuration

For Db2 running on Solaris or HP-UX, the **db2osconf** utility is available to check and suggested kernel parameters based on the system size. The **db2osconf** utility allows you to specify the kernel parameters based on memory and CPU, or with a general scaling factor that compares the current system configuration to an expected future configuration. A good approach is to use a scaling factor of 2 or higher if running large systems, such as SAP applications. In general, **db2osconf** gives you a good initial starting point to configure Solaris and HP-UX, but it does not deliver the optimal value, because it cannot consider current and future workloads.

Linux configuration

The Db2 database manager automatically updates key Linux kernel parameters to satisfy the requirements of a wide variety of configurations.

For more information see “Kernel parameter requirements (Linux)” in *Installing Db2 Servers*

Partitioned database environments

The decision to use partitioned database environments is not generally made based purely on data volume, but more on the basis of the workload. As a general guideline, most partitioned database environments are in the area of data warehousing and business intelligence. Use a partitioned database environment for large complex query environments, because its shared-nothing architecture allows for outstanding scalability.

For smaller data marts (up to about 300 GB), which are unlikely to grow rapidly, a Db2 Enterprise Server Edition configuration is often a good choice. However, large or fast-growing BI environments benefit greatly from a partitioned database environment.

A typical partitioned database system usually has one processor core per data partition. For example, a system with *n* processor cores would likely have the catalog on partition 0, and have *n* additional data partitions. If the catalog partition will be heavily used (for example, to hold single partition dimension tables), it might be allocated a processor core as well. If the system will support very many concurrent active users, two cores per partition might be required.

In terms of a general guide, you should plan on about 250 GB of active raw data per partition.

The InfoSphere Balanced Warehouse documentation contains in-depth information regarding partitioned database configuration best practices. This documentation contains useful information for non-Balanced Warehouse deployments as well.

Choice of code page and collation

As well as affecting database behavior, choice of code page or code set and collating sequence can have a strong impact on performance. The use of Unicode has become very widespread because it allows you to represent a greater variety of character strings in your database than has been the case with traditional single-byte code pages. Unicode is the default for new databases in Db2 Version 9.5. However, because Unicode code sets use multiple bytes to represent some individual characters, there can be increased disk and memory requirements. For example, the UTF-8 code set, which is one of the most common Unicode code sets, uses from one to four bytes per character. An average string expansion factor due to migration from a single-byte code set to UTF-8 is very difficult to estimate because it depends on how frequently multibyte characters are used. For typical North American content, there is usually no expansion. For most western European languages, the use of accented characters typically introduces an expansion of around 10%.

On top of this, the use of Unicode can cause extra CPU consumption relative to single-byte code pages. First, if expansion occurs, the longer strings require more work to manipulate. Second, and more significantly, the algorithms used by the more sophisticated Unicode collating sequences, such as locale-sensitive UCA-based collations, can be much more expensive than the typical SYSTEM collation used with single-byte code pages. This increased expense is due to the

complexity of sorting Unicode strings in a culturally-correct way. Operations that are impacted include sorting, string comparisons, LIKE processing, and index creation.

If Unicode is required to properly represent your data, choose the collating sequence with care.

- If the database will contain data in multiple languages, and correct sort order of that data is of paramount importance, use one of the locale-sensitive UCA-based collations. Depending on the data and the application, this could have a performance overhead of 1.5 to 3 times more, relative to the IDENTITY sequence.
- There are both normalized and non-normalized varieties of locale-sensitive UCA-based collations. Normalized collations have the attribute NO specified and provide additional checks to handle malformed characters. Non-normalized collations have the attribute NX specified and do not provide any such checking. Unless the handling of malformed characters is an issue, use the non-normalized version, because there is a performance benefit in avoiding the normalization code. That said, even non-normalized locale-sensitive UCA-based collations are very expensive.
- If a database is being moved from a single-byte environment to a Unicode environment, but does not have rigorous requirements about hosting a variety of languages (most deployments will be in this category), language aware collation might be appropriate. *Language aware collations* (for example, SYSTEM_819_BE) take advantage of the fact that many Unicode databases contain data in only one language. They use the same lookup table-based collation algorithm as single-byte collations such as SYSTEM_819, and so are very efficient. As a general rule, if the collation behavior in the original single-byte database was acceptable, then as long as the language content does not change significantly following the move to Unicode, culturally aware collation should be considered. This can provide very large performance benefits relative to culturally correct collation.

Physical database design

- In general, file-based database-managed space (DMS) regular table spaces give better performance than system-managed space (SMS) regular table spaces. SMS is often used for temporary table spaces, especially when the temporary tables are very small; however, the performance advantage of SMS in this case is shrinking over time.
- In the past, DMS raw device table spaces had a fairly substantial performance advantage over DMS file table spaces; however, with the introduction of direct I/O (now defaulted through the NO FILE SYSTEM CACHING clause in the CREATE TABLESPACE and the ALTER TABLESPACE statements), DMS file table spaces provide virtually the same performance as DMS raw device table spaces.

Important: Regular table spaces that use system-managed Space (SMS) are deprecated and might be removed in a future release. Catalog table spaces, and temporary table spaces that use system-managed Space are not deprecated but it is suggested that database-managed Spaces (DMS) or Automatic Storage table spaces (AMS) be used instead.

Initial Db2 configuration settings

The Db2 configuration advisor, also known as the **AUTOCONFIGURE** command, takes basic system guidelines that you provide, and determines a good starting set of

Db2 configuration values. The **AUTOCONFIGURE** command can provide real improvements over the default configuration settings, and is suggested as a way to obtain initial configuration values. Some additional fine-tuning of the suggested generated by the **AUTOCONFIGURE** command is often required, based on the characteristics of the system.

Here are some suggestions for using the **AUTOCONFIGURE** command:

- Even though, starting in Db2 Version 9.1, the **AUTOCONFIGURE** command is run automatically at database creation time, it is still a good idea to run the **AUTOCONFIGURE** command explicitly. This is because you then have the ability to specify keyword/value pairs that help customize the results for your system.
- Run (or rerun) the **AUTOCONFIGURE** command after the database is populated with an appropriate amount of active data. This provides the tool with more information about the nature of the database. The amount of data that you use to populate the database is important, because it can affect such things as buffer pool size calculations, for example. Too much or too little data makes these calculations less accurate.
- Try different values for important **AUTOCONFIGURE** command keywords, such as **mem_percent**, **tpm**, and **num_stmts** to get an idea of which, and to what degree, configuration values are affected by these changes.
- If you are experimenting with different keywords and values, use the APPLY NONE option. This gives you a chance to compare the suggestions with the current settings.
- Specify values for all keywords, because the defaults might not suit your system. For example, **mem_percent** defaults to 25%, which is too low for a dedicated Db2 server; 85% is the suggested value in this case.

Db2 autonomics and automatic parameters

Recent releases of Db2 database products have significantly increased the number of parameters that are either automatically set at instance or database startup time, or that are dynamically tuned during operation. For most systems, automatic settings provide better performance than all but the very carefully hand-tuned systems. This is particularly due to the Db2 self-tuning memory manager (STMM), which dynamically tunes total database memory allocation as well as four of the main memory consumers in a Db2 system: the buffer pools, the lock list, the package cache, and the sort heap.

Because these parameters apply on a partition-by-partition basis, using the STMM in a partitioned database environment should be done with some caution. On partitioned database systems, the STMM continuously measures memory requirements on a single partition (automatically chosen by the Db2 system, but that choice can be overridden), and 'pushes out' heap size updates to all partitions on which the STMM is enabled. Because the same values are used on all partitions, the STMM works best in partitioned database environments where the amounts of data, the memory requirements, and the general levels of activity are very uniform across partitions. If a small number of partitions have skewed data volumes or different memory requirements, the STMM should be disabled on those partitions, and allowed to tune the more uniform ones. For example, the STMM should generally be disabled on the catalog partition.

For partitioned database environments with skewed data distribution, where continuous cross-cluster memory tuning is not advised, the STMM can be used selectively and temporarily during a 'tuning phase' to help determine good manual heap settings:

- Enable the STMM on one 'typical' partition. Other partitions continue to have the STMM disabled.
- After memory settings have stabilized, disable the STMM and manually 'harden' the affected parameters at their tuned values.
- Deploy the tuned values on other database partitions with similar data volumes and memory requirements (for example, partitions in the same partition group).
- Repeat the process if there are multiple disjointed sets of database partitions containing similar volumes and types of data and performing similar roles in the system.

The configuration advisor generally chooses to enable autonomic settings where applicable. This includes automatic statistics updates from the **RUNSTATS** command (very useful), but excludes automatic reorganization and automatic backup. These can be very useful as well, but need to be configured according to your environment and schedule for best results.

Explicit configuration settings

Some parameters do not have automatic settings, and are not set by the configuration advisor. These need to be dealt with explicitly. Only parameters that have performance implications are considered here.

- **logpath** or **newlogpath** determines the location of the transaction log. Even the configuration advisor cannot decide for you where the logs should go. As mentioned previously, the most important point is that they should not share disk devices with other Db2 objects, such as table spaces, or be allowed to remain in the default location, which is under the database path. Ideally, transaction logs should be placed on dedicated storage with sufficient throughput capacity to ensure that a bottleneck will not be created.
- **logbufsz** determines the size of the transaction logger internal buffer, in 4-KB pages. The default value of only eight pages is far too small for good performance in a production environment. The configuration advisor always increases it, but possibly not enough, depending on the input parameters. A value of 256-1000 pages is a good general range, and represents only a very small total amount of memory in the overall scheme of a database server.
- **diagpath** determines the location of various useful Db2 diagnostic files. It generally has little impact on performance, except possibly in a partitioned database environment. The default location of **diagpath** on all partitions is typically on a shared, NFS-mounted path. The best practice is to override **diagpath** to a local, non-NFS directory for each partition. This prevents all partitions from trying to update the same file with diagnostic messages. Instead, these are kept local to each partition, and contention is greatly reduced.
- **DB2_PARALLEL_IO** is not a configuration parameter, but a Db2 registry variable. It is very common for Db2 systems to use storage consisting of arrays of disks, which are presented to the operating system as a single device, or to use file systems that span multiple devices. The consequence is that by default, a Db2 database system makes only one prefetch request at a time to a table space container. This is done with the understanding that multiple requests to a single device are serialized anyway. But if a container resides on an array of disks, there is an opportunity to dispatch multiple prefetch requests to it simultaneously, without serialization. This is where **DB2_PARALLEL_IO** comes in. It tells the Db2 system that prefetch requests can be issued to a single container in parallel. The simplest setting is `DB2_PARALLEL_IO=*` (meaning that all containers reside on multiple - assumed in this case to be seven - disks), but other settings also control the degree of parallelism and which table spaces are affected. For

example, if you know that your containers reside on a RAID-5 array of four disks, you might set **DB2_PARALLEL_IO** to *:3. Whether or not particular values benefit performance also depends on the extent size, the RAID segment size, and how many containers use the same set of disks.

Considerations for SAP and other ISV environments

If you are running a Db2 database server for an ISV application such as SAP, some best practice guidelines that take into account the specific application might be available. The most straightforward mechanism is the Db2 registry variable **DB2_WORKLOAD**, which can be set to a value that enables aggregated registry variables to be optimized for specific environments and workloads. Valid settings for **DB2_WORKLOAD** include: 1C, CM, COGNOS_CS, FILENET_CM, MAXIMO, MDM, SAP, TPM, WAS, WC, and WP .

Other suggestions and best practices might apply, such as the choice of a code page or code set and collating sequence, because they must be set to a predetermined value. Refer to the application vendor's documentation for details.

For many ISV applications, such as SAP Business One, the **AUTOCONFIGURE** command can be successfully used to define the initial configuration. However, it should not be used in SAP NetWeaver installations, because an initial set of Db2 configuration parameters is applied during SAP installation. In addition, SAP has a powerful alternative best practices approach (SAP Notes) that describes the preferred Db2 parameter settings; for example, SAP Note 1086130 - DB6: Db2 9.5 Standard Parameter Settings.

Pay special attention to SAP applications when using partitioned database environments. SAP uses partitioned database environment mainly in its SAP NetWeaver Business Intelligence (Business Warehouse) product. The suggested layout has the Db2 system catalog, the dimension and master tables, plus the SAP base tables on Partition 0. This leads to a different workload on this partition compared to other partitioned database environments. Because the SAP application server runs on this partition, up to eight processors might be assigned to just this partition. As the SAP BW workload becomes more highly parallelized, with many short queries running concurrently, the number of partitions for SAP BI is typically smaller than for other applications. In other words, more than one CPU per data partition is required.

Instance configuration

When you start a new Db2 instance, there are a number of steps that you can follow to establish a basic configuration.

- You can use the Configuration Advisor to obtain recommendations for the initial values of the buffer pool size, database configuration parameters, and database manager configuration parameters. To use the Configuration Advisor, specify the **AUTOCONFIGURE** command for an existing database, or specify the **AUTOCONFIGURE** parameter on the **CREATE DATABASE** command. You can display the recommended values or apply them by using the **APPLY** parameter on the **CREATE DATABASE** command. The recommendations are based on input that you provide and system information that the advisor gathers.
- Consult the summary tables (see “Configuration parameters summary”) that list and briefly describe each configuration parameter that is available to the database manager or a database. These summary tables contain a column that indicates whether tuning a particular parameter is likely to produce a high,

medium, low, or no performance change. Use these tables to find the parameters that might help you to realize the largest performance improvements in your environment.

- Use the **ACTIVATE DATABASE** command to activate a database and starts up all necessary database services, so that the database is available for connection and use by any application. In a partitioned database environment, this command activates the database on all database partitions and avoids the startup time that is required to initialize the database when the first application connects.

Table space design

Disk-storage performance factors

Hardware characteristics, such as disk-storage configuration, can strongly influence the performance of your system.

Performance can be affected by one or more of the following aspects of disk-storage configuration:

- Division of storage

  How well you divide a limited amount of storage between indexes and data and among table spaces determines to a large degree how the system will perform in different situations.

- Distribution of disk I/O

  How well you balance the demand for disk I/O across several devices and controllers can affect the speed with which the database manager is able to retrieve data from disk.

- Disk subsystem core performance metrics

  The number of disk operations per second, or the capacity in megabytes transferred per second, has a very strong impact on the performance of the overall system.

Table space impact on query optimization

Certain characteristics of your table spaces can affect the access plans that are chosen by the query compiler.

These characteristics include:

- Container characteristics

  Container characteristics can have a significant impact on the I/O cost that is associated with query execution. When it selects an access plan, the query optimizer considers these I/O costs, including any cost differences when accessing data from different table spaces. Two columns in the SYSCAT.TABLESPACES catalog view are used by the optimizer to help estimate the I/O costs of accessing data from a table space:

  – OVERHEAD provides an estimate of the time (in milliseconds) that is required by the container before any data is read into memory. This overhead activity includes the container's I/O controller overhead as well as the disk latency time, which includes the disk seek time.

    You can use the following formula to estimate the overhead cost:

    ```
    OVERHEAD = average seek time in milliseconds
               + (0.5 * rotational latency)
    ```

    where:

    - 0.5 represents the average overhead of one half rotation

- Rotational latency (in milliseconds) is calculated for each full rotation, as follows:

  ```
  (1 / RPM) * 60 * 1000
  ```

  where:

  - You divide by rotations per minute to get minutes per rotation
  - You multiply by 60 seconds per minute
  - You multiply by 1000 milliseconds per second

  For example, assume that a disk performs 7200 rotations per minute. Using the rotational-latency formula:

  ```
  (1 / 7200) * 60 * 1000 = 8.328 milliseconds
  ```

  This value can be used to estimate the overhead as follows, assuming an average seek time of 11 milliseconds:

  ```
  OVERHEAD = 11 + (0.5 * 8.328)
           = 15.164
  ```

– TRANSFERRATE provides an estimate of the time (in milliseconds) that is required to read one page of data into memory.

  If each table space container is a single physical disk, you can use the following formula to estimate the transfer cost in milliseconds per page:

  ```
  TRANSFERRATE = (1 / spec_rate) * 1000 / 1024000 * page_size
  ```

  where:

  - You divide by *spec_rate*, which represents the disk specification for the transfer rate (in megabytes per second), to get seconds per megabyte
  - You multiply by 1000 milliseconds per second
  - You divide by 1 024 000 bytes per megabyte
  - You multiply by the page size (in bytes); for example, 4096 bytes for a 4-KB page

  For example, suppose that the specification rate for a disk is 3 megabytes per second. Then:

  ```
  TRANSFERRATE = (1 / 3) * 1000 / 1024000 * 4096
               = 1.333248
  ```

  or about 1.3 milliseconds per page.

  If the table space containers are not single physical disks, but are arrays of disks (such as RAID), you must take additional considerations into account when estimating the TRANSFERRATE.

  If the array is relatively small, you can multiply the *spec_rate* by the number of disks, assuming that the bottleneck is at the disk level. However, if the array is large, the bottleneck might not be at the disk level, but at one of the other I/O subsystem components, such as disk controllers, I/O busses, or the system bus. In this case, you cannot assume that the I/O throughput capacity is the product of the *spec_rate* and the number of disks. Instead, you must measure the actual I/O rate (in megabytes) during a sequential scan. For example, a sequential scan resulting from `select count(*) from big_table` could be several megabytes in size. In this case, divide the result by the number of containers that make up the table space in which BIG_TABLE resides. Use this result as a substitute for *spec_rate* in the formula given previously. For example, a measured sequential I/O rate of 100 megabytes

while scanning a table in a four-container table space would imply 25 megabytes per container, or a TRANSFERRATE of (1 / 25) * 1000 / 1 024 000 * 4096 = 0.16 milliseconds per page.

Containers that are assigned to a table space might reside on different physical disks. For best results, all physical disks that are used for a given table space should have the same OVERHEAD and TRANSFERRATE characteristics. If these characteristics are not the same, you should use average values when setting OVERHEAD and TRANSFERRATE.

You can obtain media-specific values for these columns from hardware specifications or through experimentation. These values can be specified on the CREATE TABLESPACE and ALTER TABLESPACE statements.

- Prefetching

  When considering the I/O cost of accessing data in a table space, the optimizer also considers the potential impact that prefetching data and index pages from disk can have on query performance. Prefetching can reduce the overhead that is associated with reading data into the buffer pool.

  The optimizer uses information from the PREFETCHSIZE and EXTENTSIZE columns of the SYSCAT.TABLESPACES catalog view to estimate the amount of prefetching that will occur.

  – EXTENTSIZE can only be set when creating a table space. An extent size of 4 or 8 pages is usually sufficient.
  – PREFETCHSIZE can be set when you create or alter a table space. The default prefetch size is determined by the value of the **dft_prefetch_sz** database configuration parameter. Review the recommendations for sizing this parameter and make changes as needed, or set it to AUTOMATIC.

After making changes to your table spaces, consider executing the runstats utility to collect the latest statistics about indexes and to ensure that the query optimizer chooses the best possible data-access plans before rebinding your applications.

Database design

Tables

Table and index management for standard tables:

In standard tables, data is logically organized as a list of data pages. These data pages are logically grouped together based on the extent size of the table space.

For example, if the extent size is four, pages zero to three are part of the first extent, pages four to seven are part of the second extent, and so on.

The number of records contained within each data page can vary, based on the size of the data page and the size of the records. Most pages contain only user records. However, a small number of pages include special internal records that are used by the data server to manage the table. For example, in a standard table, there is a free space control record (FSCR) on every 500th data page (Figure 8 on page 55). These records map the free space that is available for new records on each of the following 500 data pages (until the next FSCR).

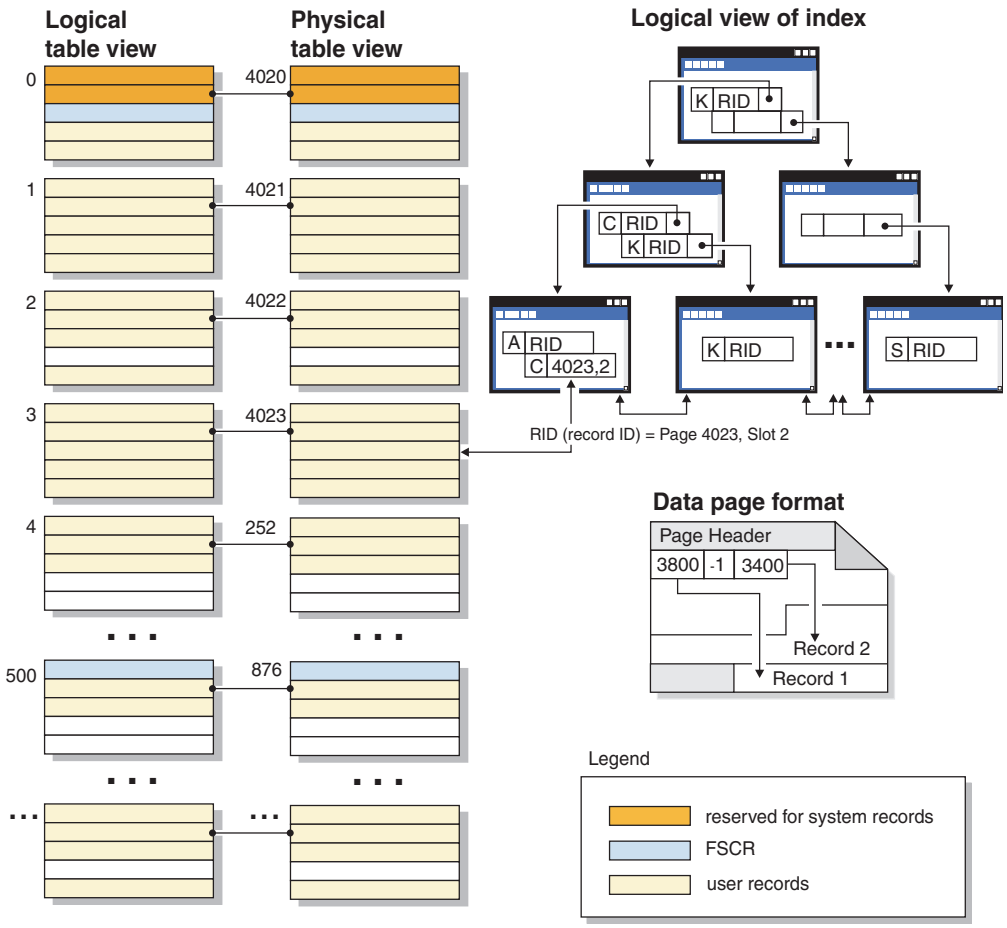


Figure 8. Logical table, record, and index structure for standard tables

Logically, index pages are organized as a B-tree that can efficiently locate table records that have a specific key value. The number of entities on an index page is not fixed, but depends on the size of the key. For tables in database managed space (DMS) table spaces, record identifiers (RIDs) in the index pages use table space-relative page numbers, not object-relative page numbers. This enables an index scan to directly access the data pages without requiring an extent map page (EMP) for mapping.

Each data page has the same format. A page begins with a page header; this is followed by a slot directory. Each entry in the slot directory corresponds to a different record on the page. An entry in the slot directory represents the byte-offset on the data page where a record begins. Entries of -1 correspond to deleted records.

Record identifiers and pages

Record identifiers consist of a page number followed by a slot number (Figure 9 on page 56). Index records contain an additional field called the ridFlag. The ridFlag stores information about the status of keys in the index, such as whether they have been marked deleted. After the index is used to identify a RID, the RID is used to identify the correct data page and slot number on that page. After a record is assigned a RID, the RID does not change until the table is reorganized.

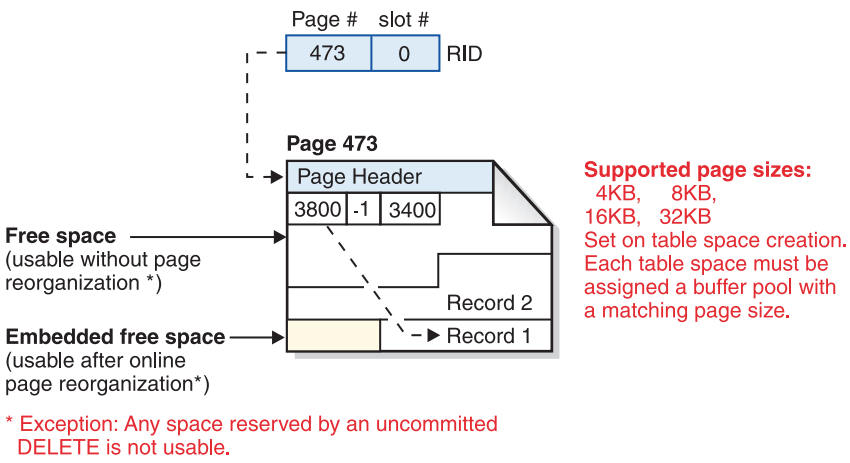


Figure 9. Data page and record ID (RID) format

When a table page is reorganized, embedded free space that is left on the page after a record is physically deleted is converted to usable free space.

The Db2 data server supports different page sizes. Use larger page sizes for workloads that tend to access rows sequentially. For example, sequential access is commonly used for decision support applications, or when temporary tables are being used extensively. Use smaller page sizes for workloads that tend to access rows randomly. For example, random access is often used in online transaction processing (OLTP) environments.

Index management in standard tables

Db2 indexes use an optimized B-tree implementation that is based on an efficient and high concurrency index management method using write-ahead logging. A B-tree index is arranged as a balanced hierarchy of pages that minimizes access time by realigning data keys as items are inserted or deleted.

The optimized B-tree implementation has bidirectional pointers on the leaf pages that allow a single index to support scans in either forward or reverse direction. Index pages are usually split in half, except at the high-key page where a 90/10 split is used, meaning that the highest ten percent of index keys are placed on a new page. This type of index page split is useful for workloads in which insert operations are often completed with new high-key values.

Deleted index keys are removed from an index page only if there is an X lock on the table. If keys cannot be removed immediately, they are marked deleted and physically removed later.

If you enabled online index defragmentation by specifying a positive value for MINPCTUSED when the index was created, index leaf pages can be merged online. MINPCTUSED represents the minimum percentage of used space on an index leaf page. If the amount of used space on an index page becomes lower than this value after a key is removed, the database manager attempts to merge the remaining keys with those of a neighboring page. If there is sufficient room, the merge is performed and an index leaf page is deleted. Because online defragmentation occurs only when keys are removed from an index page, this does not occur if keys are merely marked deleted, but have not been physically

removed from the page. Online index defragmentation can improve space reuse, but if the MINPCTUSED value is too high, the time that is needed for a merge increases, and a successful merge becomes less likely. The recommended value for MINPCTUSED is fifty percent or less.

The INCLUDE clause of the CREATE INDEX statement lets you specify one or more columns (beyond the key columns) for the index leaf pages. These include columns, which are not involved in ordering operations against the index B-tree, can increase the number of queries that are eligible for index-only access. However, they can also increase index space requirements and, possibly, index maintenance costs if the included columns are updated frequently. The maintenance cost of updating include columns is less than the cost of updating key columns, but more than the cost of updating columns that are not part of an index.

Table and index management for MDC and ITC tables:

Table and index organization for multidimensional (MDC) and insert time clustering (ITC) tables is based on the same logical structures as standard table organization.

Like standard tables, MDC and ITC tables are organized into pages that contain rows of data divided into columns. The rows on each page are identified by record IDs (RIDs). However, the pages for MDC and ITC tables are grouped into extent-sized blocks. For example, Figure 10 on page 58, shows a table with an extent size of four. The first four pages, numbered 0 through 3, represent the first block in the table. The next four pages, numbered 4 through 7, represent the second block in the table.

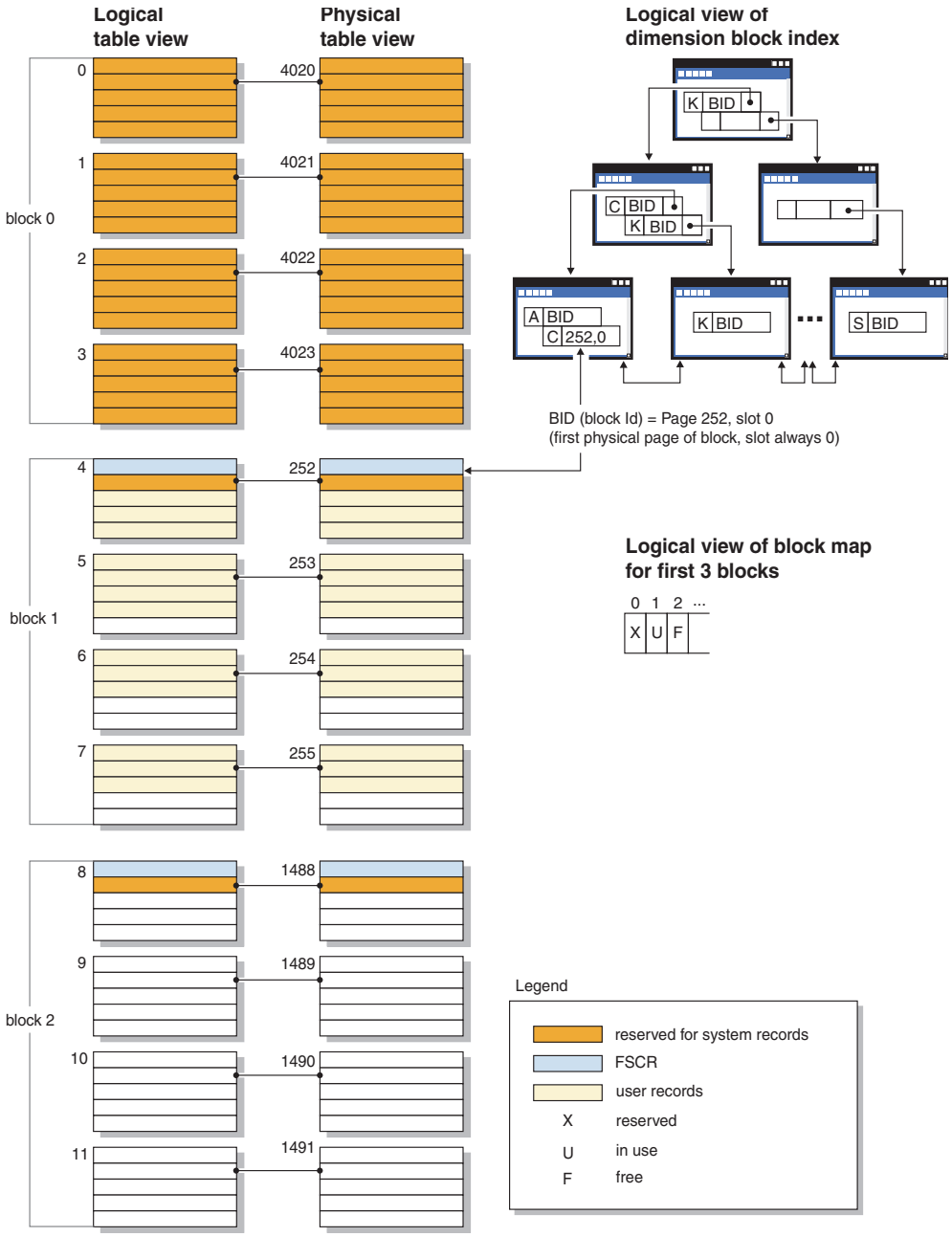


Figure 10. Logical table, record, and index structure for MDC and ITC tables

The first block contains special internal records, including the free space control record (FSCR), that are used by the Db2 server to manage the table. In subsequent blocks, the first page contains the FSCR. An FSCR maps the free space for new records that exists on each page of the block. This available free space is used when inserting records into the table.

As the name implies, MDC tables cluster data on more than one dimension. Each dimension is determined by a column or set of columns that you specify in the ORGANIZE BY DIMENSIONS clause of the CREATE TABLE statement. When you create an MDC table, the following two indexes are created automatically:

- A dimension-block index, which contains pointers to each occupied block for a single dimension

- A composite-block index, which contains all dimension key columns, and which is used to maintain clustering during insert and update activity

The optimizer considers access plans that use dimension-block indexes when it determines the most efficient access plan for a particular query. When queries have predicates on dimension values, the optimizer can use the dimension-block index to identify-and fetch from-the extents that contain these values. Because extents are physically contiguous pages on disk, this minimizes I/O and leads to better performance.

You can also create specific RID indexes if analysis of data access plans indicates that such indexes would improve query performance.

As the name implies, ITC tables cluster data based on row insert time. The differences between MDC and ITC tables are:

- block indexes are not used for any data access,
- only a single composite block index is created for the table, and that index consists of a virtual dimension, and
- the index is never chosen by the optimizer for plans because the column it contains cannot be referenced by any SQL statement.

MDC and ITC tables can have their empty blocks released to the table space.

Indexes

Index structure:

The database manager uses a B+ tree structure for index storage.

A B+ tree has several levels, as shown in Figure 11 on page 60; “rid” refers to a record ID (RID).

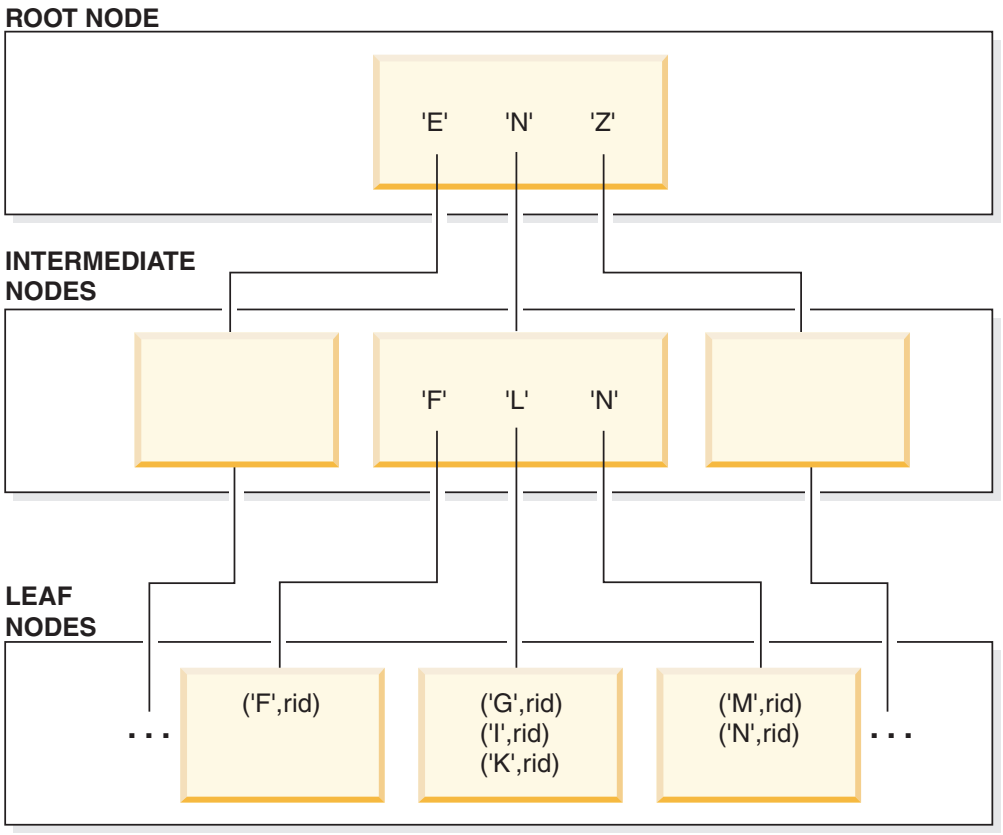


Figure 11. Structure of a B+ Tree Index

The top level is known as the *root node*. The bottom level consists of *leaf nodes* that store index key values with pointers to the table rows that contain the corresponding data. Levels between the root and leaf node levels are known as *intermediate nodes*.

When it looks for a particular index key value, the index manager searches the index tree, starting at the root node. The root node contains one key for each (intermediate) node in the next level. The value of each of these keys is the largest existing key value for the corresponding node at the next level. For example, suppose that an index has three levels, as shown in the figure. To find a particular index key value, the index manager searches the root node for the first key value that is greater than or equal to the search key value. The root node key points to a specific intermediate node. The index manager follows this procedure through each intermediate node until it finds the leaf node that contains the index key that it needs.

Suppose that the key being looked for in Figure 11 is “I”. The first key in the root node that is greater than or equal to “I” is “N”, which points to the middle node at the next level. The first key in that intermediate node that is greater than or equal to “I” is “L”, which, in turn, points to a specific leaf node on which the index key for “I” and its corresponding RID can be found. The RID identifies the corresponding row in the base table.

The leaf node level can also contain pointers to previous leaf nodes. These pointers enable the index manager to scan across leaf nodes in either direction to retrieve a

range of values after it finds one value in the range. The ability to scan in either direction is possible only if the index was created with the ALLOW REVERSE SCANS option.

In the case of a multidimensional clustering (MDC) or insert time clustering (ITC) table, a block index is created automatically for each clustering dimension that you specify for the table. A composite block index is also created; this index contains a key part for each column that is involved in any dimension of the table. Such indexes contain pointers to block IDs (BIDs) instead of RIDs, and provide data-access improvements.

A one-byte *ridFlag*, stored for each RID on the leaf page of an index, is used to mark the RID as logically deleted, so that it can be physically removed later. After an update or delete operation commits, the keys that are marked as deleted can be removed. For each variable-length column in the index, two additional bytes store the actual length of the column value.

Index cleanup and maintenance:

After you create an index, performance might degrade with time unless you keep the index compact and organized.

The following recommendations help you keep indexes as small and efficient as possible:

- Enable online index defragmentation.

  Create indexes with the MINPCTUSED clause. Drop and recreate existing indexes, if necessary.
- Perform frequent commits, or acquire table-level X locks, either explicitly or through lock escalation, if frequent commits are not possible.

  Index keys that are marked deleted can be physically removed from the table after a commit. X locks on tables enable the deleted keys to be physically removed when they are marked deleted.
- Use the **REORGCHK** command to help determine when to reorganize indexes or tables, and when to use the **REORG INDEXES** command with the **CLEANUP** parameter.

  To allow read and write access to the index during reorganization, use the **REORG INDEXES** command with the **ALLOW WRITE ACCESS** option.

  With Db2 Version 9.7 Fix Pack 1 and later releases, issue the **REORG INDEXES** command with the **ON DATA PARTITION** parameter on a data partitioned table to reorganize the partitioned indexes of the specified partition. During index reorganization, the unaffected partitions remain read and write accessible access is restricted only to the affected partition.
- If the objective is to reclaim space, use the **REORG INDEXES** command with the **CLEANUP** and **RECLAIM EXTENTS** parameters. The **CLEANUP** parameter maximizes the amount of reclaimable space.

  The RECLAIMABLE_SPACE output of the ADMIN_GET_INDEX_INFO function shows how much space is reclaimable, in kilobytes.

Index keys that are marked deleted are cleaned up:

- During subsequent insert, update, or delete activity

  During key insertion, keys that are marked deleted and that are known to have been committed are cleaned up if that might avoid the need to perform a page split and prevent the index from increasing in size.

During key deletion, when all keys on a page have been marked deleted, an attempt is made to find another index page where all the keys are marked deleted and all those deletions have committed. If such a page is found, it is deleted from the index tree. If there is an X lock on the table when a key is deleted, the key is physically deleted instead of just being marked deleted. During physical deletion, any deleted keys on the same page are also removed if they are marked deleted and known to be committed.

- When you execute the **REORG INDEXES** command with the **CLEANUP** parameter:

  The **CLEANUP PAGES** option searches for and frees index pages on which all keys are marked deleted and known to be committed.

  The **CLEANUP ALL** option frees not only index pages on which all keys are marked deleted and known to be committed, but it also removes record identifiers (RIDs) that are marked deleted and known to be committed from pages that contain some non-deleted RIDs. This option also tries to merge adjacent leaf pages if doing so results in a merged leaf page that has at least PCTFREE free space. The PCTFREE value is defined when an index is created. The default PCTFREE value is 10 percent. If two pages can be merged, one of the pages is freed.

  For data partitioned tables, it is recommended that you invoke the **RUNSTATS** command after an asynchronous index cleanup has completed. To determine whether there are detached data partitions in the table, query the STATUS field in the SYSCAT.DATAPARTITIONS catalog view and look for the value 'L' (logically detached), 'D' (detached partition having detach dependent tables such as a materialized query tables) or 'I' (index cleanup).

- When an index is rebuilt (or, in the case of data partitioned indexes, when an index partition is rebuilt)

  Utilities that rebuild indexes include the following:

  – **REORG INDEXES** with the default **REBUILD** parameter
  – **REORG TABLE** without the **INPLACE** or **RECLAIM EXTENTS** parameter
  – **IMPORT** with the **REPLACE** parameter
  – **LOAD** with the **INDEXING MODE REBUILD** parameter

Asynchronous index cleanup:

Asynchronous index cleanup (AIC) is the deferred cleanup of indexes following operations that invalidate index entries. Depending on the type of index, the entries can be record identifiers (RIDs) or block identifiers (BIDs). Invalid index entries are removed by index cleaners, which operate asynchronously in the background.

AIC accelerates the process of detaching a data partition from a partitioned table, and is initiated if the partitioned table contains one or more nonpartitioned indexes. In this case, AIC removes all nonpartitioned index entries that refer to the detached data partition, and any pseudo-deleted entries. After all of the indexes have been cleaned, the identifier that is associated with the detached data partition is removed from the system catalog. In Db2 Version 9.7 Fix Pack 1 and later releases, AIC is initiated by an asynchronous partition detach task.

Prior to Db2 Version 9.7 Fix Pack 1, if the partitioned table has dependent materialized query tables (MQTs), AIC is not initiated until after a SET INTEGRITY statement is executed.

Normal table access is maintained while AIC is in progress. Queries accessing the indexes ignore any invalid entries that have not yet been cleaned.

In most cases, one cleaner is started for each nonpartitioned index that is associated with the partitioned table. An internal task distribution daemon is responsible for distributing the AIC tasks to the appropriate table partitions and assigning database agents. The distribution daemon and cleaner agents are internal system applications that appear in **LIST APPLICATIONS** command output with the application names **db2taskd** and **db2aic**, respectively. To prevent accidental disruption, system applications cannot be forced. The distribution daemon remains online as long as the database is active. The cleaners remain active until cleaning has been completed. If the database is deactivated while cleaning is in progress, AIC resumes when you reactivate the database.

AIC impact on performance

AIC incurs minimal performance impact.

An instantaneous row lock test is required to determine whether a pseudo-deleted entry has been committed. However, because the lock is never acquired, concurrency is unaffected.

Each cleaner acquires a minimal table space lock (IX) and a table lock (IS). These locks are released if a cleaner determines that other applications are waiting for locks. If this occurs, the cleaner suspends processing for 5 minutes.

Cleaners are integrated with the utility throttling facility. By default, each cleaner has a utility impact priority of 50. You can change the priority by using the **SET UTIL_IMPACT_PRIORITY** command or the db2UtilityControl API.

Monitoring AIC

You can monitor AIC with the **LIST UTILITIES** command. Each index cleaner appears as a separate utility in the output. The following is an example of output from the **LIST UTILITIES SHOW DETAIL** command:

```
ID                                = 2
Type                              = ASYNCHRONOUS INDEX CLEANUP
Database Name                     = WSDB
Partition Number                  = 0
Description                       = Table: USER1.SALES, Index: USER1.I2
Start Time                        = 12/15/2005 11:15:01.967939
State                             = Executing
Invocation Type                   = Automatic
Throttling:
   Priority                       = 50
Progress Monitoring:
      Total Work                  = 5 pages
      Completed Work              = 0 pages
      Start Time                  = 12/15/2005 11:15:01.979033

ID                                = 1
Type                              = ASYNCHRONOUS INDEX CLEANUP
Database Name                     = WSDB
Partition Number                  = 0
Description                       = Table: USER1.SALES, Index: USER1.I1
Start Time                        = 12/15/2005 11:15:01.978554
State                             = Executing
Invocation Type                   = Automatic
Throttling:
   Priority                       = 50
Progress Monitoring:
```

```
Total Work                  = 5 pages
Completed Work              = 0 pages
Start Time                  = 12/15/2005 11:15:01.980524
```

In this case, there are two cleaners operating on the USERS1.SALES table. One cleaner is processing index I1, and the other is processing index I2. The progress monitoring section shows the estimated total number of index pages that need cleaning and the current number of clean index pages.

The `State` field indicates the current state of a cleaner. The normal state is Executing, but the cleaner might be in Waiting state if it is waiting to be assigned to an available database agent or if the cleaner is temporarily suspended because of lock contention.

Note that different tasks on different database partitions can have the same utility ID, because each database partition assigns IDs to tasks that are running on that database partition only.

Asynchronous index cleanup for MDC tables:

You can enhance the performance of a rollout deletion-an efficient method for deleting qualifying blocks of data from multidimensional clustering (MDC) tables-by using asynchronous index cleanup (AIC). AIC is the deferred cleanup of indexes following operations that invalidate index entries.

Indexes are cleaned up synchronously during a standard rollout deletion. When a table contains many record ID (RID) indexes, a significant amount of time is spent removing the index keys that reference the table rows that are being deleted. You can speed up the rollout by specifying that these indexes are to be cleaned up after the deletion operation commits.

To take advantage of AIC for MDC tables, you must explicitly enable the *deferred index cleanup rollout* mechanism. There are two methods of specifying a deferred rollout: setting the **DB2_MDC_ROLLOUT** registry variable to `DEFER` or issuing the SET CURRENT MDC ROLLOUT MODE statement. During a deferred index cleanup rollout operation, blocks are marked as rolled out without an update to the RID indexes until after the transaction commits. Block identifier (BID) indexes are cleaned up during the delete operation because they do not require row-level processing.

AIC rollout is invoked when a rollout deletion commits or, if the database was shut down, when the table is first accessed following database restart. While AIC is in progress, queries against the indexes are successful, including those that access the index that is being cleaned up.

There is one coordinating cleaner per MDC table. Index cleanup for multiple rollouts is consolidated within the cleaner, which spawns a cleanup agent for each RID index. Cleanup agents update the RID indexes in parallel. Cleaners are also integrated with the utility throttling facility. By default, each cleaner has a utility impact priority of 50 (acceptable values are between 1 and 100, with 0 indicating no throttling). You can change this priority by using the **SET UTIL_IMPACT_PRIORITY** command or the db2UtilityControl API.

Note: In Db2 Version 9.7 and later releases, deferred cleanup rollout is not supported on range-partitioned tables with partitioned RID indexes. Only the NONE and IMMEDIATE modes are supported. The cleanup rollout type is

IMMEDIATE if the **DB2_MDC_ROLLOUT** registry variable is set to DEFER, or if the CURRENT MDC ROLLOUT MODE special register is set to DEFERRED to override the **DB2_MDC_ROLLOUT** setting.

If only nonpartitioned RID indexes exist on the table, deferred index cleanup rollout is supported. The MDC block indexes can be partitioned or nonpartitioned.

Monitoring the progress of deferred index cleanup rollout operation

Because the rolled-out blocks on an MDC table are not reusable until after the cleanup is complete, it is useful to monitor the progress of a deferred index cleanup rollout operation. Use the **LIST UTILITIES** command to display a utility monitor entry for each index being cleaned up. You can also retrieve the total number of MDC table blocks in the database that are pending asynchronous cleanup following a rollout deletion (BLOCKS_PENDING_CLEANUP) by using the ADMIN_GET_TAB_INFO table function or the **GET SNAPSHOT** command.

In the following sample output for the **LIST UTILITIES SHOW DETAIL** command, progress is indicated by the number of pages in each index that have been cleaned up. Each phase represents one RID index.

```
ID                                = 2
Type                              = MDC ROLLOUT INDEX CLEANUP
Database Name                     = WSDB
Partition Number                  = 0
Description                       = TABLE.<schema_name>.<table_name>
Start Time                        = 06/12/2006 08:56:33.390158
State                             = Executing
Invocation Type                   = Automatic
Throttling:
   Priority                       = 50
Progress Monitoring:
   Estimated Percentage Complete = 83
   Phase Number                   = 1
      Description                 = <schema_name>.<index_name>
      Total Work                  = 13 pages
      Completed Work              = 13 pages
      Start Time                  = 06/12/2006 08:56:33.391566
   Phase Number                   = 2
      Description                 = <schema_name>.<index_name>
      Total Work                  = 13 pages
      Completed Work              = 13 pages
      Start Time                  = 06/12/2006 08:56:33.391577
   Phase Number                   = 3
      Description                 = <schema_name>.<index_name>
      Total Work                  = 9 pages
      Completed Work              = 3 pages
      Start Time                  = 06/12/2006 08:56:33.391587
```

Online index defragmentation:

Online index defragmentation is enabled by the user-definable threshold, MINPCTUSED, for the minimum amount of used space on an index leaf page.

When an index key is deleted from a leaf page and this threshold is exceeded, the neighboring index leaf pages are checked to determine whether two leaf pages can be merged. If there is sufficient space on a page, and the merging of two neighboring pages is possible, the merge occurs immediately. The resulting empty index leaf page is then deleted.

The MINPCTUSED clause cannot be altered by the ALTER INDEX statement. If existing indexes require the ability to be merged via online index defragmentation, they must be dropped and then recreated with the CREATE INDEX statement specifying the MINPCTUSED clause. When enabling online index defragmentation, to increase the likelihood that pages can be merged when neighboring pages are checked, MINPCTUSED should be set to a value less than 50. A value of zero, which is the default, disables online defragmentation. Whether MINPCTFREE is set or not, the ability to perform a REORG CLEANUP on that index is not affected. Setting MINPCTFREE to a low value, 1-50, might reduce the work left for REORG CLEANUP to do as more page merging is performed automatically at run time.

Index nonleaf pages are not merged during online index defragmentation. However, empty nonleaf pages are deleted and made available for reuse by other indexes on the same table. To free deleted pages for other objects in a database managed space (DMS) storage model there are two reorganization options, REBUILD or RECLAIM EXTENTS. For system managed space (SMS) storage model only REORG REBUILD is allowed. RECLAIM EXTENTS moves pages to create full extents of deleted pages and then frees them. REBUILD rebuilds the index from scratch making the index as small as possible respecting PCTFREE.

Only REBUILD addresses the number of levels in an index. If reducing the number of levels in an index is a concern perform a reorganization with the REBUILD option.

When there is an X lock on a table, keys are physically removed from a page during key deletion. In this case, online index defragmentation is effective. However, if there is no X lock on the table during key deletion, keys are marked deleted but are not physically removed from the index page, and index defragmentation is not attempted.

To defragment indexes regardless of the value of MINPCTUSED, invoke the REORG INDEXES command with the CLEANUP ALL option. The whole index is checked, and whenever possible two neighboring leaf pages are merged. This merge is possible if at least PCTFREE free space is left on the merged page. PCTFREE can be specified at index creation time; the default value is 10 (percent).

Using relational indexes to improve performance:

Indexes can be used to improve performance when accessing table data. Relational indexes are used when accessing relational data, and indexes over XML data are used when accessing XML data.

Although the query optimizer decides whether to use a relational index to access relational table data, it is up to you to decide which indexes might improve performance and to create those indexes. The only exceptions to this are the dimension block indexes and the composite block index that are created automatically for each dimension when you create a multidimensional clustering (MDC) table.

Execute the runstats utility to collect new index statistics after you create a relational index or after you change the prefetch size. You should execute the runstats utility at regular intervals to keep the statistics current; without up-to-date statistics about indexes, the optimizer cannot determine the best data-access plan for queries.

To determine whether a relational index is used in a specific package, use the explain facility. To get advice about relational indexes that could be exploited by one or more SQL statements, use the **db2advis** command to launch the Design Advisor.

IBM InfoSphere Optim™ Query Workload Tuner provides tools for improving the performance of single SQL statements and the performance of groups of SQL statements, which are called query workloads. For more information about this product, see the product overview page at http://www.ibm.com/software/data/optim/query-workload-tuner-db2-luw/index.html. In Version 3.1.1 or later, you can also use the Workload Design Advisor to perform many operations that were available in the Db2 Design Advisor wizard. For more information see the documentation for the Workload Design Advisor at http://www.ibm.com/support/knowledgecenter/SS62YD_4.1.1/com.ibm.datatools.qrytune.workloadtunedb2luw.doc/topics/genrecsdsgn.html.

Advantages of a relational index over no index

If no index on a table exists, a table scan must be performed for each table that is referenced in an SQL query. The larger the table, the longer such a scan will take, because a table scan requires that each row be accessed sequentially. Although a table scan might be more efficient for a complex query that requires most of the rows in a table, an index scan can access table rows more efficiently for a query that returns only some table rows.

The optimizer chooses an index scan if the relational index columns are referenced in the SELECT statement and if the optimizer estimates that an index scan will be faster than a table scan. Index files are generally smaller and require less time to read than an entire table, especially when the table is large. Moreover, it might not be necessary to scan an entire index. Any predicates that are applied to the index will reduce the number of rows that must be read from data pages.

If an ordering requirement on the output can be matched with an index column, scanning the index in column order will enable the rows to be retrieved in the correct order without the need for a sort operation. Note that the existence of a relational index on the table being queried does not guarantee an ordered result set. Only an ORDER BY clause ensures the order of a result set.

A relational index can also contain include columns, which are non-indexed columns in an indexed row. Such columns can make it possible for the optimizer to retrieve required information from the index alone, without having to access the table itself.

Disadvantages of a relational index over no index

Although indexes can reduce access time significantly, they can also have adverse effects on performance. Before you create indexes, consider the effects of multiple indexes on disk space and processing time. Choose indexes carefully to address the needs of your application programs.

- Each index requires storage space. The exact amount depends on the size of the table and the size and number of columns in the relational index.
- Each insert or delete operation against a table requires additional updating of each index on that table. This is also true for each update operation that changes the value of an index key.

- Each relational index represents another potential access plan for the optimizer to consider, which increases query compilation time.

Relational index planning tips:

A well-designed index can make it easier for queries to access relational data.

Use the Design Advisor (**db2advis** command) to find the best indexes for a specific query or for the set of queries that defines a workload. This tool can make performance-enhancing recommendations, such as include columns or indexes that are enabled for reverse scans.

The following guidelines can also help you to create useful relational indexes.

- Retrieving data efficiently
  - To improve data retrieval, add *include columns* to unique indexes. Good candidates are columns that:
    - Are accessed frequently and would benefit from index-only access
    - Are not required to limit the range of index scans
    - Do not affect the ordering or uniqueness of the index key

    For example:

    ```
    create unique index idx on employee (workdept) include (lastname)
    ```

    Specifying LASTNAME as an include column rather than part of the index key means that LASTNAME is stored only on the leaf pages of the index.
  - Create relational indexes on columns that are used in the WHERE clauses of frequently run queries.

    In the following example, the WHERE clause will likely benefit from an index on WORKDEPT, unless the WORKDEPT column contains many duplicate values.

    ```
    where workdept='A01' or workdept='E21'
    ```
  - Create relational indexes with a compound key that names each column referenced in a query. When an index is specified in this way, relational data can be retrieved from the index only, which is more efficient than accessing the table.

    For example, consider the following query:

    ```
    select lastname
      from employee
      where workdept in ('A00','D11','D21')
    ```

    If a relational index is defined on the WORKDEPT and LASTNAME columns of the EMPLOYEE table, the query might be processed more efficiently by scanning the index rather than the entire table. Because the predicate references WORKDEPT, this column should be the first key column of the relational index.
- Searching tables efficiently

  Decide between ascending and descending key order, depending on the order that will be used most often. Although values can be searched in reverse direction if you specify the ALLOW REVERSE SCANS option on the CREATE INDEX statement, scans in the specified index order perform slightly better than reverse scans.
- Accessing larger tables efficiently

Use relational indexes to optimize frequent queries against tables with more than a few data pages, as recorded in the NPAGES column of the SYSCAT.TABLES catalog view. You should:

  - Create an index on any column that you will use to join tables.
  - Create an index on any column that you will be searching for specific values on a regular basis.

- Improving the performance of update or delete operations
  - To improve the performance of such operations against a parent table, create relational indexes on foreign keys.
  - To improve the performance of such operations against REFRESH IMMEDIATE and INCREMENTAL materialized query tables (MQTs), create unique relational indexes on the implied unique key of the MQT, which is composed of the columns in the GROUP BY clause of the MQT definition.
- Improving join performance

  If you have more than one choice for the first key column in a multiple-column relational index, use the column that is most often specified with an equijoin predicate (*expression1* = *expression2*) or the column with the greatest number of distinct values as the first key column.
- Sorting
  - For fast sort operations, create relational indexes on columns that are frequently used to sort the relational data.
  - To avoid some sorts, use the CREATE INDEX statement to define primary keys and unique keys whenever possible.
  - Create a relational index to order the rows in whatever sequence is required by a frequently run query. Ordering is required by the DISTINCT, GROUP BY, and ORDER BY clauses.

    The following example uses the DISTINCT clause:

    ```
    select distinct workdept
      from employee
    ```

    The database manager can use an index that is defined on the WORKDEPT column to eliminate duplicate values. The same index could also be used to group values, as in the following example that uses a GROUP BY clause:

    ```
    select workdept, average(salary)
      from employee
      group by workdept
    ```
- Keeping newly inserted rows clustered and avoiding page splits

  Define a clustering index, which should significantly reduce the need to reorganize the table. Use the PCTFREE option on the CREATE TABLE statement to specify how much free space should be left on each page so that rows can be inserted appropriately. You can also specify the `pagefreespace` file type modifier on the **LOAD** command.
- Saving index maintenance costs and storage space
  - Avoid creating indexes that are partial keys of other existing indexes. For example, if there is an index on columns A, B, and C, another index on columns A and B is generally not useful.
  - Do not create arbitrary indexes on many columns. Unnecessary indexes not only waste space, but also cause lengthy prepare times.
    - For online transaction processing (OLTP) environments, create one or two indexes per table.

- For read-only query environments, you might create more than five indexes per table.

  Note: For workloads involving many ad-hoc queries with many different predicates where index gap cardinality is small and the selectivity of non index gaps after the index gap is small, it is likely more appropriate to create a few large composite indexes for a table, as opposed to many smaller indexes.
- For mixed query and OLTP environments, between two and five indexes per table is likely appropriate.

- Enabling online index defragmentation

  Use the MINPCTUSED option when you create relational indexes. MINPCTUSED enables online index defragmentation; it specifies the minimum amount of space that must be in use on an index leaf page.

Relational index performance tips:

There are a number of actions that you can take to ensure that your relational indexes perform well.

- Specify a large utility heap

  If you expect a lot of update activity against the table on which a relational index is being created or reorganized, consider configuring a large utility heap (**util_heap_sz** database configuration parameter), which will help to speed up these operations.
- To avoid sort overflows in a symmetric multiprocessor (SMP) environment, increase the value of the **sheapthres** database manager configuration parameter
- Create separate table spaces for relational indexes

  You can create index table spaces on faster physical devices, or assign index table spaces to a different buffer pool, which might keep the index pages in the buffer longer because they do not compete with data pages.

  If you use a different table space for indexes, you can optimize the configuration of that table space for indexes. Because indexes are usually smaller than tables and are spread over fewer containers, indexes often have smaller extent sizes. The query optimizer considers the speed of the device that contains a table space when it chooses an access plan.
- Ensure a high degree of clustering

  If your SQL statement requires ordering of the result (for example, if it contains an ORDER BY, GROUP BY, or DISTINCT clause), the optimizer might not choose an available index if:

  – Index clustering is poor. For information about the degree of clustering in a specific index, query the CLUSTERRATIO and CLUSTERFACTOR columns of the SYSCAT.INDEXES catalog view.

  – The table is so small that it is cheaper to scan the table and to sort the result set in memory.

  – There are competing indexes for accessing the table.

  A clustering index attempts to maintain a particular order of the data, improving the CLUSTERRATIO or CLUSTERFACTOR statistics that are collected by the runstats utility. After you create a clustering index, perform an offline table reorg operation. In general, a table can only be clustered on one index. Build additional indexes after you build the clustering index.

A table's PCTFREE value determines the amount of space on a page that is to remain empty for future data insertions, so that this inserted data can be clustered appropriately. If you do not specify a PCTFREE value for a table, reorganization eliminates all extra space.

Except in the case of range-clustered tables, data clustering is not maintained during update operations. That is, if you update a record so that its key value in the clustering index changes, the record is not necessarily moved to a new page to maintain the clustering order. To maintain clustering, delete the record and then insert an updated version of the record, instead of using an update operation.

- Keep table and index statistics up-to-date

  After you create a new relational index, execute the runstats utility to collect index statistics. These statistics help the optimizer to determine whether using the index can improve data-access performance.

- Enable online index defragmentation

  Online index defragmentation is enabled if MINPCTUSED for the relational index is set to a value that is greater than zero. Online index defragmentation enables indexes to be compacted through the merging of index leaf pages when the amount of free space on a page is less than the specified MINPCTUSED value.

- Reorganize relational indexes as necessary

  To get the best performance from your indexes, consider reorganizing them periodically, because updates to tables can cause index page prefetching to become less effective.

  To reorganize an index, either drop it and recreate it, or use the reorg utility.

  To reduce the need for frequent reorganization, specify an appropriate PCTFREE value on the CREATE INDEX statement to leave sufficient free space on each index leaf page as it is being created. During future activity, records can be inserted into the index with less likelihood of index page splitting, which decreases page contiguity and, therefore, the efficiency of index page prefetching. The PCTFREE value that is specified when you create a relational index is preserved when the index is reorganized.

- Analyze explain information about relational index use

  Periodically issue EXPLAIN statements against your most frequently used queries and verify that each of your relational indexes is being used at least once. If an index is not being used by any query, consider dropping that index.

  Explain information also lets you determine whether a large table being scanned is processed as the inner table of a nested-loop join. If it is, an index on the join-predicate column is either missing or considered to be ineffective for applying the join predicate.

- Declare tables that vary widely in size as “volatile”

  A *volatile table* is a table whose cardinality at run time can vary greatly. For this kind of table, the optimizer might generate an access plan that favors a table scan instead of an index scan.

  Use the ALTER TABLE statement with the VOLATILE clause to declare such a table as volatile. The optimizer will use an index scan instead of a table scan against such tables, regardless of statistics, if:

  – All referenced columns are part of the index

  – The index can apply a predicate during the index scan

  In the case of typed tables, the ALTER TABLE...VOLATILE statement is supported only for the root table of a typed table hierarchy.

Using CURRENT MEMBER default value in a Db2 pureScale environment to improve contention issues:

In a Db2 pureScale environment, you can set the default value for a column to the CURRENT MEMBER special register. This member information can then be used to partition a table or an index, and therefore reduce database contention.

The following scenarios outline some of the situations where creating a new index using a CURRENT MEMBER column improves database contention issues. Once this new index is created, the Db2 pureScale cluster can make use of the member number information to reduce the amount of active sharing between members when referencing the table index. This resource reduction can improve the speed and overall performance of the Db2 pureScale environment.

Inserts with unique and increasing sequence values or timestamps

One row in the table can be used to store an ever-increasing value sequence (for example, purchase order numbers). These values can be generated automatically as an identity column, or programmatically through an external application. However, when these inserts occur with a significant frequency, contention issues may result, slowing down user queries.

To improve query performance, you can create a new CURRENT MEMBER column, and then index that data in conjunction with the column that stores the sequence.

1. Add the new column to store the member number:

```
ALTER TABLE ordernumber
  ADD COLUMN curmem SMALLINT DEFAULT CURRENT MEMBER IMPLICITLY HIDDEN
```

2. Create (or drop and re-create) the index on the sequence column (`seqnumber` in this example), adding the new column to the index:

```
CREATE INDEX seqindex
  ON ordernumber (curmem, seqnumber)
```

A similar approach can be taken with database designs where the sequence is a series of timestamp values. The index for the timestamp column would use the PAGE SPLIT HIGH option, and include the new CURRENT MEMBER column as well.

Indexes with only a few unique values

Some tables may have columns with only a few different values, relative to the number of rows in the table. This is another case where adding a CURRENT MEMBER column can be beneficial.

To improve the database performance in this case:

1. Add the new column to store the member number:

```
ALTER TABLE customerinfo
  ADD COLUMN curmem SMALLINT DEFAULT CURRENT MEMBER IMPLICITLY HIDDEN
```

2. Create (or drop and re-create) the index on the one or more columns with the few different values (for example, `zipcode` and `country`):

```
CREATE INDEX lowvaridx
  ON customerinfo (zipcode, country, curmem)
```

In all these cases, index compression will likely reduce the size of the index on the new CURRENT MEMBER values.

To reduce the impact on existing SQL statements, use the IMPLICITY HIDDEN option in the CREATE TABLE or ALTER TABLE command to hide the new column. This hidden column option ensures that the member information is only available to explicit SQL statement references.

Partitioning and clustering

Index behavior on partitioned tables:

Indexes on partitioned tables operate similarly to indexes on nonpartitioned tables. However, indexes on partitioned tables are stored using a different storage model, depending on whether the indexes are partitioned or nonpartitioned.

Although the indexes for a regular nonpartitioned table all reside in a shared index object, a *nonpartitioned index* on a partitioned table is created in its own index object in a single table space, even if the data partitions span multiple table spaces. Both database managed space (DMS) and system managed space (SMS) table spaces support the use of indexes in a different location than the table data. Each nonpartitioned index can be placed in its own table space, including large table spaces. Each index table space must use the same storage mechanism as the data partitions, either DMS or SMS. Indexes in large table spaces can contain up to 2^{29} pages. All of the table spaces must be in the same database partition group.

A *partitioned index* uses an index organization scheme in which index data is divided across multiple *index partitions*, according to the partitioning scheme of the table. Each index partition refers only to table rows in the corresponding data partition. All index partitions for a specific data partition reside in the same index object.

Starting in Db2 Version 9.7 Fix Pack 1, user-created indexes over XML data on XML columns in partitioned tables can be either partitioned or nonpartitioned. The default is partitioned. System-generated XML region indexes are always partitioned, and system-generated column path indexes are always nonpartitioned. In Db2 V9.7, indexes over XML data are nonpartitioned.

Benefits of a nonpartitioned index include:

- The fact that indexes can be reorganized independently of one another
- Improved performance of drop index operations
- The fact that when individual indexes are dropped, space becomes immediately available to the system without the need for index reorganization

Benefits of a partitioned index include:

- Improved data roll-in and roll-out performance
- Less contention on index pages, because the index is partitioned
- An index B-tree structure for each index partition, which can result in the following benefits:
  - Improved insert, update, delete, and scan performance because the B-tree for an index partition normally contains fewer levels than an index that references all data in the table
  - Improved scan performance and concurrency when partition elimination is in effect. Although partition elimination can be used for both partitioned and nonpartitioned index scans, it is more effective for partitioned index scans because each index partition contains keys for only the corresponding data

partition. This configuration can result in having to scan fewer keys and fewer index pages than a similar query over a nonpartitioned index.

Although a nonpartitioned index always preserves order on the index columns, a partitioned index might lose some order across partitions in certain scenarios; for example, if the partitioning columns do not match the index columns, and more than one partition is to be accessed.

During online index creation, concurrent read and write access to the table is permitted. After an online index is built, changes that were made to the table during index creation are applied to the new index. Write access to the table is blocked until index creation completes and the transaction commits. For partitioned indexes, each data partition is quiesced to read-only access *only* while changes that were made to that data partition (during the creation of the index partition) are applied.

Partitioned index support becomes particularly beneficial when you are rolling data in using the ALTER TABLE...ATTACH PARTITION statement. If nonpartitioned indexes exist (not including the XML columns path index, if the table has XML data), issue a SET INTEGRITY statement after partition attachment. This statement is necessary for nonpartitioned index maintenance, range validation, constraints checking, and materialized query table (MQT) maintenance. Nonpartitioned index maintenance can be time-consuming and require large amounts of log space. Use partitioned indexes to avoid this maintenance cost.

If there are nonpartitioned indexes (except XML columns path indexes) on the table to maintain after an attach operation, the SET INTEGRITY...ALL IMMEDIATE UNCHECKED statement behaves as though it were a SET INTEGRITY...IMMEDIATE CHECKED statement. All integrity processing, nonpartitioned index maintenance, and table state transitions are performed as though a SET INTEGRITY...IMMEDIATE CHECKED statement was issued.

The Figure 12 on page 75 diagram shows two nonpartitioned indexes on a partitioned table, with each index in a separate table space.

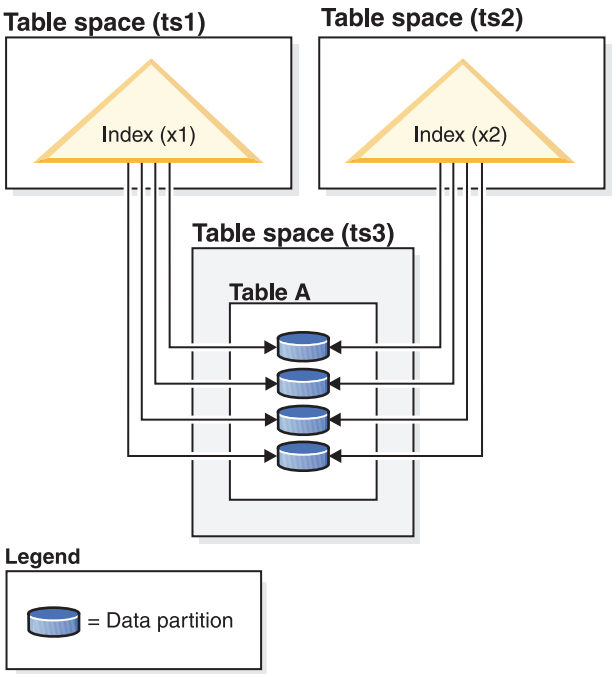


Figure 12. Nonpartitioned indexes on a partitioned table

The Figure 13 on page 76 diagram shows a partitioned index on a partitioned table that spans two database partitions and resides in a single table space.

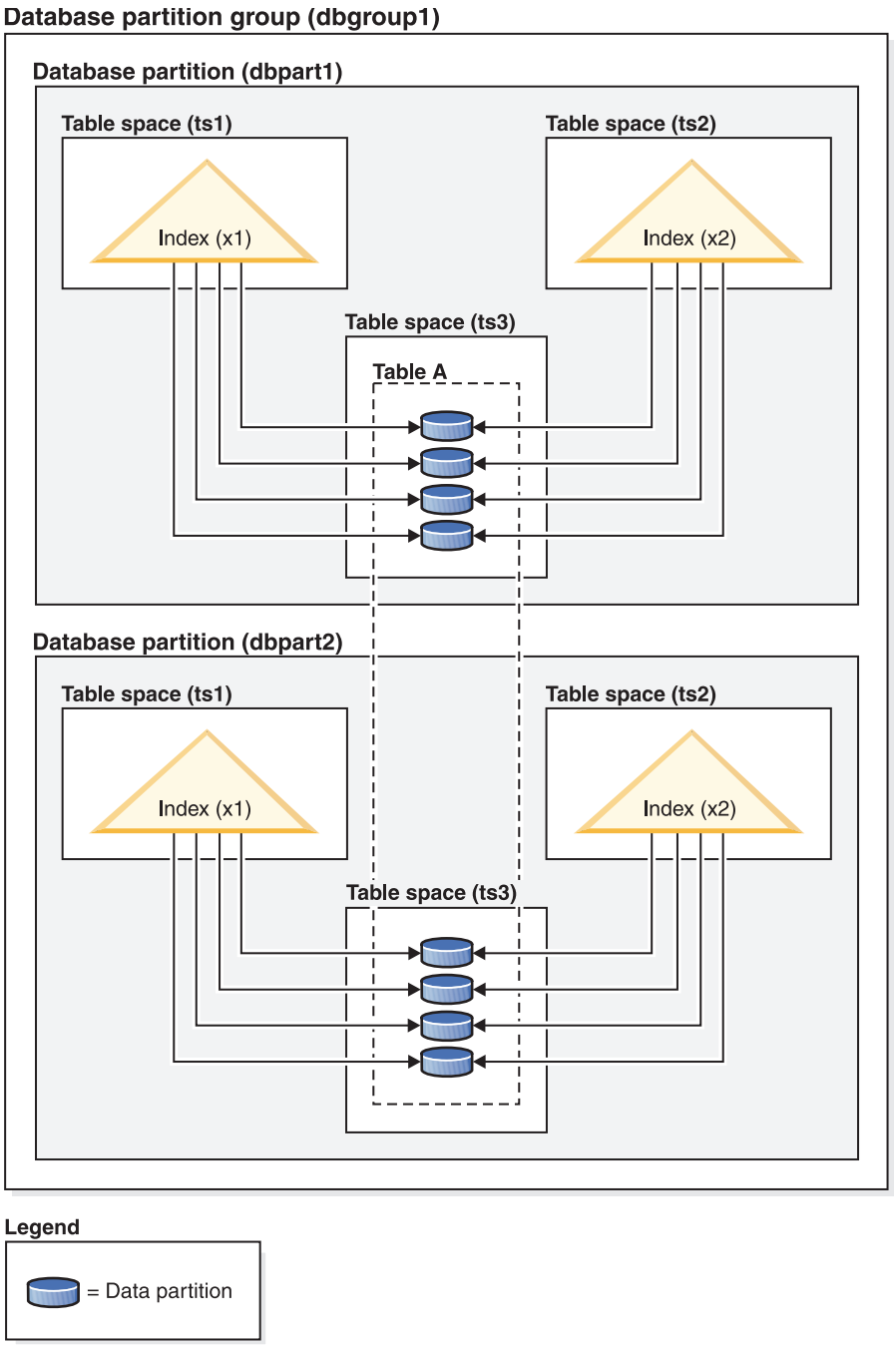


Figure 13. Nonpartitioned index on a table that is both distributed and partitioned

The Figure 14 on page 77 diagram shows a mix of partitioned and nonpartitioned indexes on a partitioned table.

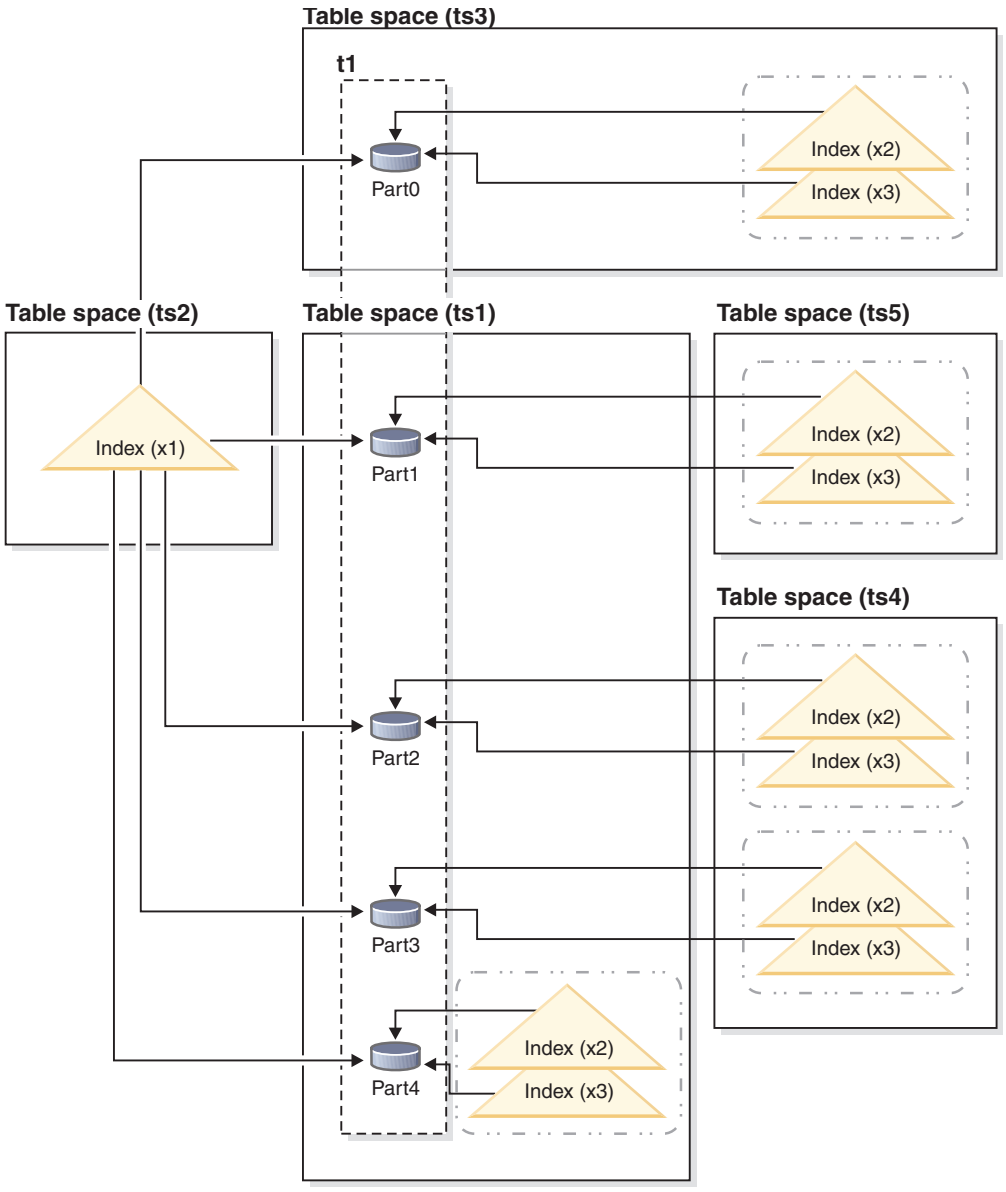


Figure 14. Partitioned and nonpartitioned indexes on a partitioned table

The nonpartitioned index X1 refers to rows in all of the data partitions. By contrast, the partitioned indexes X2 and X3 refer only to rows in the data partition with which they are associated. Table space TS3 also shows the index partitions sharing the table space of the data partitions with which they are associated. This configuration is the default for partitioned indexes.

You can override the default location for nonpartitioned and partitioned indexes, although the way that you do this is different for each. With nonpartitioned indexes, you can specify a table space when you create the index; for partitioned indexes, you need to determine the table spaces in which the index partitions are stored when you create the table.

Nonpartitioned indexes

To override the index location for nonpartitioned indexes, use the IN clause on the CREATE INDEX statement to specify an alternative table space location for the index. You can place different indexes in different

table spaces, as required. If you create a partitioned table without specifying where to place its nonpartitioned indexes, and you then create an index by using a CREATE INDEX statement that does not specify a table space, the index is created in the table space of the first attached or visible data partition. Each of the following three possible cases is evaluated in order, starting with case 1, to determine where the index is to be created. This evaluation to determine table space placement for the index stops when a matching case is found.

Case 1:
When an index table space is specified in the CREATE INDEX...IN *tbspace* statement, use the specified table space for this index.

Case 2:
When an index table space is specified in the CREATE TABLE... INDEX IN *tbspace* statement, use the specified table space for this index.

Case 3:
When no table space is specified, choose the table space that is used by the first attached or visible data partition.

Partitioned indexes

By default, index partitions are placed in the same table space as the data partitions that they reference. To override this default behavior, you must use the INDEX IN clause for each data partition that you define by using the CREATE TABLE statement. In other words, if you plan to use partitioned indexes for a partitioned table, you must anticipate where you want the index partitions to be stored when you create the table. If you try to use the INDEX IN clause when creating a partitioned index, you receive an error message.

Example 1: Given partitioned table SALES (a int, b int, c int), create a unique index A_IDX.

```
create unique index a_idx on sales (a)
```

Because the table SALES is partitioned, index a_idx is also created as a partitioned index.

Example 2: Create index B_IDX.

```
create index b_idx on sales (b)
```

Example 3: To override the default location for the index partitions in a partitioned index, use the INDEX IN clause for each partition that you define when creating the partitioned table. In the example that follows, indexes for the table Z are created in table space TS3.

```
create table z (a int, b int)
      partition by range (a) (starting from (1)
      ending at (100)  index in ts3)

   create index c_idx on z (a) partitioned
```

Clustering of nonpartitioned indexes on partitioned tables:

Clustering indexes offer the same benefits for partitioned tables as they do for regular tables. However, care must be taken with the table partitioning key definitions when choosing a clustering index.

You can create a clustering index on a partitioned table using any clustering key. The database server attempts to use the clustering index to cluster data locally within each data partition. During a clustered insert operation, an index lookup is performed to find a suitable record identifier (RID). This RID is used as a starting point in the table when looking for space in which to insert the record. To achieve optimal clustering with good performance, there should be a correlation between the index columns and the table partitioning key columns. One way to ensure such correlation is to prefix the index columns with the table partitioning key columns, as shown in the following example:

```
partition by range (month, region)
create index...(month, region, department) cluster
```

Although the database server does not enforce this correlation, there is an expectation that all keys in the index will be grouped together by partition IDs to achieve good clustering. For example, suppose that a table is partitioned on QUARTER and a clustering index is defined on DATE. There is a relationship between QUARTER and DATE, and optimal clustering of the data with good performance can be achieved because all keys of any data partition are grouped together within the index. Figure 15 on page 80 shows that optimal scan performance is achieved only when clustering correlates with the table partitioning key.

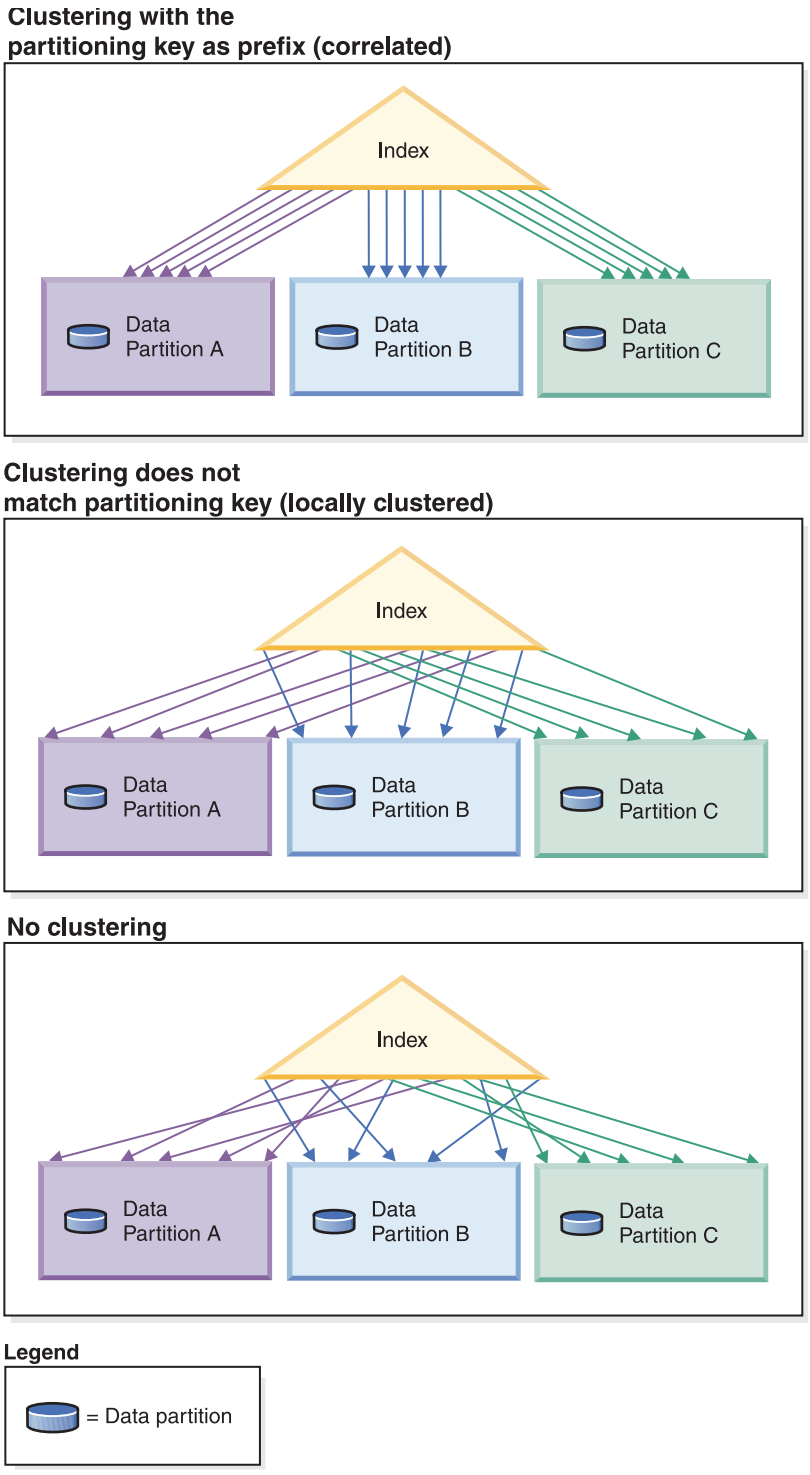


Figure 15. The possible effects of a clustered index on a partitioned table.

Benefits of clustering include:

- Rows are in key order within each data partition.
- Clustering indexes improve the performance of scans that traverse the table in key order, because the scanner fetches the first row of the first page, then each row in that same page before moving on to the next page. This means that only one page of the table needs to be in the buffer pool at any given time. In

contrast, if the table is not clustered, rows are likely fetched from different pages. Unless the buffer pool can hold the entire table, most pages will likely be fetched more than once, greatly slowing down the scan.

If the clustering key is not correlated with the table partitioning key, but the data is locally clustered, you can still achieve the full benefit of the clustered index if there is enough space in the buffer pool to hold one page of each data partition. This is because each fetched row from a particular data partition is near the row that was previously fetched from that same partition (see the second example in Figure 15 on page 80).

Federated databases

Server options that affect federated databases:

A federated database system is composed of a Db2 data server (the federated database) and one or more data sources. You identify the data sources to the federated database when you issue CREATE SERVER statements. You can also specify server options that refine and control various aspects of federated system operation.

You must install the distributed join installation option and set the **federated** database manager configuration parameter to YES before you can create servers and specify server options. To change server options later, use the ALTER SERVER statement.

The server option values that you specify on the CREATE SERVER statement affect query pushdown analysis, global optimization, and other aspects of federated database operations. For example, you can specify performance statistics as server option values. The *cpu_ratio* option specifies the relative speeds of the processors at the data source and the federated server, and the *io_ratio* option specifies the relative rates of the data I/O divides at the data source and the federated server.

Server option values are written to the system catalog (SYSCAT.SERVEROPTIONS), and the optimizer uses this information when it develops access plans for the data source. If a statistic changes (for example, when a data source processor is upgraded), use the ALTER SERVER statement to update the catalog with the new value.

Resource utilization

Memory allocation

Memory allocation and deallocation occurs at various times. Memory might be allocated to a particular memory area when a specific event occurs (for example, when an application connects), or it might be reallocated in response to a configuration change.

Figure 16 on page 82 shows the different memory areas that the database manager allocates for various uses and the configuration parameters that enable you to control the size of these memory areas. Note that in a partitioned database environment, each database partition has its own database manager shared memory set.

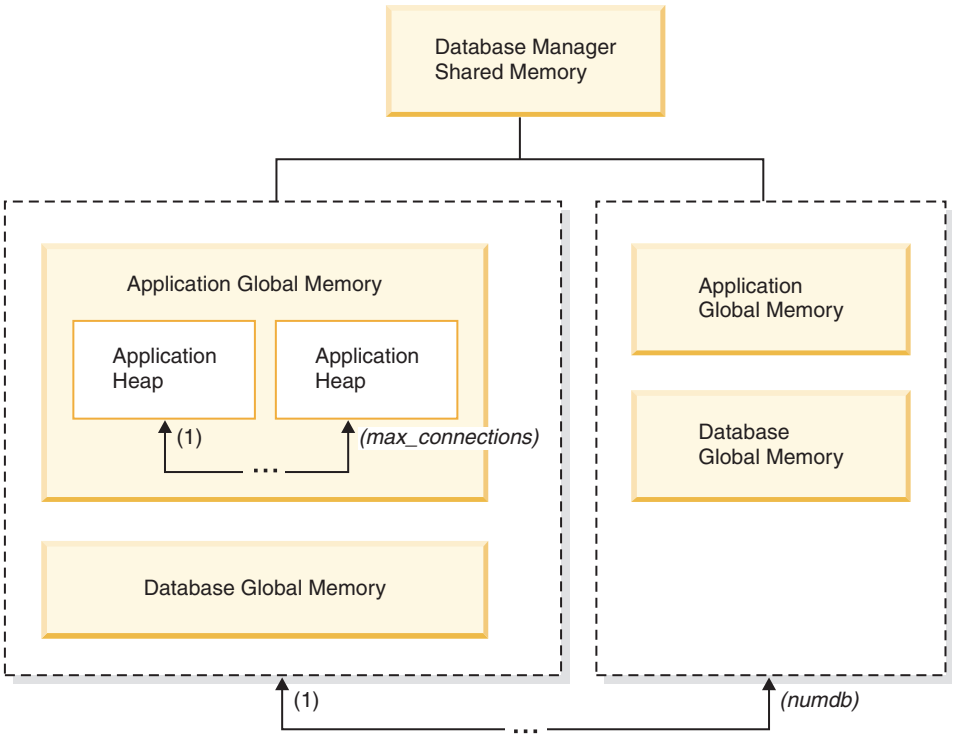


Figure 16. Types of memory allocated by the database manager

Memory is allocated by the database manager whenever one of the following events occurs:

When the database manager starts (`db2start`)
Database manager shared memory (also known as *instance shared memory*) remains allocated until the database manager stops (**`db2stop`**). This area contains information that the database manager uses to manage activity across all database connections. Db2 automatically controls the size of the database manager shared memory.

When a database is activated or connected to for the first time
Database global memory is used across all applications that connect to the database. The size of the database global memory is specified by the **`database_memory`** database configuration parameter. By default, this parameter is set to automatic, allowing Db2 to calculate the initial amount of memory allocated for the database and to automatically configure the database memory size during run time based on the needs of the database.

The following memory areas can be dynamically adjusted:

- Buffer pools (using the ALTER BUFFERPOOL statement)
- Database heap (including log buffers)
- Utility heap
- Package cache
- Catalog cache
- Lock list

The **`sortheap`**, **`sheapthres_shr`**, and **`sheapthres`** configuration parameters are also dynamically updatable. The only restriction is that **`sheapthres`** cannot be dynamically changed from 0 to a value that is greater than zero, or vice versa.

Shared sort operations are performed by default, and the amount of database shared memory that can be used by sort memory consumers at any one time is determined by the value of the **sheapthres_shr** database configuration parameter. Private sort operations are performed only if intrapartition parallelism, database partitioning, and the connection concentrator are all disabled, and the **sheapthres** database manager configuration parameter is set to a non-zero value.

When an application connects to a database
Each application has its own *application heap*, part of the *application global memory*. You can limit the amount of memory that any one application can allocate by using the **applheapsz** database configuration parameter, or limit overall application memory consumption by using the **appl_memory** database configuration parameter.

When an agent is created
Agent private memory is allocated for an agent when that agent is assigned as the result of a connect request or a new SQL request in a partitioned database environment. Agent private memory contains memory that is used only by this specific agent. If private sort operations have been enabled, the private sort heap is allocated from agent private memory.

The following configuration parameters limit the amount of memory that is allocated for each type of memory area. Note that in a partitioned database environment, this memory is allocated on each database partition.

numdb This database manager configuration parameter specifies the maximum number of concurrent active databases that different applications can use. Because each database has its own global memory area, the amount of memory that can be allocated increases if you increase the value of this parameter.

maxappls
This database configuration parameter specifies the maximum number of applications that can simultaneously connect to a specific database. The value of this parameter affects the amount of memory that can be allocated for both agent private memory and application global memory for that database.

max_connections
This database manager configuration parameter limits the number of database connections or instance attachments that can access the data server at any one time.

max_coordagents
This database manager configuration parameter limits the number of database manager coordinating agents that can exist simultaneously across all active databases in an instance (and per database partition in a partitioned database environment). Together with **maxappls** and **max_connections**, this parameter limits the amount of memory that is allocated for agent private memory and application global memory.

You can use the memory tracker, invoked by the **db2mtrk** command, to view the current allocation of memory within the instance. You can also use the ADMIN_GET_MEM_USAGE table function to determine the total memory consumption for the entire instance or for just a single database partition. Use the MON_GET_MEMORY_SET and MON_GET_MEMORY_POOL table functions to examine the current memory usage at the instance, database, or application level.

On UNIX and Linux operating systems, although the **ipcs** command can be used to list all the shared memory segments, it does not accurately reflect the amount of resources consumed. You can use the **db2mtrk** command as an alternative to **ipcs**.

Memory sets overview:

The Db2 memory manager groups memory allocations from the operating system into *memory sets*.

The memory in a given memory set shares common attributes, such as the general purpose for which the memory is used, its expected volatility, and what, if any constraints there might be on its growth. For example, buffer pools are allocated from the database memory set, and are allocated as long as the database is active. Statement heaps are allocated from the application memory set on behalf of specific SQL preparation requests from an application, and last only as long as the statement compilation operation.

Within each memory set, specific areas of memory get assigned for purposes that are generally related to the specific memory set type. For example, certain types of database-level processing use memory from the database memory set; memory required for the fast communications manager is allocated from the FCM memory set.

Table 1 lists the different types of memory sets.

Table 1. Memory sets

Memory set type*	Description	Scope of memory allocated from this set
DBMS	Database memory manager set. Most memory in this set is used for basic infrastructure purposes, including communication services that are not specific to a database. This memory set does not require any configuration, although user-configurable memory from this set include the monitor heap size (**mon_heap_sz**) and the audit buffer size (**audit_buf_sz**).	Instance
FMP	Fenced mode process memory set. Memory allocated from this set is used for communications between agents and fenced mode processes. Memory allocations from this set can be configured with the **DB2_FMP_COMM_HEAPSZ** registry variable and the **aslheapsz** configuration parameter.	Instance
PRIVATE	Memory allocated from this set is used for general purposes, including basic infrastructure and diagnostics support. With the exception of systems that use the private sort model, where the **sheapthres** configuration parameter is set to a value other than 0, the private memory set does not require any configuration.	Instance

Table 1. Memory sets (continued)

Memory set type*	Description	Scope of memory allocated from this set
DATABASE	Database memory set. Memory allocated from this set is generally used for processing that is specific to a single database, but not specific to an application. Examples of memory allocated from this set includes the buffer pools, database heap, locklist, utility heap, package cache, catalog cache, and the shared sort heap. This set can be configured through the **database_memory** database configuration parameter. You can also use the self-tuning memory manager (STMM) to tune this memory area.	Database
APPLICATION	Application memory set. Memory allocated from this set is generally used for application-specific processing. Memory allocated from this set includes application, statistics and statement heaps, and a non-configurable shared work area. This set can be configured through the **appl_memory** database configuration parameter.	Database
FCM	Fast communication manager memory set. Memory allocated from this set is used exclusively by the fast communications manager. Memory from this set can be configured with the **fcm_num_buffers** and the **fcm_num_channels** database manager configuration parameters.	Host
* The names shown in the first column are the names returned for the **memory_set_type** monitor element.		

Database manager shared memory:

Database manager shared memory is organized into several different memory areas. Configuration parameters enable you to control the sizes of these areas.

Figure 17 on page 86 shows how database manager shared memory is allocated.

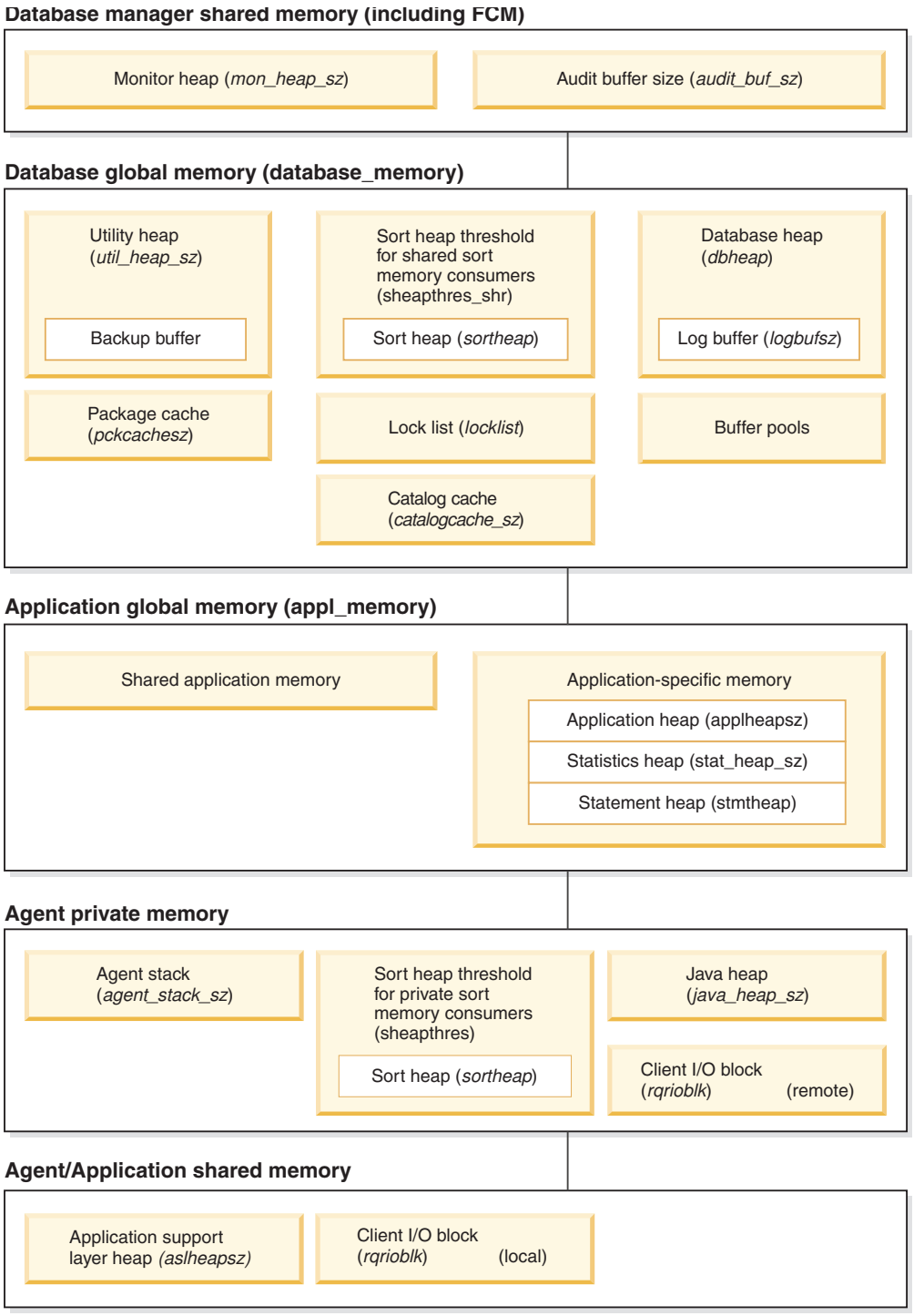


Figure 17. How memory is used by the database manager

Database Manager Shared Memory

Database Manager Shared Memory is affected by the following configuration parameters:

- The **audit_buf_sz** configuration parameter determines the size of the buffer used in database auditing activities.

- The **mon_heap_sz** configuration parameter determines the size of the memory area used for database system monitoring data.
- For partitioned database systems, the Fast Communications Manager (FCM) requires substantial memory space, especially if the value of **fcm_num_buffers** is large. The FCM memory requirements are allocated from the FCM Buffer Pool.

Database global memory

Database global memory is affected by the size of the buffer pools and by the following database configuration parameters:

- **catalogcache_sz**
- **database_memory**
- **dbheap**
- **locklist**
- **pckcachesz**
- **sheapthres_shr**
- **util_heap_sz**

Application global memory

Application global memory can be controlled by the **appl_memory** configuration parameter. The following database configuration parameters can be used to limit the amount of memory that any one application can consume:

- **applheapsz**
- **stat_heap_sz**
- **stmtheap**

Agent private memory

Each agent requires its own private memory region. The data server creates as many agents as it needs and in accordance with configured memory resources. You can control the maximum number of coordinator agents using the **max_coordagents** database manager configuration parameter. The maximum size of each agent's private memory region is determined by the values of the following configuration parameters:

- **agent_stack_sz**
- **sheapthres** and **sortheap**

Agent/Application shared memory

The total number of agent/application shared memory segments for local clients is limited by the lesser of the following two values:

- The total value of the **maxappls** database configuration parameter for all active databases
- The value of the **max_coordagents** database configuration parameter

Note: In configurations where engine concentration is enabled (**max_connections** > **max_coordagents**), application memory consumption is limited by **max_connections**.

Agent/Application shared memory is also affected by the following database configuration parameters:

- **aslheapsz**
- **rqrioblk**

The FCM buffer pool and memory requirements:

In a partitioned database system, the fast communication manager (FCM) buffer shared memory is allocated from the database manager shared memory.

This is shown in Figure 18.

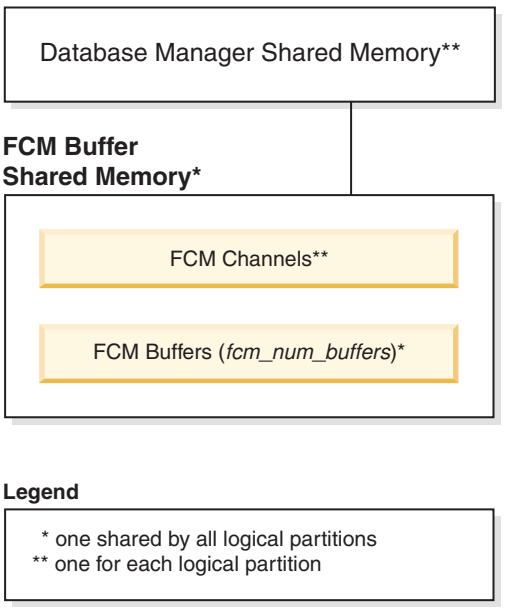


Figure 18. The FCM buffer pool when multiple logical partitions are used

The number of FCM buffers for each database partition is controlled by the **fcm_num_buffers** database manager configuration parameter. By default, this parameter is set to automatic. To tune this parameter manually, use data from the **buff_free** and **buff_free_bottom** system monitor elements.

The number of FCM channels for each database partition is controlled by the **fcm_num_channels** database manager configuration parameter. By default, this parameter is set to automatic. To tune this parameter manually, use data from the **ch_free** and **ch_free_bottom** system monitor elements.

The Db2 database manager can automatically manage FCM memory resources by allocating more FCM buffers and channels as needed. This leads to improved performance and prevents "out of FCM resource" runtime errors. On the Linux operating system, the database manager can preallocate a larger amount of system memory for FCM buffers and channels, up to a maximum default amount of 4 GB. Memory space is impacted only when additional FCM buffers or channels are required. To enable this behavior, set the FCM_MAXIMIZE_SET_SIZE option of the **DB2_FCM_SETTINGS** registry variable to YES (or TRUE). YES is the default value.

Guidelines for tuning parameters that affect memory usage:

When tuning memory manually (that is, when not using the self-tuning memory manager), benchmark tests provide the best information about setting appropriate values for memory parameters.

In benchmark testing, representative and worst-case SQL statements are run against the server, and the values of memory parameters are changed until a point

of diminishing returns for performance is found. This is the point at which additional memory allocation provides no further performance value to the application.

The upper memory allocation limits for several parameters might be beyond the scope of existing hardware and operating systems. These limits allow for future growth. It is good practice to not set memory parameters at their highest values unless those values can be justified. This applies even to systems that have plenty of available memory. The idea is to prevent the database manager from quickly taking up all of the available memory on a system. Moreover, managing large amounts of memory incurs additional overhead.

For most configuration parameters, memory is committed as it is required, and the parameter settings determine the maximum size of a particular memory heap. For buffer pools and the following configuration parameters, however, all of the specified memory is allocated:

- **aslheapsz**
- **fcm_num_buffers**
- **fcm_num_channels**
- **locklist**

For valid parameter ranges, refer to the detailed information about each parameter.

Self-tuning memory overview

Self-tuning memory simplifies the task of memory configuration by automatically setting values for memory configuration parameters and sizing buffer pools. When enabled, the memory tuner dynamically distributes available memory resources among the following memory consumers: buffer pools, locking memory, package cache, and sort memory.

Self-tuning memory is enabled through the **self_tuning_mem** database configuration parameter.

The following memory-related database configuration parameters can be automatically tuned:

- **database_memory** - Database shared memory size
- **locklist** - Maximum storage for lock list
- **maxlocks** - Maximum percent of lock list before escalation
- **pckcachesz** - Package cache size
- **sheapthres_shr** - Sort heap threshold for shared sorts
- **sortheap** - Sort heap size

Starting withDb2 Cancun Release 10.5.0.4, each member in a Db2 pureScale environment has its own self-tuning memory manager (STMM) tuner.

Starting withDb2 Version 10.5 Fix Pack 5, the following memory-related database configuration parameters can also be automatically tuned in a Db2 pureScale environment:

- **cf_db_mem_sz** - CF Database memory
- **cf_gbp_sz** - Group buffer pool
- **cf_lock_sz**- CF Lock manager
- **cf_sca_sz** - Shared communication area

Self-tuning memory:

A memory-tuning feature simplifies the task of memory configuration by automatically setting values for several memory configuration parameters. When enabled, the memory tuner dynamically distributes available memory resources among the following memory consumers: buffer pools, locking memory, package cache, and sort memory.

The tuner works within the memory limits that are defined by the **database_memory** configuration parameter. The value of this parameter can be automatically tuned as well. When self-tuning is enabled (when the value of **database_memory** has been set to AUTOMATIC), the tuner determines the overall memory requirements for the database and increases or decreases the amount of memory allocated for database shared memory, depending on current database requirements. For example, if current database requirements are high and there is sufficient free memory on the system, more memory is allocated for database shared memory. If the database memory requirements decrease, or if the amount of free memory on the system becomes too low, some database shared memory is released. If large pages or pinned memory are enabled, the STMM will not tune the overall database memory configuration, and you need to assign a specific amount of memory to database memory by setting the DATABASE_MEMORY configuration parameter to a specific value. For more information, see database_memory, DB2_LARGE_PAGE_MEM in Performance variables, and Enabling large page support (AIX).

If the **database_memory** configuration parameter is not set to AUTOMATIC, the database uses the amount of memory that has been specified for this parameter, distributing it across the memory consumers as required. You can specify the amount of memory in one of two ways: by setting **database_memory** to some numeric value or by setting it to COMPUTED. In the latter case, the total amount of memory is based on the sum of the initial values of the database memory heaps at database startup time.

You can also enable the memory consumers for self tuning as follows:

- For buffer pools, use the ALTER BUFFERPOOL or the CREATE BUFFERPOOL statement (specifying the AUTOMATIC keyword)
- For locking memory, use the **locklist** or the **maxlocks** database configuration parameter (specifying a value of AUTOMATIC)
- For the package cache, use the **pckcachesz** database configuration parameter (specifying a value of AUTOMATIC)
- For sort memory, use the **sheapthres_shr** or the **sortheap** database configuration parameter (specifying a value of AUTOMATIC)

Changes resulting from self-tuning operations are recorded in memory tuning log files that are located in the `stmmlog` subdirectory. These log files contain summaries of the resource demands from each memory consumer during specific tuning intervals, which are determined by timestamps in the log entries.

If little memory is available, the performance benefits of self tuning will be limited. Because tuning decisions are based on database workload, workloads with rapidly changing memory requirements limit the effectiveness of the self-tuning memory manager (STMM). If the memory characteristics of your workload are constantly changing, the STMM will tune less frequently and under shifting target conditions. In this scenario, the STMM will not achieve absolute convergence, but will try instead to maintain a memory configuration that is tuned to the current workload.

Enabling self-tuning memory:

Self-tuning memory simplifies the task of memory configuration by automatically setting values for memory configuration parameters and sizing buffer pools.

About this task

When enabled, the memory tuner dynamically distributes available memory resources between several memory consumers, including buffer pools, locking memory, package cache, and sort memory.

Procedure

1. Enable self-tuning memory for the database by setting the **self_tuning_mem** database configuration parameter to ON using the **UPDATE DATABASE CONFIGURATION** command or the db2CfgSet API.
2. To enable the self tuning of memory areas that are controlled by memory configuration parameters, set the relevant configuration parameters to AUTOMATIC using the **UPDATE DATABASE CONFIGURATION** command or the db2CfgSet API.
3. To enable the self tuning of a buffer pool, set the buffer pool size to AUTOMATIC using the CREATE BUFFERPOOL statement or the ALTER BUFFERPOOL statement. In a partitioned database environment, that buffer pool should not have any entries in SYSCAT.BUFFERPOOLDBPARTITIONS.

Results

Note:

1. Because self-tuned memory is distributed between different memory consumers, at least two memory areas must be concurrently enabled for self tuning at any given time; for example, locking memory and database shared memory. The memory tuner actively tunes memory on the system (the value of the **self_tuning_mem** database configuration parameter is ON) when one of the following conditions is true:
   - One configuration parameter or buffer pool size is set to AUTOMATIC, and the **database_memory** database configuration parameter is set to either a numeric value or to AUTOMATIC
   - Any two of **locklist**, **sheapthres_shr**, **pckcachesz**, or buffer pool size is set to AUTOMATIC
   - The **sortheap** database configuration parameter is set to AUTOMATIC
2. The value of the **locklist** database configuration parameter is tuned together with the **maxlocks** database configuration parameter. Disabling self tuning of the **locklist** parameter automatically disables self tuning of the **maxlocks** parameter, and enabling self tuning of the **locklist** parameter automatically enables self tuning of the **maxlocks** parameter.
3. Automatic tuning of **sortheap** or the **sheapthres_shr** database configuration parameter is allowed only when the database manager configuration parameter **sheapthres** is set to 0.
4. The value of **sortheap** is tuned together with **sheapthres_shr**. Disabling self tuning of the **sortheap** parameter automatically disables self tuning of the **sheapthres_shr** parameter, and enabling self tuning of the **sheapthres_shr** parameter automatically enables self tuning of the **sortheap** parameter.
5. Self-tuning memory runs only on the high availability disaster recovery (HADR) primary server. When self-tuning memory is activated on an HADR

system, it will never run on the secondary server, and it runs on the primary server only if the configuration is set properly. If the HADR database roles are switched, self-tuning memory operations will also switch so that they run on the new primary server. After the primary database starts, or the standby database converts to a primary database through takeover, the self-tuning memory manager (STMM) engine dispatchable unit (EDU) might not start until the first client connects.

Disabling self-tuning memory:

Self-tuning memory can be disabled for the entire database or for one or more configuration parameters or buffer pools.

About this task

If self-tuning memory is disabled for the entire database, the memory configuration parameters and buffer pools that are set to AUTOMATIC remain enabled for automatic tuning; however, the memory areas remain at their current size.

Procedure

1. Disable self-tuning memory for the database by setting the **self_tuning_mem** database configuration parameter to OFF using the **UPDATE DATABASE CONFIGURATION** command or the db2CfgSet API.
2. To disable the self tuning of memory areas that are controlled by memory configuration parameters, set the relevant configuration parameters to MANUAL or specify numeric parameter values using the **UPDATE DATABASE CONFIGURATION** command or the db2CfgSet API.
3. To disable the self tuning of a buffer pool, set the buffer pool size to a specific value using the ALTER BUFFERPOOL statement.

Results

Note:

- In some cases, a memory configuration parameter can be enabled for self tuning only if another related memory configuration parameter is also enabled. This means that, for example, disabling self-tuning memory for the **locklist** or the **sortheap** database configuration parameter disables self-tuning memory for the **maxlocks** or the **sheapthres_shr** database configuration parameter, respectively.

Determining which memory consumers are enabled for self tuning:

You can view the self-tuning memory settings that are controlled by configuration parameters or that apply to buffer pools.

About this task

It is important to note that responsiveness of the memory tuner is limited by the time required to resize a memory consumer. For example, reducing the size of a buffer pool can be a lengthy process, and the performance benefits of trading buffer pool memory for sort memory might not be immediately realized.

Procedure

- To view the settings for configuration parameters, use one of the following methods:

– Use the **GET DATABASE CONFIGURATION** command, specifying the **SHOW DETAIL** parameter.

  The memory consumers that can be enabled for self tuning are grouped together in the output as follows:

```
Description                        Parameter         Current Value       Delayed Value
-------------------------------------------------------------------------------------------
Self tuning memory                (SELF_TUNING_MEM) = ON (Active)        ON
Size of database shared memory (4KB)  (DATABASE_MEMORY) = AUTOMATIC(37200)   AUTOMATIC(37200)
Max storage for lock list (4KB)             (LOCKLIST) = AUTOMATIC(7456)    AUTOMATIC(7456)
Percent. of lock lists per application      (MAXLOCKS) = AUTOMATIC(98)      AUTOMATIC(98)
Package cache size (4KB)                 (PCKCACHESZ) = AUTOMATIC(5600)    AUTOMATIC(5600)
Sort heap thres for shared sorts (4KB) (SHEAPTHRES_SHR) = AUTOMATIC(5000)    AUTOMATIC(5000)
Sort list heap (4KB)                        (SORTHEAP) = AUTOMATIC(256)     AUTOMATIC(256)
```

– Use the db2CfgGet API.

  The following values are returned:

```
SQLF_OFF                    0
SQLF_ON_ACTIVE              2
SQLF_ON_INACTIVE            3
```

  SQLF_ON_ACTIVE indicates that self tuning is both enabled and active, whereas SQLF_ON_INACTIVE indicates that self tuning is enabled but currently inactive.

- To view the self-tuning settings for buffer pools, use one of the following methods:
  – To retrieve a list of the buffer pools that are enabled for self tuning from the command line, use the following query:

```
SELECT BPNAME, NPAGES FROM SYSCAT.BUFFERPOOLS
```

    When self tuning is enabled for a buffer pool, the NPAGES field in the SYSCAT.BUFFERPOOLS view for that particular buffer pool is set to -2. When self tuning is disabled, the NPAGES field is set to the current size of the buffer pool.

  – To determine the current size of buffer pools that are enabled for self tuning, use the **GET SNAPSHOT** command and examine the current size of the buffer pools (the value of the **bp_cur_buffsz** monitor element):

```
GET SNAPSHOT FOR BUFFERPOOLS ON database-alias
```

    An ALTER BUFFERPOOL statement that specifies the size of a buffer pool on a particular database partition creates an exception entry (or updates an existing entry) for that buffer pool in the SYSCAT.BUFFERPOOLDBPARTITIONS catalog view. If an exception entry for a buffer pool exists, that buffer pool does not participate in self-tuning operations when the default buffer pool size is set to AUTOMATIC.

Member self-tuning memory in Db2 pureScale environments:

In a Db2 pureScale environment, each member has its own self-tuning memory manager (STMM) tuner, which actively tunes the memory configurations of the particular member that is based on dynamic workload characteristics and local resources.

For a new database that is created in the V11.1 release, the default value of STMM tuning member in the SYSCAT table is -2. This setting ensures that each member has its own tuner, tuning independently to balance the following factors:

- Workload
- Db2 memory requirements
- System memory requirements

The memory requirements might be different on each member.

Ensuring that STMM tuning capabilities are present on each member is important in the following scenarios:

- A consolidated environment where multiple databases can have workload peak at different time of the day.
- The "member subsetting" capability is enabled so that a workload can be spread across selected members.

STMM decides on which member the STMM tuner is active based on the value in the SYSCAT table. The SYSCAT table is updated by using the **UPDATE STMM TUNING MEMBER** stored procedure, as shown:

```
CALL SYSPROC.ADMIN_CMD('update stmm tuning member member')
```

Here is a summary of the members on which the STMM tuner is active based on the value in the SYSCAT table.

Table 2. Determining the members on which STMM tuner is active

Value in SYSCAT table	Member where STMM Tuner is running
-2	All members.
-1	One member, which is chosen by STMM.
Any number that matches a member number	Member with number that matches the value in SYSCAT
Any number that does not match a member number	Defaults to -1, where the tuner runs on one member, which is chosen by STMM

Note that when the tuning member changes, some data collected from the member which was running the tuner, is discarded. This data must be recollected on the new tuning member. During this short period of time when the data is being recollected, the memory tuner will still tune the system; however, the tuning can occur slightly differently than it did on the original member.

Starting the memory tuner in a Db2 pureScale environment

In a Db2 pureScale environment, the memory tuner will run whenever the database is active on one or more members that have **self_tuning_mem** set to ON.

Disabling self-tuning memory for a specific member

- To disable self-tuning memory for a subset of database members, set the **self_tuning_mem** database configuration parameter to OFF for those members.
- To disable self-tuning memory for a subset of the memory consumers that are controlled by configuration parameters on a specific member, set the value of the relevant configuration parameter to a fixed value on that member. It is recommended that self-tuning memory configuration parameter values be consistent across all running members.
- To disable self-tuning memory for a particular buffer pool on a specific member, issue the ALTER BUFFERPOOL statement, specifying a size value and the member on which self-tuning memory is to be disabled.

An ALTER BUFFERPOOL statement that specifies the size of a buffer pool on a particular member will create an exception entry (or update an existing entry) for that buffer pool in the SYSCAT.BUFFERPOOLEXCEPTIONS catalog view. If an exception entry for a buffer pool exists, that buffer pool will not participate in

self-tuning operations when the default buffer pool size is set to AUTOMATIC. To remove an exception entry so that a buffer pool can be used for self tuning:

1. Disable self tuning for this buffer pool by issuing an ALTER BUFFERPOOL statement, setting the buffer pool size to a specific value.
2. Issue another ALTER BUFFERPOOL statement to set the size of the buffer pool on this member to the default.
3. Enable self tuning for this buffer pool by issuing another ALTER BUFFERPOOL statement, setting the buffer pool size to AUTOMATIC.

CF self-tuning memory parameter configuration:

In a Db2 pureScale instance, several configuration parameters and registry variables work together to control memory allocation for the cluster caching facility, also known as the CF.

In Db2 Version 10.5 Fix Pack 5 and later fix packs of Version 10.5, when explicitly enabled, cluster caching facility (CF) self-tuning memory optimizes memory distribution and avoids out of memory conditions. Unless explicitly enabled, CF memory parameter configuration functions as before.

In Version 11.1, (CF) self-turning memory is enabled by default.

In a Db2 pureScale instance, CF self-tuning memory automatically and continuously monitors the usage of memory and optimizes the distribution of memory. CF self-tuning memory monitors database memory parameter **cf_db_mem_sz**, and these CF memory consumer parameters:

- Group buffer pool (**cf_gbp_sz**),
- Lock manager (**cf_lock_sz**), and
- Shared communication area (**cf_sca_sz**).

CF memory is dynamically distributed between these CF memory consumer parameters. In addition, in multiple database environments, CF memory can be dynamically distributed between databases based on their need for memory.

CF self-tuning memory is set at the instance level, and is enabled (that is, turned on) by default or by setting registry variable **DB2_DATABASE_CF_MEMORY** to AUTO.

```
db2set DB2_DATABASE_CF_MEMORY=AUTO
```

Setting **DB2_DATABASE_CF_MEMORY** to AUTO turns on CF self-tuning memory for all databases that have the CF memory consumer parameters (**cf_gbp_sz**, **cf_lock_sz**, **cf_sca_sz**) set to AUTOMATIC. However, for databases that have the CF memory consumer parameters set to fixed values, CF self-tuning memory remains off.

Note:

1. In Db2 Version 10.5 Fix Pack 5 and later fix packs, if you are applying an online fix pack, you cannot set registry variable **DB2_DATABASE_CF_MEMORY** until after the instance is committed to the new fix pack level.
2. High availability disaster recovery (HADR) in a Db2 pureScale environment can use CF self-tuning memory. However, CF memory tuning occurs on the primary site only. If the registry variable is set on the standby site, the registry variable takes effect when the standby site becomes the primary site.

CF self-tuning memory can be used at different levels:

- To dynamically adjust the amount of CF memory a database is assigned, set **cf_db_mem_sz** to AUTOMATIC.

  In this case, depending on workload, CF memory is available for use between all databases that also have **cf_db_mem_sz** set to AUTOMATIC, and database memory within one database.

  Database memory is also dynamically distributed between the CF memory consumer parameters (**cf_gbp_sz**, **cf_lock_sz**, **cf_sca_sz**).
- To dynamically distribute the CF memory for a database among the memory consumer parameters, set **cf_db_mem_sz** to a fix value.

  In this case, memory is not dynamically distributed between databases, but memory is dynamically distributed between the CF memory consumer parameters (**cf_gbp_sz**, **cf_lock_sz**, **cf_sca_sz**).

If you have more than one database, while it is optimal for all databases to use CF self-tuning memory, it is possible to have only a subset of the databases in the instance use CF self-tuning memory. That is, one or more databases has **cf_db_mem_sz** set to a fixed value. When using a fixed value, the databases do not need to have the same fixed value. Also, in this case, to avoid the possibility of insufficient free CF memory, the order of database activation is important. Activate the databases with a fixed value first.

Before Db2 Version 10.5 Fix Pack 5, the value of database memory parameter **cf_db_mem_sz** was static. In Db2 Version 10.5 Fix Pack 5 and later fix packs, when **cf_db_mem_sz** is set to AUTOMATIC, CF self-tuning memory limits the maximum CF database memory that is based on database manager configuration parameter **numdb**. Set **numdb** as close as possible to the expected number of active databases. For example, in the case of a single database environment, the value of **numdb** determines how much CF memory is used by the single database. If **numdb** is set to 1, all CF memory is used by the single database. However, if **numdb** is set to a value greater than 1, a small amount of memory remains unusable for that single database.

In a multiple database environment, CF memory is configured automatically based on workload and available memory. When a new database is activated, databases that are already active automatically give up CF memory for the newly activated database until a workload-based distribution of CF memory is reached.

CF self-tuning memory calculates an initial starting size for a database on the first database activation. If this is a newly created database, an initial starting size is calculated for the database. For a new database, the starting size is based on:

- The maximum number of concurrently active databases that are specified by the **numdb** configuration parameter, and
- CF memory configuration parameter **cf_mem_sz**.

If this is an existing database, the starting size is based on the last configured values.

Changes resulting from the CF self-tuning operations are recorded in memory tuning log files that are in the db2dump/stmmlog directory. The log provides useful diagnostic information, including CF memory consumer parameter sizes.

When **cf_db_mem_sz** is set to AUTOMATIC, CF self-tuning memory ensures that the database uses a reasonable portion of the CF memory and attempts to activate the database with the parameter values that the CF memory consumer parameters were set to. CF self-tuning memory monitors the memory consumer parameters,

determines the new values, and dynamically resizes the affected configuration parameters. Manual intervention is not required. These small but frequent changes are required to satisfy changing CF memory needs. If you have more than one database, in the case of CF memory contention, CF self-tuning memory makes changes towards a workload-based distribution for the active databases.

When **cf_db_mem_sz** is set to a fixed value, you specify the amount of memory that a database can take from CF memory. CF self-tuning memory works within the CF memory that is limited by this database memory parameter.

To see the actual values of the CF memory configuration parameters, run the **GET DB CFG SHOW DETAIL** command. For example:

```
      Database Configuration for Database

 Description                                   Parameter   Current Value             Delayed Value
 ------------------------------------------------------------------------------------------------------
...
 CF Resource Configuration:

 CF self-tuning memory                                    = ON
 CF database memory size (4KB)            (CF_DB_MEM_SZ) = AUTOMATIC(8383744)         AUTOMATIC(8383744)
 Group buffer pool size (4KB)                (CF_GBP_SZ) = AUTOMATIC(6589696)         AUTOMATIC(6589696)
 Global lock memory size (4KB)              (CF_LOCK_SZ) = AUTOMATIC(1257728)         AUTOMATIC(1257728)
 Shared communication area size (4KB)        (CF_SCA_SZ) = AUTOMATIC(135245)          AUTOMATIC(135245)
 Smart array size (4KB)                     (CF_LIST_SZ) = AUTOMATIC(315571)          AUTOMATIC(315571)
```

Parameter combinations are summarized in this table:

Table 3. Parameter combinations

CF memory consumer parameters **(cf_gbp_sz, cf_lock_sz, cf_sca_sz)**	**CF database memory parameter** **cf_db_mem_sz**	**CF self-tuning memory** **(on or off**
AUTOMATIC	AUTOMATIC	Set to or remains ON for the database. **cf_db_mem_sz** and the three memory consumer parameters are tuned.
AUTOMATIC	Fixed value	Set to or remains ON for the database. **cf_db_mem_sz** is not tuned.The three memory consumer parameters are tuned.
Fixed value	Fixed value	Turned off for the database. There is no CF self-tuning memory for this database.

When explicitly disabled by changing the registry variable **DB2_DATABASE_CF_MEMORY** from AUTO to a numeric value, CF memory tuning is turned off for all databases that are activated. However, the CF memory usage by the database and the CF memory consumer parameters is not adjusted immediately for these databases. The current CF memory sizes remain at the current level. A database manager restart is required.

CF self-tuning memory in a multiple database environment with cf_db_mem_sz set to AUTOMATIC

If you are running in an environment with one database with database manager configuration parameter **numdb** set to 2 (or higher), when CF self-tuning memory is turned on, that database can use almost all CF memory. (Some memory is reserved for additional database activation.) Later when another database is added, after that database is started, the already active database automatically gives up CF memory for the newly activated database until needed, or until a workload-based distribution of CF memory is reached.

For example:

- **DB2_DATABASE_CF_MEMORY** is set to AUTO
- **numdb** is set to 3, indicating a maximum number of three active databases in the instance
- **cf_db_mem_sz** is set to AUTOMATIC
- CF memory consumer parameters (**cf_gbp_sz**, **cf_lock_sz**, or **cf_sca_sz**) are set to AUTOMATIC

When the first database is activated, since only one database is activated, that database can use almost all CF memory. Later, when another database is activated, the two databases dynamically distribute the CF memory. (Again, some memory is reserved for additional database activation.) When another database is activated, all of the CF memory can be used by the databases since memory is no longer reserved for additional database activation because the **numdb** value (3) is already met.

In addition, in this case, the databases can dynamically distribute memory that is based on their need for CF memory. This results in moving memory from the database with lower need to the database with higher need. When all databases have equal demand for CF memory, each database has its one third share of CF instance memory.

CF self-tuning memory in a multiple database environment with cf_db_mem_sz set to a fixed value, and memory consumer parameters set to AUTOMATIC

If you are running in an environment with two databases, when CF self-tuning memory is turned on, each database consumes the percentage of CF memory as specified by the fixed values.

For example:

- **DB2_DATABASE_CF_MEMORY** is set to AUTO
- **numdb** is set to 2, indicating a maximum number of two active databases in the instance
- **cf_db_mem_sz** is set to a fixed value:
  - The first database has **cf_db_mem_sz** set to a fixed value that is 70% of **cf_mem_sz**
  - The second database has **cf_db_mem_sz** set to a fixed value that is 30% of **cf_mem_sz**
- CF memory consumer parameters (**cf_gbp_sz**, **cf_lock_sz**, or **cf_sca_sz**) are set to AUTOMATIC

When the first database is activated, it consumes 70% of CF memory. Later, when the second database is activated, it consumes 30% of CF memory. The two databases do not trade memory with each other. However, for each database, its CF memory is dynamically distributed between the CF memory consumer parameters that are associated with the database (**cf_gbp_sz**, **cf_lock_sz**, or **cf_sca_sz**).

Self-tuning memory in partitioned database environments:

When using the self-tuning memory feature in partitioned database environments, there are a few factors that determine whether the feature will tune the system appropriately.

When self-tuning memory is enabled for partitioned databases, a single database partition is designated as the tuning partition, and all memory tuning decisions are based on the memory and workload characteristics of that database partition. After tuning decisions on that partition are made, the memory adjustments are distributed to the other database partitions to ensure that all database partitions maintain similar configurations.

The single tuning partition model assumes that the feature will be used only when all of the database partitions have similar memory requirements. Use the following guidelines when determining whether to enable self-tuning memory on your partitioned database.

Cases where self-tuning memory for partitioned databases is recommended

When all database partitions have similar memory requirements and are running on similar hardware, self-tuning memory can be enabled without any modifications. These types of environments share the following characteristics:

- All database partitions are on identical hardware, and there is an even distribution of multiple logical database partitions to multiple physical database partitions
- There is a perfect or near-perfect distribution of data
- Workloads are distributed evenly across database partitions, meaning that no database partition has higher memory requirements for one or more heaps than any of the others

In such an environment, if all database partitions are configured equally, self-tuning memory will properly configure the system.

Cases where self-tuning memory for partitioned databases is recommended with qualification

In cases where most of the database partitions in an environment have similar memory requirements and are running on similar hardware, it is possible to use self-tuning memory as long as some care is taken with the initial configuration. These systems might have one set of database partitions for data, and a much smaller set of coordinator partitions and catalog partitions. In such environments, it can be beneficial to configure the coordinator partitions and catalog partitions differently than the database partitions that contain data.

Self-tuning memory should be enabled on all of the database partitions that contain data, and one of these database partitions should be designated as the tuning partition. And because the coordinator and catalog partitions might be configured differently, self-tuning memory should be disabled on those partitions. To disable self-tuning memory on the coordinator and catalog partitions, set the **self_tuning_mem** database configuration parameter on these partitions to OFF.

Cases where self-tuning memory for partitioned databases is not recommended

If the memory requirements of each database partition are different, or if different database partitions are running on significantly different hardware, it is good

practice to disable the self-tuning memory feature. You can disable the feature by setting the **self_tuning_mem** database configuration parameter to OFF on all partitions.

Comparing the memory requirements of different database partitions

The best way to determine whether the memory requirements of different database partitions are sufficiently similar is to consult the snapshot monitor. If the following snapshot elements are similar on all database partitions (differing by no more than 20%), the memory requirements of the database partitions can be considered sufficiently similar.

Collect the following data by issuing the command: `get snapshot for database on <dbname>`

```
Locks held currently                        = 0
Lock waits                                  = 0
Time database waited on locks (ms)          = 0
Lock list memory in use (Bytes)             = 4968
Lock escalations                            = 0
Exclusive lock escalations                  = 0

Total Shared Sort heap allocated            = 0
Shared Sort heap high water mark            = 0
Post threshold sorts (shared memory)        = 0
Sort overflows                              = 0

Package cache lookups                       = 13
Package cache inserts                       = 1
Package cache overflows                     = 0
Package cache high water mark (Bytes)       = 655360

Number of hash joins                        = 0
Number of hash loops                        = 0
Number of hash join overflows               = 0
Number of small hash join overflows         = 0
Post threshold hash joins (shared memory)   = 0

Number of OLAP functions                    = 0
Number of OLAP function overflows           = 0
Active OLAP functions                       = 0
```

Collect the following data by issuing the command: `get snapshot for bufferpools on <dbname>`

```
Buffer pool data logical reads              = 0
Buffer pool data physical reads             = 0
Buffer pool index logical reads             = 0
Buffer pool index physical reads            = 0
Total buffer pool read time (milliseconds) = 0
Total buffer pool write time (milliseconds)= 0
```

Using self-tuning memory in partitioned database environments:

When self-tuning memory is enabled in partitioned database environments, there is a single database partition (known as the *tuning partition*) that monitors the memory configuration and propagates any configuration changes to all other database partitions to maintain a consistent configuration across all the participating database partitions.

The tuning partition is selected on the basis of several characteristics, such as the number of database partitions in the partition group and the number of buffer pools.

- To determine which database partition is currently specified as the tuning partition, call the **ADMIN_CMD** procedure as follows:

  ```
  CALL SYSPROC.ADMIN_CMD('get stmm tuning dbpartitionnum')
  ```

- To change the tuning partition, call the **ADMIN_CMD** procedure as follows:

  ```
  CALL SYSPROC.ADMIN_CMD('update stmm tuning dbpartitionnum <partitionnum>')
  ```

  The tuning partition is updated asynchronously or at the next database startup. To have the memory tuner automatically select the tuning partition, enter -1 for the *partitionnum* value.

Starting the memory tuner in partitioned database environments

In a partitioned database environment, the memory tuner will start only if the database is activated by an explicit **ACTIVATE DATABASE** command, because self-tuning memory requires that all partitions be active.

Disabling self-tuning memory for a specific database partition

- To disable self-tuning memory for a subset of database partitions, set the **self_tuning_mem** database configuration parameter to OFF for those database partitions.
- To disable self-tuning memory for a subset of the memory consumers that are controlled by configuration parameters on a specific database partition, set the value of the relevant configuration parameter or the buffer pool size to MANUAL or to some specific value on that database partition. It is recommended that self-tuning memory configuration parameter values be consistent across all running partitions.
- To disable self-tuning memory for a particular buffer pool on a specific database partition, issue the ALTER BUFFERPOOL statement, specifying a size value and the partition on which self-tuning memory is to be disabled.

  An ALTER BUFFERPOOL statement that specifies the size of a buffer pool on a particular database partition will create an exception entry (or update an existing entry) for that buffer pool in the SYSCAT.BUFFERPOOLDBPARTITIONS catalog view. If an exception entry for a buffer pool exists, that buffer pool will not participate in self-tuning operations when the default buffer pool size is set to AUTOMATIC. To remove an exception entry so that a buffer pool can be enabled for self tuning:

  1. Disable self tuning for this buffer pool by issuing an ALTER BUFFERPOOL statement, setting the buffer pool size to a specific value.
  2. Issue another ALTER BUFFERPOOL statement to set the size of the buffer pool on this database partition to the default.
  3. Enable self tuning for this buffer pool by issuing another ALTER BUFFERPOOL statement, setting the buffer pool size to AUTOMATIC.

Enabling self-tuning memory in nonuniform environments

Ideally, data should be distributed evenly across all database partitions, and the workload that is run on each partition should have similar memory requirements. If the data distribution is skewed, so that one or more of your database partitions contain significantly more or less data than other database partitions, these anomalous database partitions should not be enabled for self tuning. The same is true if the memory requirements are skewed across the database partitions, which can happen, for example, if resource-intensive sorts are only performed on one partition, or if some database partitions are associated with different hardware and more available memory than others. Self tuning memory can still be enabled on

some database partitions in this type of environment. To take advantage of self-tuning memory in environments with skew, identify a set of database partitions that have similar data and memory requirements and enable them for self tuning. Memory in the remaining partitions should be configured manually.

Buffer pool management

A buffer pool provides working memory and cache for database pages.

Buffer pools improve database system performance by allowing data to be accessed from memory instead of from disk. Because most page data manipulation takes place in buffer pools, configuring buffer pools is the single most important tuning area.

When an application accesses a table row, the database manager looks for the page containing that row in the buffer pool. If the page cannot be found there, the database manager reads the page from disk and places it in the buffer pool. The data can then be used to process the query.

Memory is allocated for buffer pools when a database is activated. The first application to connect might cause an implicit database activation. Buffer pools can be created, re-sized, or dropped while the database manager is running. The ALTER BUFFERPOOL statement can be used to increase the size of a buffer pool. By default, and if sufficient memory is available, the buffer pool is re-sized as soon as the statement executes. If sufficient memory is unavailable when the statement executes, memory is allocated when the database reactivates. If you decrease the size of the buffer pool, memory is deallocated when the transaction commits. Buffer pool memory is freed when the database deactivates.

To ensure that an appropriate buffer pool is available in all circumstances, Db2 creates small system buffer pools, one with each of the following page sizes: 4 KB, 8 KB, 16 KB, and 32 KB. The size of each buffer pool is 16 pages. These buffer pools are hidden; they are not in the system catalog or in the buffer pool system files. You cannot use or alter them directly, but Db2 uses these buffer pools in the following circumstances:

- When a specified buffer pool is not started because it was created using the DEFERRED keyword, or when a buffer pool of the required page size is inactive because insufficient memory is available to create it

  A message is written to the administration notification log. If necessary, table spaces are remapped to a system buffer pool. Performance might be drastically reduced.
- When buffer pools cannot be brought up during a database connection attempt

  This problem is likely to have a serious cause, such as an out-of-memory condition. Although Db2 will continue to be fully functional because of the system buffer pools, performance will degrade drastically. You should address this problem immediately. You will receive a warning when this occurs, and a message is written to the administration notification log.

When you create a buffer pool, the page size will be the one specified when the database was created, unless you explicitly specify a different page size. Because pages can be read into a buffer pool only if the table space page size is the same as the buffer pool page size, the page size of your table spaces should determine the page size that you specify for buffer pools. You cannot alter the page size of a buffer pool after you create it.

The memory tracker, which you can invoke by issuing the **db2mtrk** command, enables you to view the amount of database memory that has been allocated to buffer pools. You can also use the **GET SNAPSHOT** command and examine the current size of the buffer pools (the value of the **bp_cur_buffsz** monitor element).

The buffer pool priority for activities can be controlled as part of the larger set of workload management functionality provided by the Db2 workload manager. For more information, see "Introduction to Db2 workload manager concepts" and "Buffer pool priority of service classes".

Buffer pool management of data pages:

Buffer pool pages can be either in-use or not, and dirty or clean.

- *In-use pages* are pages that are currently being read or updated. If a page is being updated, it can only be accessed by the updater. However, if the page is not being updated, there can be numerous concurrent readers.
- *Dirty pages* contain data that has been changed but not yet written to disk.

Pages remain in the buffer pool until the database shuts down, the space occupied by a page is required for another page, or the page is explicitly purged from the buffer pool, for example, as part of dropping an object. The following criteria determine which page is removed when another page requires its space:

- How recently was the page referenced?
- What is the probability that the page will be referenced again?
- What type of data does the page contain?
- Was the page changed in memory but not written out to disk?

You can use the FLUSH BUFFERPOOLS statement to reduce the recovery window of a database in the event of a crash or before database operations such as online backups. For more information, see the FLUSH BUFFERPOOLS statement.

Changed pages are always written out to disk before being overwritten. Changed pages that are written out to disk are not automatically removed from the buffer pool unless the space is needed.

Page-cleaner agents

In a well-tuned system, it is usually the page-cleaner agents that write changed or dirty pages to disk. Page-cleaner agents perform I/O as background processes and allow applications to run faster because their agents can perform actual transaction work. Page-cleaner agents are sometimes referred to as *asynchronous page cleaners* or *asynchronous buffer writers*, because they are not coordinated with the work of other agents and work only when required.

To improve performance for update-intensive workloads, you might want to enable *proactive page cleaning*, whereby page cleaners behave more proactively in choosing which dirty pages get written out at any given point in time. This is particularly true if snapshots reveal that there are a significant number of synchronous data-page or index-page writes in relation to the number of asynchronous data-page or index-page writes.

Figure 19 on page 104 illustrates how the work of managing the buffer pool can be shared between page-cleaner agents and database agents.

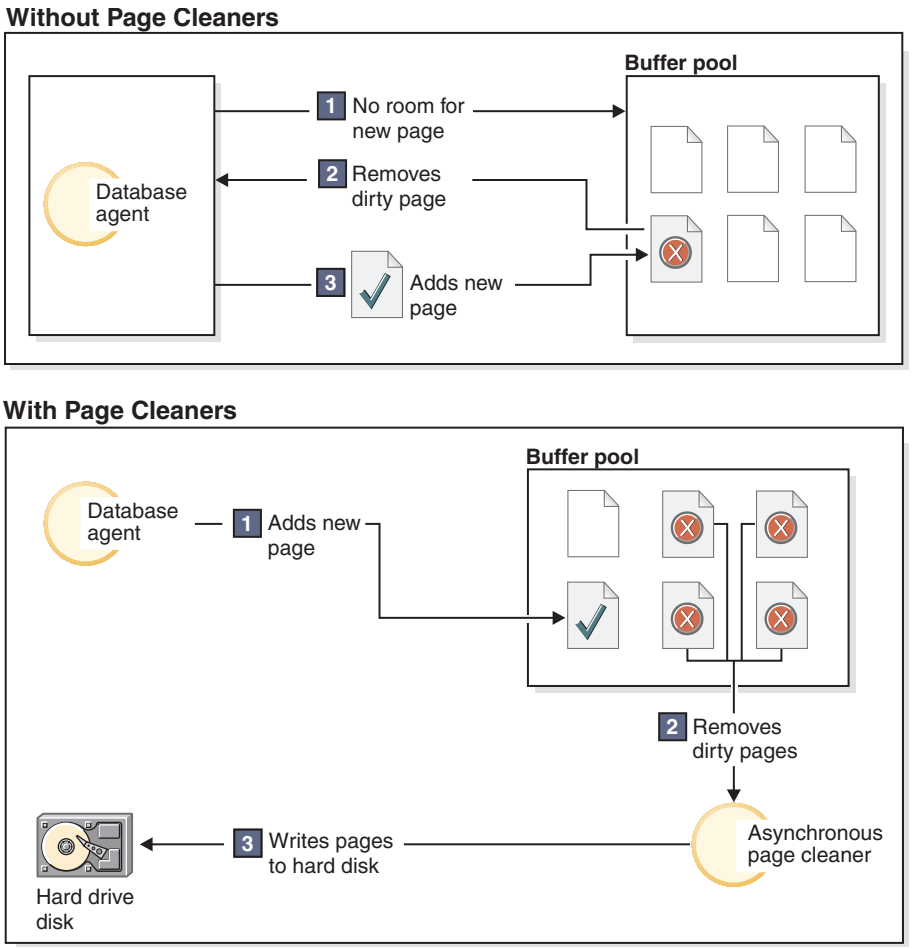


Figure 19. Asynchronous page cleaning. Dirty pages are written out to disk.

Page cleaning and fast recovery

Database recovery after a system crash is faster if more pages have been written to disk, because the database manager can rebuild more of the buffer pool from disk than by replaying transactions from the database log files.

The size of the log that must be read during recovery is the difference between the location of the following records in the log:

- The most recently written log record
- The log record that describes the oldest change to data in the buffer pool

Page cleaners write dirty pages to disks in such a manner that dirty pages are buffered in memory for up to **page_age_trgt_mcr** number of seconds.

To minimize log read time during recovery, use the database system monitor to track the number of times that page cleaning is performed. The **pool_lsn_gap_clns** (buffer pool log space cleaners triggered) monitor element provides this information if you have not enabled proactive page cleaning for your database. If you have enabled proactive page cleaning, this condition should not occur, and the value of **pool_lsn_gap_clns** is 0.

The **log_held_by_dirty_pages** monitor element can be used to determine whether the page cleaners are not cleaning enough pages to meet the recovery criteria set

by the user. If **log_held_by_dirty_pages** is consistently and significantly greater than **logfilsiz** * **softmax**, then either more page cleaners are required, or **softmax** needs to be adjusted.

Management of multiple database buffer pools:

Although each database requires at least one buffer pool, you can create several buffer pools, each of a different size or with a different page size, for a single database that has table spaces of more than one page size.

You can use the ALTER BUFFERPOOL statement to resize a buffer pool.

A new database has a default buffer pool called IBMDEFAULTBP, with a default page size that is based on the page size that was specified at database creation time. The default page size is stored as an informational database configuration parameter called **pagesize**. When you create a table space with the default page size, and if you do not assign it to a specific buffer pool, the table space is assigned to the default buffer pool. You can resize the default buffer pool and change its attributes, but you cannot drop it.

Page sizes for buffer pools

After you create or upgrade a database, you can create additional buffer pools. If you create a database with an 8-KB page size as the default, the default buffer pool is created with the default page size (in this case, 8 KB). Alternatively, you can create a buffer pool with an 8-KB page size, as well as one or more table spaces with the same page size. This method does not require that you change the 4-KB default page size when you create the database. You cannot assign a table space to a buffer pool that uses a different page size.

Note: If you create a table space with a page size greater than 4 KB (such as 8 KB, 16 KB, or 32 KB), you need to assign it to a buffer pool that uses the same page size. If this buffer pool is currently not active, Db2 attempts to assign the table space temporarily to another active buffer pool that uses the same page size, if one exists, or to one of the default system buffer pools that Db2 creates when the first client connects to the database. When the database is activated again, and the originally specified buffer pool is active, Db2 assigns the table space to that buffer pool.

If, when you create a buffer pool with the CREATE BUFFERPOOL statement, you do not specify a size, the buffer pool size is set to AUTOMATIC and is managed by Db2. To change the bufferpool size later, use the ALTER BUFFERPOOL statement.

In a partitioned database environment, each buffer pool for a database has the same default definition on all database partitions, unless it was specified otherwise in the CREATE BUFFERPOOL statement, or the bufferpool size for a particular database partition was changed by the ALTER BUFFERPOOL statement.

Advantages of large buffer pools

Large buffer pools provide the following advantages:

- They enable frequently requested data pages to be kept in the buffer pool, which allows quicker access. Fewer I/O operations can reduce I/O contention, thereby providing better response time and reducing the processor resource needed for I/O operations.

- They provide the opportunity to achieve higher transaction rates with the same response time.
- They prevent I/O contention for frequently used disk storage devices, such as those that store the catalog tables and frequently referenced user tables and indexes. Sorts required by queries also benefit from reduced I/O contention on the disk storage devices that contain temporary table spaces.

Advantages of many buffer pools

Use only a single buffer pool if any of the following conditions apply to your system:

- The total buffer pool space is less than 10 000 4-KB pages
- Persons with the application knowledge to perform specialized tuning are not available
- You are working on a test system

In all other circumstances, and for the following reasons, consider using more than one buffer pool:

- Temporary table spaces can be assigned to a separate buffer pool to provide better performance for queries (especially sort-intensive queries) that require temporary storage.
- If data must be accessed repeatedly and quickly by many short update-transaction applications, consider assigning the table space that contains the data to a separate buffer pool. If this buffer pool is sized appropriately, its pages have a better chance of being found, contributing to a lower response time and a lower transaction cost.
- You can isolate data into separate buffer pools to favor certain applications, data, and indexes. For example, you might want to put tables and indexes that are updated frequently into a buffer pool that is separate from those tables and indexes that are frequently queried but infrequently updated.
- You can use smaller buffer pools for data that is accessed by seldom-used applications, especially applications that require very random access into a very large table. In such cases, data need not be kept in the buffer pool for longer than a single query. It is better to keep a small buffer pool for this type of data, and to free the extra memory for other buffer pools.

After separating your data into different buffer pools, good and relatively inexpensive performance diagnosis data can be produced from statistics and accounting traces.

The self-tuning memory manager (STMM) is ideal for tuning systems that have multiple buffer pools.

Buffer pool memory allocation at startup

When you create a buffer pool or alter a buffer pool, the total memory that is required by all buffer pools must be available to the database manager so that all of the buffer pools can be allocated when the database starts. If you create or alter buffer pools while the database manager is online, additional memory should be available in database global memory. If you specify the DEFERRED keyword when you create a new buffer pool or increase the size of an existing buffer pool, and the required memory is unavailable, the database manager executes the changes the next time the database is activated.

If this memory is not available when the database starts, the database manager uses only the system buffer pools (one for each page size) with a minimal size of 16 pages, and a warning is returned. The database continues in this state until its configuration is changed and the database can be fully restarted. Although performance might be suboptimal, you can connect to the database, re-configure the buffer pool sizes, or perform other critical tasks. When these tasks are complete, restart the database. Do not operate the database for an extended time in this state.

To avoid starting the database with system buffer pools only, use the **DB2_OVERRIDE_BPF** registry variable to optimize use of the available memory.

Proactive page cleaning:

You can configure page cleaners on your system to be proactive with the **DB2_USE_ALTERNATE_PAGE_CLEANING** registry variable. When you set the **DB2_USE_ALTERNATE_PAGE_CLEANING** registry variable to `ON`, page cleaners behave more proactively in choosing which dirty pages get written out.

This proactive page cleaning method differs from the default page cleaning method in two major ways:

- Page cleaners no longer respond to the value of the **chngpgs_thresh** database configuration parameter.

  When the number of good victim pages is less than an acceptable value, page cleaners search the entire buffer pool, writing out potential victim pages and informing the agents of the location of these pages.
- Page cleaners now proactively handle log sequence number (LSN) gaps, and no longer responding to LSN gap triggers issued by the logger.

  When the oldest dirty page in the buffer pool exceeds the configured time for the **page_age_trgt_mcr** database configuration parameter, the database is said to be experiencing an *LSN gap*.

  Under the default page cleaning method, a logger that detects an LSN gap triggers the page cleaners to write out all the pages that are contributing to the LSN gap; that is, the page cleaners write out those pages that are older than what is allowed by the value of **page_age_trgt_mcr**. Page cleaners alternate between idleness and bursts of activity writing large numbers of pages. This can result in saturation of the I/O subsystem, which then affects other agents that are reading or writing pages. Moreover, by the time that an LSN gap is detected, the page cleaners might not be able to clean fast enough, and Db2 might run out of log space.

  The proactive page cleaning method modulates this behavior by distributing the same number of writes over a longer period of time. The page cleaners do this by cleaning not only the pages that are contributing to an LSN gap, but also pages that are likely to contribute to an impending LSN gap, based on the current level of activity.

To use the new page cleaning method, set the **DB2_USE_ALTERNATE_PAGE_CLEANING** registry variable to on.

Improving update performance:

When an agent updates a page, the database manager uses a protocol to minimize the I/O that is required by the transaction and to ensure recoverability.

This protocol includes the following steps:

1. The page that is to be updated is pinned and latched with an exclusive lock. A log record is written to the log buffer, describing how to undo and redo the change. As part of this action, a log sequence number (LSN) is obtained and stored in the header of the page that is being updated.
2. The update is applied to the page.
3. The page is unlatched. The page is considered to be "dirty", because changes to the page have not yet been written to disk.
4. The log buffer is updated. Both data in the log buffer and the dirty data page are written to disk.

For better performance, these I/O operations are delayed until there is a lull in system load, or until they are necessary to ensure recoverability or to limit recovery time. More specifically, a dirty page is written to disk when:

- Another agent chooses it as a victim
- A page cleaner works on the page. This can occur when:
  - Another agent chooses the page as a victim
  - The **chngpgs_thresh** database configuration parameter value is exceeded, causing asynchronous page cleaners to wake up and write changed pages to disk. If proactive page cleaning is enabled for the database, this value is irrelevant and does not trigger page cleaning.
  - The **page_age_trgt_mcr** database configuration parameter value is exceeded, causing asynchronous page cleaners to wake up and write changed pages to disk. If proactive page cleaning is enabled for the database, and the number of page cleaners has been properly configured for the database, this value should never be exceeded.
  - The number of clean pages drops too low. Page cleaners only react to this condition under proactive page cleaning.
  - A dirty page currently contributes to, or is expected to contribute to an LSN gap condition. Page cleaners only react to this condition under proactive page cleaning.
- The page is part of a table that was defined with the NOT LOGGED INITIALLY clause, and the update is followed by a COMMIT statement. When the COMMIT statement executes, all changed pages are flushed to disk to ensure recoverability.

Prefetching data into the buffer pool:

Prefetching pages means that one or more pages are retrieved from disk in the expectation that they will be required by an application.

Prefetching index and data pages into the buffer pool can help to improve performance by reducing I/O wait times. In addition, parallel I/O enhances prefetching efficiency.

There are three categories of prefetching:

- *Sequential prefetching* reads consecutive pages into the buffer pool before the pages are required by the application.
- *Readahead prefetching* looks ahead in the index to determine the exact pages that ISCAN-FETCH and index scan operations access, and prefetches them.
- *List prefetching* (sometimes called *list sequential prefetching*) prefetches a set of nonconsecutive data pages efficiently.

Smart data prefetching is an approach where either sequential detection or readahead prefetching is used, depending on the degree of data clustering. Sequential detection prefetching is used when the data pages are stored sequentially, and readahead prefetching is used when the data pages are badly clustered. Smart data prefetching enables the database system to capitalize on the potential performance benefits of data stored in sequential pages, while also enabling badly clustered data to be prefetched efficiently. Since badly clustered data is no longer as detrimental to query performance, this reduces the need for an expensive operation like a table reorganization.

Smart index prefetching is an approach where either sequential detection or readahead prefetching is used, depending on the density of the indexes. Sequential detection prefetching is used when indexes are stored sequentially, and readahead prefetching is used when the indexes have a low density. Smart index prefetching enables the database system to capitalize on the potential performance benefits of indexes stored sequentially, while also enabling low density indexes to be prefetched efficiently. Smart index prefetching reduces the need for an expensive operation like a index reorganization.

Smart data and index prefetching support only applies to index scan operations and does not support XML, extended, and Text Search text indexes. Smart data prefetching cannot be used during scans of global range-clustered table indexes, since they are logical indexes and not physical. Also, for smart data prefetching, if the ISCAN-FETCH scans a global range partitioned index, data readahead prefetching is not used. If index predicates are evaluated during the index scan for smart data prefetching, and the optimizer determines that not many rows qualify for that index scan, readahead prefetching is disabled. Smart index prefetching also cannot be used for range-clustered table indexes

Note: The optimizer determines the type of data or index prefetching that should be enabled for an ISCAN-FETCH or index scan operation. The following are the types of prefetching techniques for smart data and smart index prefetching:

- Sequential detection prefetching
- Readahead prefetching
- Sequential,Readahead prefetching

  For this technique both sequential and readahead prefetching is enabled, which is the default. In this technique, sequential detection prefetching is initially used until a threshold of non-prefetched pages is reached or, in some cases, if the MAXPAGES estimate made by the optimizer is exceeded. When the threshold of non-prefetched pages is reached or if the MAXPAGES estimate is exceeded, readahead prefetching is enabled.

However, in some cases the optimizer might not select any prefetching technique if the index scan is on a fully qualified key, the index is unique, and the optimizer's estimated number of pages for the index scan (MAXPAGES) is less than or equal to one. In this case, the MAXPAGES estimate is likely to be reliable, and no prefetching is required.

Prefetching data pages is different than a database manager agent read, which is used when one or a few consecutive pages are retrieved, but only one page of data is transferred to an application.

Prefetching and intrapartition parallelism

Prefetching has an important influence on the performance of intrapartition parallelism, which uses multiple subagents when scanning an index or a table. Such parallel scans result in larger data consumption rates which, in turn, require higher prefetch rates.

The cost of inadequate prefetching is higher for parallel scans than for serial scans. If prefetching does not occur during a serial scan, the query runs more slowly because the agent waits for I/O. If prefetching does not occur during a parallel scan, all subagents might need to wait while one subagent waits for I/O.

Because of its importance in this context, prefetching under intrapartition parallelism is performed more aggressively; the sequential detection mechanism tolerates larger gaps between adjacent pages, so that the pages can be considered sequential. The width of these gaps increases with the number of subagents involved in the scan.

Sequential prefetching:

Reading several consecutive pages into the buffer pool using a single I/O operation can greatly reduce your application overhead.

Prefetching starts when the database manager determines that sequential I/O is appropriate and that prefetching might improve performance. In cases such as table scans and table sorts, the database manager chooses the appropriate type of prefetching. The following example, which probably requires a table scan, would be a good candidate for sequential prefetching:

```
SELECT NAME FROM EMPLOYEE
```

With smart data and smart index prefetching, both sequential and readahead prefetching is enabled, which is the default. Sequential detection prefetching is initially used until a threshold of non-prefetched pages is reached or, in some cases, if the MAXPAGES estimate made by the optimizer is exceeded. When the threshold of non-prefetched pages is reached or if the MAXPAGES estimate is exceeded, readahead prefetching is enabled.

Sequential detection

Sometimes, it is not immediately apparent that sequential prefetching will improve performance. In such cases, the database manager can monitor I/O and activate prefetching if sequential page reading is occurring. This type of sequential prefetching, known as *sequential detection*, applies to both index and data pages. Use the **seqdetect** database configuration parameter to control whether the database manager performs sequential detection or readahead prefetching.

For example, if sequential detection is enabled, the following SQL statement might benefit from sequential prefetching:

```
SELECT NAME FROM EMPLOYEE
WHERE EMPNO BETWEEN 100 AND 3000
```

In this example, the optimizer might have started to scan the table using an index on the EMPNO column. If the table is highly clustered with respect to this index, the data page reads will be almost sequential, and prefetching might improve performance. Similarly, if many index pages must be examined, and the database

manager detects that sequential page reading of the index pages is occurring, index page prefetching is likely.

Implications of the PREFETCHSIZE option for table spaces

The PREFETCHSIZE clause on either the CREATE TABLESPACE or the ALTER TABLESPACE statement lets you specify the number of prefetched pages that will be read from the table space when data prefetching is being performed. The value that you specify (or 'AUTOMATIC') is stored in the PREFETCHSIZE column of the SYSCAT.TABLESPACES catalog view.

It is good practice to explicitly set the PREFETCHSIZE value as a multiple of the number of table space containers, the number of physical disks under each container (if a RAID device is used), and the EXTENTSIZE value (the number of pages that the database manager writes to a container before it uses a different container) for your table space. For example, if the extent size is 16 pages and the table space has two containers, you might set the prefetch size to 32 pages. If there are five physical disks per container, you might set the prefetch size to 160 pages.

The database manager monitors buffer pool usage to ensure that prefetching does not remove pages from the buffer pool if another unit of work needs them. To avoid problems, the database manager can limit the number of prefetched pages to be fewer than what was specified for the table space.

The prefetch size can have significant performance implications, particularly for large table scans. Use the database system monitor and other system monitor tools to help tune the prefetch size for your table spaces. You can gather information about whether:

- There are I/O waits for your query, using monitoring tools that are available for your operating system
- Prefetching is occurring, by looking at the **pool_async_data_reads** (buffer pool asynchronous data reads) data element provided by the database system monitor

If there are I/O waits while a query is prefetching data, you can increase the value of PREFETCHSIZE. If the prefetcher is not the cause of these I/O waits, increasing the PREFETCHSIZE value will not improve the performance of your query.

In all types of prefetching, multiple I/O operations might be performed in parallel if the prefetch size is a multiple of the extent size for the table space, and the extents are in separate containers. For better performance, configure the containers to use separate physical devices.

Block-based buffer pools for improved sequential prefetching:

Prefetching pages from disk is expensive because of I/O overhead. Throughput can be significantly improved if processing overlaps with I/O.

Most platforms provide high performance primitives that read contiguous pages from disk into noncontiguous portions of memory. These primitives are usually called *scattered read* or *vectored I/O*. On some platforms, performance of these primitives cannot compete with doing I/O in large block sizes.

By default, buffer pools are page-based, which means that contiguous pages on disk are prefetched into noncontiguous pages in memory. Sequential prefetching can be enhanced if contiguous pages can be read from disk into contiguous pages within a buffer pool.

You can create block-based buffer pools for this purpose. A block-based buffer pool consists of both a page area and a block area. The page area is required for nonsequential prefetching workloads. The block area consist of blocks; each block contains a specified number of contiguous pages, which is referred to as the *block size*.

The optimal use of a block-based buffer pool depends on the specified block size. The block size is the granularity at which I/O servers doing sequential prefetching consider doing block-based I/O. The extent is the granularity at which table spaces are striped across containers. Because multiple table spaces with different extent sizes can be bound to a buffer pool defined with the same block size, consider how the extent size and the block size will interact for efficient use of buffer pool memory. Buffer pool memory can be wasted if:

- The extent size, which determines the prefetch request size, is smaller than the block size specified for the buffer pool
- Some pages in the prefetch request are already present in the page area of the buffer pool

The I/O server allows some wasted pages in each buffer pool block, but if too many pages would be wasted, the I/O server does non-block-based prefetching into the page area of the buffer pool, resulting in suboptimal performance.

For optimal performance, bind table spaces of the same extent size to a buffer pool whose block size equals the table space extent size. Good performance can be achieved if the extent size is larger than the block size, but not if the extent size is smaller than the block size.

To create block-based buffer pools, use the CREATE BUFFERPOOL or ALTER BUFFERPOOL statement.

Note: Block-based buffer pools are intended for sequential prefetching. If your applications do not use sequential prefetching, the block area of the buffer pool is wasted.

Readahead prefetching:

Readahead prefetching looks ahead in the index to determine the exact data pages and index leaf pages that ISCAN-FETCH and index scan operations will access, and prefetches them.

While readahead prefetching provides all the data and index pages needed during the index scan (and no pages that are not needed), it also requires additional resources to locate those pages. For highly sequential data and indexes, sequential detection prefetching will normally out-perform readahead prefetching.

With smart data and smart index prefetching, both sequential and readahead prefetching is enabled, which is the default. Sequential detection prefetching is initially used until a threshold of non-prefetched pages is reached or, in some cases, if the MAXPAGES estimate made by the optimizer is exceeded. When the threshold of non-prefetched pages is reached or if the MAXPAGES estimate is exceeded, readahead prefetching is enabled.

Use the **seqdetect** database configuration parameter to control whether the database manager performs readahead or sequential detection prefetching

Restrictions

If index predicates must be evaluated during an index scan, and the optimizer determines that the index predicates for a particular index scan have a compound selectivity rate less than 90% (not many rows qualify), data readahead prefetching is disabled for that index scan. Note that this is a compound selectivity taking into account all index predicates for that particular index scan. If the query optimizer enables readahead prefetching for lower predicate selectivities it might cause many unnecessary pages to be prefetched.

Data readahead prefetching is also disabled while scanning a non-partitioned index on a range-partitioned table to prevent a prefetch request from containing page references from multiple partitions.

For smart data and smart index prefetching, which can use readahead prefetching, these prefetching techniques apply only to index scan operations and do not support XML, extended, and Text Search text indexes.

List prefetching:

List prefetching (or *list sequential prefetching*) is a way to access data pages efficiently, even when those pages are not contiguous.

List prefetching can be used in conjunction with either single or multiple index access.

If the optimizer uses an index to access rows, it can defer reading the data pages until all of the row identifiers (RIDs) have been obtained from the index. For example, the optimizer could perform an index scan to determine the rows and data pages to retrieve.

```
INDEX IX1:  NAME    ASC,
            DEPT    ASC,
            MGR     DESC,
            SALARY  DESC,
            YEARS   ASC
```

And then use the following search criteria:

```
WHERE NAME BETWEEN 'A' and 'I'
```

If the data is not clustered according to this index, list prefetching includes a step that sorts the list of RIDs that were obtained from the index scan.

Dynamic list prefetching:

Dynamic list prefetching is used to prefetch only those pages that are accessed while a specific portion of a table is scanned.

This prefetch method maximizes the number of pages that are retrieved by asynchronous prefetching (while it minimizes synchronous reads) by queuing work until the required pages are loaded into the buffer pool.

The number of pages that each subagent can prefetch simultaneously is limited by the prefetch size of the table space that is being accessed (PREFETCHSIZE). The

prefetch size can have significant performance implications, particularly for large table scans. You can use the PREFETCHSIZE clause on either the CREATE TABLESPACE or the ALTER TABLESPACE statement to specify the number of prefetched pages that are read from the table space when prefetching is being done. The value that you specify (or AUTOMATIC) is stored in the PREFETCHSIZE column of the SYSCAT.TABLESPACES catalog view. Although dynamic list prefetching typically prefetches up to PREFETCHSIZE pages at a time, this amount of prefetching might not always be possible and in some cases, the PREFETCHSIZE value might be automatically adjusted for performance reasons. If you observe I/O waits while a query is prefetching data by using dynamic list prefetching, try increasing the value of PREFETCHSIZE. Setting PREFETCHSIZE to a large value might increase memory requirements.

I/O server configuration for prefetching and parallelism:

To enable prefetching, the database manager starts separate threads of control, known as *I/O servers*, to read data pages.

As a result, query processing is divided into two parallel activities: data processing (CPU) and data page I/O. The I/O servers wait for prefetch requests from the CPU activity. These prefetch requests contain a description of the I/O that is needed to satisfy the query.

Configuring enough I/O servers (with the **num_ioservers** database configuration parameter) can greatly enhance the performance of queries that can benefit from prefetching. To maximize the opportunity for parallel I/O, set **num_ioservers** to at least the number of physical disks in the database.

It is better to overestimate than to underestimate the number of I/O servers. If you specify extra I/O servers, these servers are not used, and their memory pages are paged out with no impact on performance. Each I/O server process is numbered. The database manager always uses the lowest numbered process, and as a result, some of the higher numbered processes might never be used.

To estimate the number of I/O servers that you might need, consider the following:

- The number of database agents that could be writing prefetch requests to the I/O server queue concurrently
- The highest degree to which the I/O servers can work in parallel

Consider setting the value of **num_ioservers** to AUTOMATIC so that the database manager can choose intelligent values based on the system configuration.

Illustration of prefetching with parallel I/O:

I/O servers are used to prefetch data into a buffer pool.

This process is shown in Figure 20 on page 115.

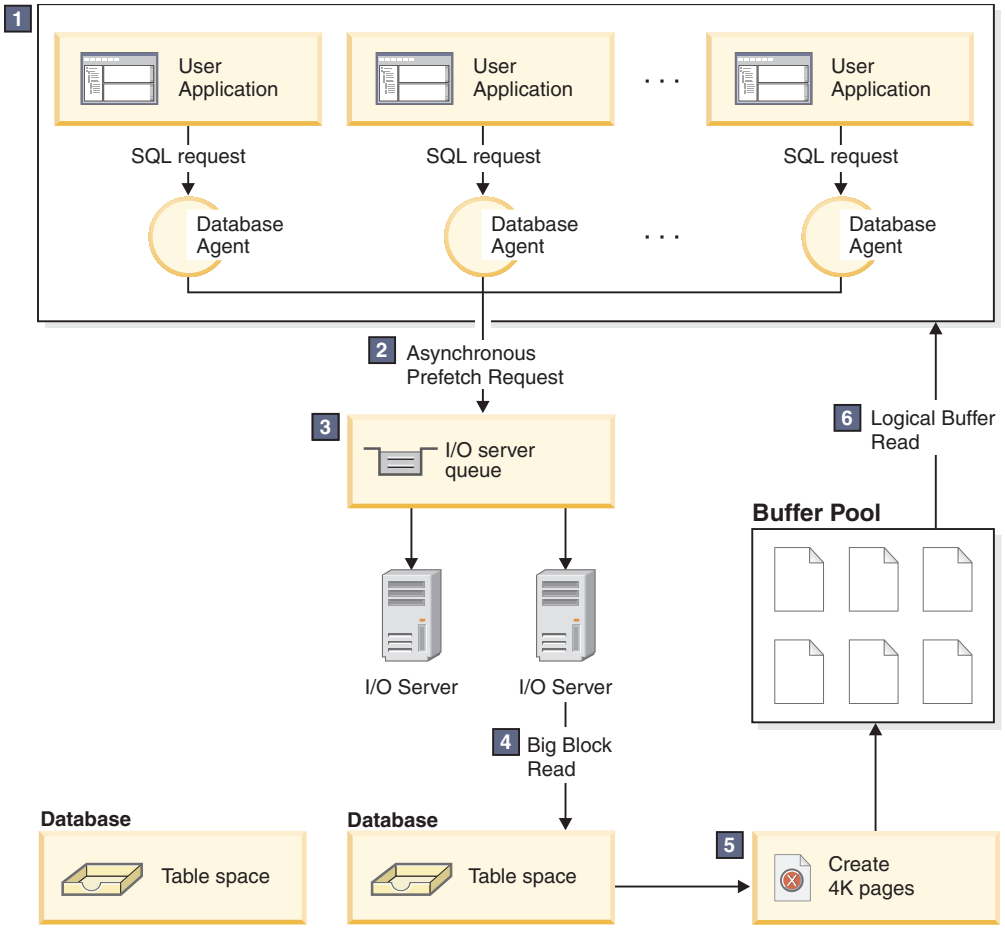


Figure 20. Prefetching data using I/O servers

1 The user application passes the request to the database agent that has been assigned to the user application by the database manager.

2, **3** The database agent determines that prefetching should be used to obtain the data that is required to satisfy the request, and writes a prefetch request to the I/O server queue.

4, **5** The first available I/O server reads the prefetch request from the queue and then reads the data from the table space into the buffer pool. The number of I/O servers that can simultaneously fetch data from a table space depends on the number of prefetch requests in the queue and the number of I/O servers specified by the **num_ioservers** database configuration parameter.

6 The database agent performs the necessary operations on the data pages in the buffer pool and returns the result to the user application.

Parallel I/O management:

If multiple containers exist for a table space, the database manager can initiate *parallel I/O*, whereby the database manager uses multiple I/O servers to process the I/O requirements of a single query.

Each I/O server processes the I/O workload for a separate container, so that several containers can be read in parallel. Parallel I/O can result in significant improvements in I/O throughput.

Although a separate I/O server can handle the workload for each container, the actual number of I/O servers that can perform parallel I/O is limited to the number of physical devices over which the requested data is spread. For this reason, you need as many I/O servers as physical devices.

Parallel I/O is initiated differently in the following cases:

- For *sequential prefetching*, parallel I/O is initiated when the prefetch size is a multiple of the extent size for a table space. Each prefetch request is divided into smaller requests along extent boundaries. These small requests are then assigned to different I/O servers.
- For *readahead prefetching*, each list of data pages is divided into smaller lists according to extent boundaries. These small lists are then assigned to different I/O servers.
- For *list prefetching*, each list of pages is divided into smaller lists according to the container in which the data pages are stored. These small lists are then assigned to different I/O servers.
- For *database or table space backup and restore*, the number of parallel I/O requests is equal to the backup buffer size divided by the extent size, up to a maximum value that is equal to the number of containers.
- For *database or table space restore*, the parallel I/O requests are initiated and divided the same way as what is done for sequential prefetching. The data is not restored into a buffer pool; it moves directly from the restore buffer to disk.
- When you *load* data, you can specify the level of I/O parallelism with the DISK_PARALLELISM command option. If you do not specify this option, the database manager uses a default value that is based on the cumulative number of table space containers for all table spaces that are associated with the table.

For optimal parallel I/O performance, ensure that:

- There are enough I/O servers. Specify slightly more I/O servers than the number of containers that are used for all table spaces within the database.
- The extent size and the prefetch size are appropriate for the table space. To prevent overuse of the buffer pool, the prefetch size should not be too large. An ideal size is a multiple of the extent size, the number of physical disks under each container (if a RAID device is used), and the number of table space containers. The extent size should be fairly small, with a good value being in the range of 8 to 32 pages.
- The containers reside on separate physical drives.
- All containers are the same size to ensure a consistent degree of parallelism.

  If one or more containers are smaller than the others, they reduce the potential for optimized parallel prefetching. Consider the following examples:

  – After a smaller container is filled, additional data is stored in the remaining containers, causing the containers to become unbalanced. Unbalanced containers reduce the performance of parallel prefetching, because the number of containers from which data can be prefetched might be less than the total number of containers.

  – If a smaller container is added at a later date and the data is rebalanced, the smaller container will contain less data than the other containers. Its small amount of data relative to the other containers will not optimize parallel prefetching.

  - If one container is larger and all of the other containers fill up, the larger container is the only container to store additional data. The database manager cannot use parallel prefetching to access this additional data.
- There is adequate I/O capacity when using intrapartition parallelism. On SMP machines, intrapartition parallelism can reduce the elapsed time for a query by running the query on multiple processors. Sufficient I/O capacity is required to keep each processor busy. Additional physical drives are usually required to provide that I/O capacity.

  The prefetch size must be larger for prefetching to occur at higher rates, and to use I/O capacity effectively.

  The number of physical drives required depends on the speed and capacity of the drives and the I/O bus, and on the speed of the processors.

Configuring IOCP (AIX):

Configuring I/O completion ports (IOCPs) on AIX servers is not required for the installation of Db2 software. However, this configuration step is recommended for performance purposes. You need to perform these steps on each host you want to participate in the Db2 instance. AIX 5.3 TL9 SP2 and AIX 6.1 TL2 have the I/O completion ports (IOCP) file set included as part of the base installation. However, if the minimum operating system requirements were applied using an operating system upgrade rather than using a new operating system installation, you must configure I/O completion ports (IOCP) separately.

Before you begin

Install the Db2 software by running the **db2setup** command, which enables IOCP and sets the status of the IOCP port to `Available`.

Procedure

To configure IOCP:

1. To check whether the IOCP module is installed on your system, enter the following command:

   ```
   $ lslpp -l bos.iocp.rte
   ```

   The output should be similar to that in the following example:

   ```
   Fileset                      Level     State       Description
   ----------------------------------------------------------------------------
   Path: /usr/lib/objrepos
     bos.iocp.rte               5.3.9.0   APPLIED     I/O Completion Ports API

   Path: /etc/objrepos
     bos.iocp.rte               5.3.0.50  COMMITTED   I/O Completion Ports API
   ```

2. Check whether the status of the IOCP port is `Available` by entering the following command:

   ```
   $ lsdev -Cc iocp
   ```

   The output should be as follows:

   ```
   iocp0 Available  I/O Completion Ports
   ```

3. If the IOCP port status is `Defined`, perform the following steps:
   a. Log in to the server as root.
   b. Enter the following command:

      ```
      # smitty iocp
      ```

c. Select **Change / Show Characteristics of I/O Completion Ports**.
d. At system restart, change the configured state from `Defined` to `Available`.
e. Make the state change effective by either rebooting the system or entering the **cfgmgr** command.
f. Confirm that the status of the IOCP port changed to `Available`"by entering the following command again:

```
$ lsdev -Cc iocp
```

Database deactivation behavior in first-user connection scenarios

A database is activated when a user first connects to it. In a single-partition environment, the database is loaded into memory and remains in this state until the last user disconnects. The same behavior applies to multi-partition environments, where any first-user connection activates the database on both local and catalog partitions for that database.

When the last user disconnects, the database shuts down on both local and any remote partitions where this user is the last active user connection for the database. This activation and deactivation of the database based on first connection and last disconnection is known as *implicit activation*. Activation is initiated by the first user connection, and the activation remains in effect until the user executes a CONNECT RESET (or until the user terminates or drops the connection), which results in the database being implicitly deactivated.

The process of loading a database into memory is very involved. It encompasses initialization of all database components, including buffer pools, and is the type of processing that should be minimized, particularly in performance-sensitive environments. This behavior is of particular importance in multi-partition environments, where queries that are issued from one database partition reach other partitions that contain part of the target data set. Those database partitions are activated or deactivated, depending on the connect and disconnect behavior of the user applications. When a user issues a query that reaches a database partition for the first time, the query assumes the cost of first activating that partition. When that user disconnects, the database is deactivated unless other connections were previously established against that remote partition. If the next incoming query needs to access that remote partition, the database on that partition will first have to be activated. This cost is accrued for each activation and deactivation of the database (or database partition, where applicable).

The only exception to this behavior occurs if the user chooses to explicitly activate the database by issuing the **ACTIVATE DATABASE** command. After this command completes successfully, the database remains in memory, even if the last user disconnects. This applies to both single- and multi-partition environments. To deactivate such a database, issue the **DEACTIVATE DATABASE** command. Both commands are global in scope, meaning that they will activate or deactivate the database on all database partitions, if applicable. Given the processing-intensive nature of loading a database into memory, consider explicitly activating databases by using the **ACTIVATE DATABASE** command, rather than relying on implicit activation through database connections.

Tuning sort performance

Because queries often require sorted or grouped results, proper configuration of the sort heap is crucial to good query performance.

Sorting is required when:

- No index that satisfies a requested order exists (for example, a SELECT statement that uses the ORDER BY clause)
- An index exists, but sorting would be more efficient than using the index
- An index is created
- An index is dropped, which causes index page numbers to be sorted

Elements that affect sorting

The following factors affect sort performance:

- Settings for the following configuration parameters:
  - Sort heap size (**sortheap**), which specifies the amount of memory to be used for each sort
  - Sort heap threshold (**sheapthres**) and the sort heap threshold for shared sorts (**sheapthres_shr**), which control the total amount of memory that is available for sorting across the instance
- The number of statements in a workload that require a large amount of sorting
- The presence or absence of indexes that might help avoid unnecessary sorting
- The use of application logic that does not minimize the need for sorting
- Parallel sorting, which improves sort performance, but which can only occur if the statement uses intrapartition parallelism
- Whether or not the sort is *overflowed*. If the sorted data cannot fit into the sort heap, which is a block of memory that is allocated each time a sort is performed, the data overflows into a temporary table that is owned by the database.
- Whether or not the results of the sort are *piped*. If sorted data can return directly without requiring a temporary table to store the sorted list, it is a piped sort.

  In a piped sort, the sort heap is not freed until the application closes the cursor that is associated with the sort. A piped sort can continue to use up memory until the cursor is closed.

Although a sort can be performed entirely in sort memory, this might cause excessive page swapping. In this case, you lose the advantage of a large sort heap. For this reason, you should use an operating system monitor to track changes in system paging whenever you adjust the sorting configuration parameters.

Techniques for managing sort performance

Identify particular applications and statements where sorting is a significant performance problem:

1. Set up event monitors at the application and statement level to help you identify applications with the longest total sort time.
2. Within each of these applications, find the statements with the longest *total sort time*.

   You can also search through the explain tables to identify queries that have sort operations.
3. Use these statements as input to the Design Advisor, which will identify and can create indexes to reduce the need for sorting.

You can use the self-tuning memory manager (STMM) to automatically and dynamically allocate and deallocate memory resources required for sorting. To use this feature:

- Enable self-tuning memory for the database by setting the **self_tuning_mem** configuration parameter to ON.
- Set the **sortheap** and **sheapthres_shr** configuration parameters to AUTOMATIC.
- Set the **sheapthres** configuration parameter to 0.

You can also use the database system monitor and benchmarking techniques to help set the **sortheap**, **sheapthres_shr**, and **sheapthres** configuration parameters. For each database manager and for each database:

1. Set up and run a representative workload.
2. For each applicable database, collect average values for the following performance variables over the benchmark workload period:
   - Total sort heap in use (the value of the **sort_heap_allocated** monitor element)
   - Active sorts and active hash joins (the values of the **active_sorts** and **active_hash_joins** monitor elements)
3. Set **sortheap** to the average *total sort heap in use* for each database.

   Note: If long keys are used for sorts, you might need to increase the value of the **sortheap** configuration parameter.
4. Set the **sheapthres**. To estimate an appropriate size:
   a. Determine which database in the instance has the largest **sortheap** value.
   b. Determine the average size of the sort heap for this database.

      If this is too difficult to determine, use 80% of the maximum sort heap.
   c. Set **sheapthres** to the average number of active sorts, times the average size of the sort heap computed previously. This is a recommended initial setting. You can then use benchmark techniques to refine this value.

IBM InfoSphere Optim Query Workload Tuner provides tools for improving the performance of single SQL statements and the performance of groups of SQL statements, which are called query workloads. For more information about this product, see the product overview page at http://www.ibm.com/software/data/optim/query-workload-tuner-db2-luw/index.html. In Version 3.1.1 or later, you can also use the Workload Design Advisor to perform many operations that were available in the Db2 Design Advisor wizard. For more information see the documentation for the Workload Design Advisor at http://www.ibm.com/support/knowledgecenter/SS62YD_4.1.1/com.ibm.datatools.qrytune.workloadtunedb2luw.doc/topics/genrecsdsgn.html.

Data organization

Over time, data in your tables can become fragmented, increasing the size of tables and indexes as records become distributed over more and more data pages. This can increase the number of pages that need to be read during query execution. Reorganization of tables and indexes compacts your data, reclaiming wasted space and improving data access.

Procedure

The steps to perform an index or table reorganization are as follows:

1. Determine whether you need to reorganize any tables or indexes.
2. Choose a reorganization method.
3. Perform the reorganization of identified objects.

4. Optional: Monitor the progress of reorganization.
5. Determine whether or not the reorganization was successful. For offline table reorganization and any index reorganization, the operation is synchronous, and the outcome is apparent upon completion of the operation. For online table reorganization, the operation is asynchronous, and details are available from the history file.
6. Collect statistics on reorganized objects.
7. Rebind applications that access reorganized objects.

Table reorganization

After many changes to table data, logically sequential data might reside on nonsequential data pages, so that the database manager might need to perform additional read operations to access data. Also, if many rows have been deleted, additional read operations are also required. In this case, you might consider reorganizing the table to match the index and to reclaim space.

You can also reorganize the system catalog tables.

Because reorganizing a table usually takes more time than updating statistics, you could execute the **RUNSTATS** command to refresh the current statistics for your data, and then rebind your applications. If refreshed statistics do not improve performance, reorganization might help.

The following factors can indicate a need for table reorganization:

- There has been a high volume of insert, update, and delete activity against tables that are accessed by queries.
- There have been significant changes in the performance of queries that use an index with a high cluster ratio.
- Executing the **RUNSTATS** command to refresh table statistics does not improve performance.
- Output from the **REORGCHK** command indicates a need for table reorganization.

Note: With Db2 V9.7 Fix Pack 1 and later releases, higher data availability for a data partitioned table with only partitioned indexes (except system-generated XML path indexes) is achieved by reorganizing data for a specific data partition. Partition-level reorganization performs a table reorganization on a specified data partition while the remaining data partitions of the table remain accessible. The output from the **REORGCHK** command for a partitioned table contains statistics and recommendations for performing partition-level reorganizations.

REORG TABLE commands and **REORG INDEXES ALL** commands can be issued on a data partitioned table to concurrently reorganize different data partitions or partitioned indexes on a partition. When concurrently reorganizing data partitions or the partitioned indexes on a partition, users can access the unaffected partitions but cannot access the affected partitions. All the following criteria must be met to issue REORG commands that operate concurrently on the same table:

- Each REORG command must specify a different partition with the **ON DATA PARTITION** clause.
- Each REORG command must use the ALLOW NO ACCESS mode to restrict access to the data partitions.
- The partitioned table must have only partitioned indexes if issuing **REORG TABLE** commands. No nonpartitioned indexes (except system-generated XML path indexes) can be defined on the table.

In IBM Data Studio Version 3.1 or later, you can use the task assistant for reorganizing tables. Task assistants can guide you through the process of setting options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

Choosing a table reorganization method:

There are four approaches to table reorganization: CLASSIC reorganization (offline), INPLACE reorganization with FULL to recluster or reclaim space (online), INPLACE reorganization with CLEANUP OVERFLOWS to only cleanup overflows (online), and RECLAIM EXTENTS (online).

Offline, or CLASSIC reorganization is the default behavior. To specify an online reorganization operation, use the INPLACE with FULL, INPLACE with CLEANUP OVERFLOWS, or RECLAIM EXTENTS table clause of the **REORG** command.

Each approach has its advantages and drawbacks, which are summarized in the following sections. When you choose a reorganization method, consider which approach offers advantages that align with your priorities. For example, if you want to minimize the duration that the affected object is unavailable, online reorganization might be preferable. If your priority is the duration that is required for the reorganization operation, offline reorganization would be preferable.

Advantages of CLASSIC reorganization

This approach offers:

- The fastest table reorganization operations, especially if large object (LOB) or long field data is not included.
- Perfectly clustered tables and indexes upon completion.
- Indexes that are automatically rebuilt after a table is reorganized; there is no separate step for rebuilding indexes.
- The use of a temporary table space for building a shadow copy. The use of a shadow copy reduces the space requirements for the table space that contains the target table or index.
- The ability to use an index other than the existing clustering index to recluster the data.

Disadvantages of CLASSIC reorganization

This approach is characterized by:

- Limited table access; read access only during the sort and build phase of a REORG operation.
- A large space requirement for the shadow copy of the table that is being reorganized.
- Less control over the REORG process; an offline REORG operation cannot be paused and restarted.
- A large active log might be required since the entire operation is handled in a single unit of work.

Advantages of INPLACE reorganization with the FULL option

This approach offers:

- Full table access, except during the truncation phase of a REORG operation.

- More control over the REORG process, which runs asynchronously in the background, and can be paused, resumed, or stopped. For example, you can pause an in-progress REORG operation if many updates or deletes are running against the table.
- A process that can be resumed in the event of a failure.
- A reduced requirement for working storage because a table is processed incrementally.
- Immediate benefits of reorganization, even before a REORG operation completes.

Disadvantages of INPLACE reorganization with the FULL option

This approach is characterized by:

- Imperfect data or index clustering, depending on the type of transactions that accesses the table during a REORG operation.
- Poorer performance than an offline REORG operation.
- Potentially high logging requirements. These requirements depend on the number of rows that are moved, the number of indexes that are defined on the table, and the size of those indexes.
- A potential need for subsequent index reorganization because indexes are maintained, not rebuilt.
- Incomplete space reclamation, because online reorganization cannot move internal records.

Advantages of INPLACE reorganization with the CLEANUP OVERFLOWS option

This approach offers:

- Full table access.
- More control over the REORG process, which runs asynchronously in the background, and can be paused, resumed, or stopped. For example, you can pause an in-progress REORG operation if many updates or deletes are running against the table.
- A process that can be resumed in the event of a failure.
- Fixes all pointer and overflow pairs that exist in the table that improves the performance characteristics of SQL access on the table.
- A reduced requirement for working storage because a table is processed incrementally.
- Immediate benefits of reorganization, even before a REORG operation completes.
- Less overall logging and impact than an INPLACE reorganization with the FULL option.

Disadvantages of INPLACE reorganization with the CLEANUP OVERFLOWS option

This approach is characterized by:

- No benefit other than resolving pointer and overflow pairs. Use this mode only if your table has many pointer and overflow pairs and these pairs are causing performance issues.

Advantages of RECLAIM EXTENTS

This approach offers:

- Full table access.
- A process that can be resumed in the event of a failure. The work that is done up until the point of failure is not lost. The operation is resumed from the point of failure and through to completion.
- Lightweight operation.
- Frees space back to the table space that can then be used by any table space consumer.
- A reduced requirement for working storage because a table is processed incrementally.

Disadvantages of RECLAIM EXTENTS

This approach is characterized by:

- Does not recluster data.
- Does not fix all pointer and overflow pairs that exist in the table.
- Does not convert all existing rows to the current table schema.
- A potential need for subsequent index reorganization because indexes are maintained, not rebuilt.

Table 4. Comparison of online and offline reorganization

Characteristic	CLASSIC reorganization	INPLACE reorganization with FULL	INPLACE reorganization with CLEANUP OVERFLOWS	RECLAIM EXTENTS
Performance	Fast	Slow	Fast	Fast
Clustering factor of data at completion	Good	Not perfectly clustered	No clustering is done	No clustering is done
Concurrency (access to the table)	Ranges from no access to read-only	Ranges from read-only to full access	Ranges from read-only to full access	Ranges from no access to full access
Data storage space requirement	Significant	Not significant	Not significant	Not significant
Logging storage space requirement	Not significant	Might be significant	Might be significant	Might be significant
User control (ability to pause, restart process)	Less control	More control	More control	Less control as you cannot restart or pause
Recoverability	Recoverable, but might take more time than an online reorganization.	Recoverable	Recoverable	Recoverable
Index rebuilding	Done	Not done	Not done	Not done
Supported for all types of tables	Yes	No	No	No

Table 4. Comparison of online and offline reorganization (continued)

Characteristic	CLASSIC reorganization	INPLACE reorganization with FULL	INPLACE reorganization with CLEANUP OVERFLOWS	RECLAIM EXTENTS
Ability to specify an index other than the clustering index	Yes	No	No	No
Use of a temporary table space	Yes	No	No	No

Table 5. Table types that are supported for online and offline reorganization

Table type	Support offline reorganization	Support online reorganization
Multidimensional clustering tables (MDC)	Yes[1]	Yes[8]
Insert time clustering tables (ITC)	Yes[1, 7]	Yes[6, 7]
Range-clustered tables (RCT)	No[2]	No
Append mode tables	Yes	No[3]
Tables with long field or large object (LOB) data	Yes[5]	Yes[5]
System catalog tables: • SYSIBM.SYSCODEPROPERTIES • SYSIBM.SYSDATATYPES • SYSIBM.SYSNODEGROUPS • SYSIBM.SYSROUTINES • SYSIBM.SYSSEQUENCES • SYSIBM.SYSTABLES • SYSIBM.SYSVARIABLES	Yes	No

Notes:

1. Because clustering is automatically maintained through MDC block indexes, reorganization of an MDC table involves space reclamation only. No indexes can be specified. Similarly, for ITC tables, you cannot specify a reorganization with a clustering index.
2. The range area of an RCT always remains clustered.
3. Online reorganization can be run after append mode is disabled.
4. Reorganizing long field or large object (LOB) data can take a significant amount of time, and does not improve query performance. Reorganization is done only for space reclamation purposes.
5. Online table reorganization does not reorganize the LONG/LOB data, but reorganizes the other columns.
6. Online reorganization of an ITC table is supported. The reorganization is done with the existing RECLAIM EXTENTS table clause of the REORG command.
7. The RECLAIM EXTENTS table clause of the REORG command consolidates sparse extents implicitly. This consolidation leads to more space reclamation, but a longer duration for utility execution when compared to Db2 Version 10.1.
8. Not supported when RECLAIM EXTENTS is used.

Note: You can use the online table move stored procedure as an alternative approach to INPLACE reorganization. See "Moving tables online by using the ADMIN_MOVE_TABLE procedure".

Monitoring the progress of table reorganization

Information about the progress of a current table REORG operation is written to the history file. The history file contains a record for each reorganization event. To view this file, run the **LIST HISTORY** command against the database that contains the table that is reorganized.

You can also use table snapshots to monitor the progress of table REORG operations. Table reorganization monitoring data is recorded, regardless of the setting for the database system monitor table switch.

If an error occurs, an SQLCA message is written to the history file. If an INPLACE table REORG operation, the status is recorded as PAUSED.

CLASSIC (offline) table reorganization:

CLASSIC table reorganization uses a shadow copy approach, building a full copy of the table that is being reorganized.

There are four phases in a CLASSIC or offline table reorganization operation:

1. SORT - During this phase, if an index was specified on the **REORG TABLE** command, or a clustering index was defined on the table, the rows of the table are first sorted according to that index. If the INDEXSCAN option is specified, an index scan is used to sort the table; otherwise, a table scan sort is used. This phase applies only to a clustering table REORG operation. Space reclaiming REORG operations begin at the build phase.
2. BUILD - During this phase, a reorganized copy of the entire table is built, either in its table space or in a temporary table space that was specified on the **REORG TABLE** command.
3. REPLACE - During this phase, the original table object is replaced by a copy from the temporary table space, or a pointer is created to the newly built object within the table space of the table that is being reorganized.
4. RECREATE ALL INDEXES - During this phase, all indexes that were defined on the table are re-created.

You can monitor the progress of the table REORG operation and identify the current phase with the snapshot monitor or snapshot administrative views.

The locking conditions are more restrictive in offline mode than in online mode. Read access to the table is available while the copy is being built. However, exclusive access to the table is required when the original table is being replaced by the reorganized copy, or when indexes are being rebuilt.

An IX table space lock is required during the entire table REORG process. During the build phase, a U lock is acquired and held on the table. A U lock allows the lock owner to update the data in the table. Although no other application can update the data, read access is available. The U lock is upgraded to a Z lock after the replace phase starts. During this phase, no other applications can access the data. This lock is held until the table REORG operation completes.

A number of files are created by the offline reorganization process. These files are stored in your database directory. Their names are prefixed with the table space and object IDs; for example, `0030002.ROR` is the state file for a table REORG operation whose table space ID is 3 and table ID is 2.

The following list shows the temporary files that are created in a system managed space (SMS) table space during an offline table REORG operation:

- .DTR - data shadow copy file
- .LFR - long field file
- .LAR - long field allocation file
- .RLB - LOB data file
- .RBA - LOB allocation file
- .BMR - block object file for multidimensional clustering (MDC) and insert time clustering (ITC) tables

The following temporary file is created during an index REORG operation:

- .IN1 - Shadow copy file

The following list shows the temporary files that are created in the system temporary table space during the sort phase:

- .TDA - data file
- .TIX - index file
- .TLF - long field file
- .TLA - long field allocation file
- .TLB - LOB file
- .TBA - LOB allocation file
- .TBM - block object file

The files that are associated with the reorganization process must not be manually removed from your system.

Reorganizing tables offline:

Reorganizing tables offline is the fastest way to defragment your tables. Reorganization reduces the amount of space that is required for a table and improves data access and query performance.

Before you begin

You must have SYSADM, SYSCTRL, SYSMAINT, DBADM, or SQLADM authority, or CONTROL privilege on the table that is to be reorganized. You must also have a database connection to reorganize a table.

About this task

After you have identified the tables that require reorganization, you can run the reorg utility against those tables and, optionally, against any indexes that are defined on those tables.

Procedure

1. To reorganize a table using the **REORG TABLE** command, simply specify the name of the table. For example:

```
reorg table employee
```

You can reorganize a table using a specific temporary table space. For example:

```
reorg table employee use mytemp
```

You can reorganize a table and have the rows reordered according to a specific index. For example:

```
reorg table employee index myindex
```

2. To reorganize a table using an SQL CALL statement, specify the **REORG TABLE** command with the ADMIN_CMD procedure. For example:

```
call sysproc.admin_cmd ('reorg table employee')
```

3. To reorganize a table using the administrative application programming interface, call the db2Reorg API.

What to do next

After reorganizing a table, collect statistics on that table so that the optimizer has the most accurate data for evaluating query access plans.

Recovery of an offline table reorganization:

An offline table reorganization is an all-or-nothing process until the beginning of the replace phase. If your system crashes during the sort or build phase, the reorg operation is rolled back and will not be redone during crash recovery.

If your system crashes after the beginning of the replace phase, the reorg operation must complete, because all of the reorganization work has been done and the original table might no longer be available. During crash recovery, the temporary file for the reorganized table object is required, but not the temporary table space that is used for the sort. Recovery will restart the replace phase from the beginning, and all of the data in the copy object is required for recovery. There is a difference between system managed space (SMS) and database managed space (DMS) table spaces in this case: the reorganized table object in SMS must be copied from one object to the other, but the reorganized table object in DMS is simply pointed to, and the original table is dropped, if the reorganization was done in the same table space. Indexes are not rebuilt, but are marked invalid during crash recovery, and the database will follow the usual rules to determine when they are rebuilt, either at database restart or upon first index access.

If a crash occurs during the index rebuild phase, nothing is redone because the new table object already exists. Indexes are handled as described previously.

During rollforward recovery, the reorg operation is redone if the old version of the table is on disk. The rollforward utility uses the record IDs (RIDs) that are logged during the build phase to reapply the operations that created the reorganized table, repeating the build and replace phases. Indexes are handled as described previously. A temporary table space is required for a copy of the reorganized object only if a temporary table space was used originally. During rollforward recovery, multiple reorg operations can be redone concurrently (parallel recovery).

Improving the performance of offline table reorganization:

The performance of an offline table reorganization is largely determined by the characteristics of the database environment.

There is almost no difference in performance between a reorg operation that is running in ALLOW NO ACCESS mode and one that is running in ALLOW READ ACCESS mode. The difference is that during a reorg operation in ALLOW READ ACCESS mode, the utility might have to wait for other applications to complete their scans and release their locks before replacing the table. The table is unavailable during the index rebuild phase of a reorg operation that is running in either mode.

Tips for improving performance

- If there is enough space to do so, use the same table space for both the original table and the reorganized copy of the table, instead of using a temporary table space. This saves the time that is needed to copy the reorganized table from the temporary table space.
- Consider dropping unnecessary indexes before reorganizing a table so that fewer indexes need to be maintained during the reorg operation.
- Ensure that the prefetch size of the table spaces on which the reorganized table resides is set properly.
- Tune the **sortheap** and **sheapthres** database configuration parameters to control the space that is available for sorts. Because each processor will perform a private sort, the value of **sheapthres** should be at least **sortheap** x *number-of-processors*.
- Adjust the number of page cleaners to ensure that dirty index pages in the buffer pool are cleaned as soon as possible.

Inplace (online) table reorganization:

Inplace table reorganization reorganizes a table and allows full access to data in the table. The cost of this uninterrupted access to the data is a slower table REORG operation.

Starting inDb2 Cancun Release 10.5.0.4, inplace table reorganization is supported in Db2 pureScale environments.

During an inplace or online table REORG operation, portions of a table are reorganized sequentially. Data is not copied to a temporary table space; instead, rows are moved within the existing table object to reestablish clustering, reclaim free space, and eliminate overflow rows.

There are four main phases in an online table REORG operation:

1. SELECT *n* pages

   During this phase, the database manager selects a range of *n* pages, where *n* is the size of an extent with a minimum of 32 sequential pages for REORG processing.

2. Vacate the range

   The REORG utility moves all rows within this range to free pages in the table. Each row that is moved leaves behind a REORG table pointer (RP) record that contains the record ID (RID) of the row's new location. The row is placed on a free page in the table as a REORG table overflow (RO) record that contains the data. After the utility finishes moving a set of rows, it waits until all applications that are accessing data in the table are finished. These “old scanners” use old RIDs when table data is accessed. Any table access that starts during this waiting period (a “new scanner”) uses new RIDs to access the data.

After all of the old scanners are complete, the REORG utility cleans up the moved rows, deleting RP records and converting RO records into regular records.

3. Fill the range

   After all rows in a specific range are vacated, they are written back in a reorganized format, they are sorted according to any indexes that were used, and obeying any PCTFREE restrictions that were defined. When all of the pages in the range are rewritten, the next *n* sequential pages in the table are selected, and the process is repeated.

4. Truncate the table

   By default, when all pages in the table are reorganized, the table is truncated to reclaim space. If the NOTRUNCATE option is specified, the reorganized table is not truncated.

Files created during an online table REORG operation

During an online table REORG operation, an `.OLR` state file is created for each database partition. This binary file has a name whose format is `xxxxyyyy.OLR`, where *xxxx* is the table space ID and *yyyy* is the object ID in hexadecimal format. For a REORG of a range partitioned table, the file name has the format `aaaabbbbxxxxyyyy.OLR` where *aaaa* is the partition table space ID and *bbbb* is the partition object ID. This file contains the following information that is required to resume an online REORG operation from the paused state:

- The type of REORG operation
- The life log sequence number (LSN) of the table that is reorganized
- The next range to be vacated
- Whether the REORG operation is clustering the data or just reclaiming space
- The ID of the index that is being used to cluster the data

A checksum is run on the `.OLR` file. If the file becomes corrupted, causing checksum errors, or if the table LSN does not match the life LSN, a new REORG operation is initiated, and a new state file is created.

If the `.OLR` state file is deleted, the REORG process cannot resume, SQL2219N is returned, and a new REORG operation must be initiated.

The files that are associated with the reorganization process must not be manually removed from your system.

Reorganizing tables online:

An online or inplace table reorganization allows users to access a table while it is being reorganized.

Before you begin

You must have SYSADM, SYSCTRL, SYSMAINT, DBADM, or SQLADM authority, or CONTROL privilege on the table that is to be reorganized. You must also have a database connection to reorganize a table.

About this task

After you have identified the tables that require reorganization, you can run the reorg utility against those tables and, optionally, against any indexes that are defined on those tables.

Procedure

- To reorganize a table online using the **REORG TABLE** command, specify the name of the table and the **INPLACE** parameter. For example:

  ```
  reorg table employee inplace
  ```

- To reorganize a table online using an SQL CALL statement, specify the **REORG TABLE** command with the ADMIN_CMD procedure. For example:

  ```
  call sysproc.admin_cmd ('reorg table employee inplace')
  ```

- To reorganize a table online using the administrative application programming interface, call the db2Reorg API.

What to do next

After reorganizing a table, collect statistics on that table so that the optimizer has the most accurate data for evaluating query access plans.

Recovery of an online table reorganization:

The failure of an online table reorganization is often due to processing errors, such as disk full or logging errors. If an online table reorganization fails, an SQLCA message is written to the history file.

If a failure occurs during run time, the online table reorg operation is paused and then rolled back during crash recovery. You can subsequently resume the reorg operation by specifying the **RESUME** parameter on the **REORG TABLE** command. Because the process is fully logged, online table reorganization is guaranteed to be recoverable.

Under some circumstances, an online table reorg operation might exceed the limit that is set by the value of the **num_log_span** database configuration parameter. In this case, the database manager forces the reorg utility and puts it into PAUSE state. In snapshot monitor output, the state of the reorg utility appears as PAUSED.

The online table reorg pause is interrupt-driven, which means that it can be triggered either by a user (using the **PAUSE** parameter on the **REORG TABLE** command, or the **FORCE APPLICATION** command) or by the database manager in certain circumstances; for example, in the event of a system crash.

If one or more database partitions in a partitioned database environment encounters an error, the SQLCODE that is returned is the one from the first database partition that reports an error.

Pausing and restarting an online table reorganization:

An online table reorganization that is in progress can be paused and restarted by the user.

Before you begin

You must have SYSADM, SYSCTRL, SYSMAINT, DBADM, or SQLADM authority, or CONTROL privilege on the table whose online reorganization is to be paused or restarted. You must also have a database connection to pause or restart an online table reorganization.

Procedure

1. To pause an online table reorganization using the **REORG TABLE** command, specify the name of the table, the **INPLACE** parameter, and the **PAUSE** parameter. For example:

   ```
   reorg table employee inplace pause
   ```

2. To restart a paused online table reorganization, specify the **RESUME** parameter. For example:

   ```
   reorg table employee inplace resume
   ```

   When an online table reorg operation is paused, you cannot begin a new reorganization of that table. You must either resume or stop the paused operation before beginning a new reorganization process.

   Following a **RESUME** request, the reorganization process respects whatever truncation option is specified on the current **RESUME** request. For example, if the **NOTRUNCATE** parameter is not specified on the current **RESUME** request, a **NOTRUNCATE** parameter specified on the original **REORG TABLE** command, or with any previous **RESUME** requests, is ignored.

   A table reorg operation cannot resume after a restore and rollforward operation.

Locking and concurrency considerations for online table reorganization:

One of the most important aspects of online table reorganization-because it is so crucial to application concurrency-is how locking is controlled.

An online table reorg operation can hold the following locks:

- To ensure write access to table spaces, an IX lock is acquired on the table spaces that are affected by the reorg operation.
- A table lock is acquired and held during the entire reorg operation. The level of locking is dependent on the access mode that is in effect during reorganization:
  - If ALLOW WRITE ACCESS was specified, an IS table lock is acquired.
  - If ALLOW READ ACCESS was specified, an S table lock is acquired.
- An S lock on the table is requested during the truncation phase. Until the S lock is acquired, rows can be inserted by concurrent transactions. These inserted rows might not be seen by the reorg utility, and could prevent the table from being truncated. After the S table lock is acquired, rows that prevent the table from being truncated are moved to compact the table. After the table is compacted, it is truncated, but only after all transactions that are accessing the table at the time the truncation point is determined have completed.
- A row lock might be acquired, depending on the type of table lock:
  - If an S lock is held on the table, there is no need for individual row-level S locks, and further locking is unnecessary.
  - If an IS lock is held on the table, an NS row lock is acquired before the row is moved, and then released after the move is complete.
- Certain internal locks might also be acquired during an online table reorg operation.

Locking has an impact on the performance of both online table reorg operations and concurrent user applications. You can use lock snapshot data to help you to understand the locking activity that occurs during online table reorganizations.

Monitoring a table reorganization:

You can use the **GET SNAPSHOT** command, the SNAPTAB_REORG administrative view, or the SNAP_GET_TAB_REORG table function to obtain information about the status of your table reorganization operations.

Procedure

- To access information about reorganization operations using SQL, use the SNAPTAB_REORG administrative view. For example, the following query returns details about table reorganization operations on all database partitions for the currently connected database. If no tables have been reorganized, no rows are returned.

```
select
    substr(tabname, 1, 15) as tab_name,
    substr(tabschema, 1, 15) as tab_schema,
    reorg_phase,
    substr(reorg_type, 1, 20) as reorg_type,
    reorg_status,
    reorg_completion,
    dbpartitionnum
  from sysibmadm.snaptab_reorg
  order by dbpartitionnum
```

- To access information about reorganization operations using the snapshot monitor, use the **GET SNAPSHOT FOR TABLES** command and examine the values of the table reorganization monitor elements.

Results

Because offline table reorg operations are synchronous, errors are returned to the caller of the utility (an application or the command line processor). And because online table reorg operations are asynchronous, error messages in this case are not returned to the CLP. To view SQL error messages that are returned during an online table reorg operation, use the **LIST HISTORY REORG** command.

An online table reorg operation runs in the background as the db2Reorg process. This process continues running even if the calling application terminates its database connection.

Index reorganization

As tables are updated, index performance can degrade.

The degradation can occur in the following ways:

- Leaf pages become fragmented. When leaf pages are fragmented, I/O costs increase because more leaf pages must be read to fetch table pages.
- The physical index page order no longer matches the sequence of keys on those pages, resulting in low density indexes. When leaf pages have a low density, sequential prefetching is inefficient and the number of I/O waits increases. However, if smart index prefetching is enabled, the query optimizer switches to readahead prefetching if low density indexes exist. This action helps reduce the negative impact that low density indexes have on performance.
- The index develops too many levels. In this case, the index might be reorganized.

Index reorganization requires:

- SYSADM, SYSMAINT, SYSCTRL, DBADM, or SQLADM authority, or CONTROL privilege on the table and its indexes
- When the REBUILD option with the ALLOW READ or WRITE ACCESS options are chosen, an amount of free space in the table space where the indexes are stored is required. This space must be equal to the current size of the indexes. Consider placing indexes in a large table space when you issue the **CREATE TABLE** statement.
- Additional log space. The index REORG utility logs its activities.

If you specify the MINPCTUSED option on the CREATE INDEX statement, the database server automatically merges index leaf pages if a key is deleted and the free space becomes less than the specified value. This process is called *online index defragmentation*.

To restore index clustering, free up space, and reduce leaf levels, you can use one of the following methods:

- Drop and re-create the index.
- Use the **REORG TABLE** command with options that reorganize the table and rebuild its indexes offline.
- Use the **REORG INDEXES** command with the REBUILD option to reorganize indexes online or offline. You might choose online reorganization in a production environment. Online reorganization allows users to read from or write to the table while its indexes are being rebuilt.

If your primary objective is to free up space, consider the CLEANUP and RECLAIM EXTENTS options of the **REORG** command. See the related links for more details.

In IBM Data Studio Version 3.1 or later, you can use the task assistant for reorganizing indexes. Task assistants can guide you through the process of setting options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

With Db2 V9.7 Fix Pack 1 and later releases, if you specify a partition with the ON DATA PARTITION clause, the **REORG INDEXES ALL** command that is run on a data partitioned table reorganizes the partitioned indexes for the single data partition. During index reorganization, the unaffected partitions remain read and write accessible access is restricted only to the affected partition.

REORG TABLE commands and **REORG INDEXES ALL** commands can be issued on a data partitioned table to concurrently reorganize different data partitions or partitioned indexes on a partition. When concurrently reorganizing data partitions or the partitioned indexes on a partition, users can access the unaffected partitions. All the following criteria must be met to issue REORG commands that operate concurrently on the same table:

- Each REORG command must specify a different partition with the **ON DATA PARTITION** clause.
- Each REORG command must use the ALLOW NO ACCESS mode to restrict access to the data partitions.
- The partitioned table must have only partitioned indexes if you run **REORG TABLE** commands. No nonpartitioned indexes (except system-generated XML path indexes) can be defined on the table.

Note: The output from the **REORGCHK** command contains statistics and recommendations for reorganizing indexes. For a partitioned table, the output contains statistics and recommendations for reorganizing partitioned and nonpartitioned indexes. Alternatively, if the objective is to reclaim space, the **RECLAIMABLE_SPACE** output of the **ADMIN_GET_INDEX_INFO** function shows how much space is reclaimable. Use the **REORG INDEXES** command with the RECLAIM EXTENTS option to free this reclaimable space.

Online index reorganization

When you use the **REORG INDEXES** command with the ALLOW READ/WRITE ACCESS and REBUILD options, a shadow copy of the index object is built while the original index object remains available as read or write access to the table continues. If write access is allowed, then during reorganization, any changes to the underlying table that would affect the indexes are logged. The reorg operation processes these logged changes while rebuilding the indexes.

Changes to the underlying table that would affect the indexes are also written to an internal memory buffer, if such space is available for use. The internal buffer is a designated memory area that is allocated on demand from the utility heap. The use of a memory buffer enables the index reorg utility to process the changes by reading directly from memory first, and then reading through the logs, if necessary, but at a much later time. The allocated memory is freed after the reorg operation completes.

Extra storage space is required in the index tablespace to hold the shadow copy of the index. Once the shadow copy of the index is built and all logs affecting the shadow copy have been processed, then a super-exclusive lock is taken on the table and the original index is discarded. The space that was occupied by the original copy of the index object is free to be reused by any object in the same tablespace, however it is not automatically returned to the filesystem.

Online index reorganization in ALLOW WRITE ACCESS mode, with or without the CLEANUP option, is supported for spatial indexes or MDC and ITC tables.

Locking and concurrency considerations for online index reorganization:

In this topic, online index reorganization applies to an index reorganization that is run with the **ALLOW READ ACCESS** or **ALLOW WRITE ACCESS** parameters.

These options allow you to access the table while its indexes are being reorganized. During online index reorganization with the **REBUILD** option, new indexes are built as additional copies while the original indexes remain intact. Concurrent transactions use the original indexes while the new indexes are created. At the end of the reorganization operation, the original indexes are replaced by the new indexes. Transactions that are committed in the meantime are reflected in the new indexes after the replacement of the original indexes. If the reorganization operation fails and the transaction is rolled back, the original indexes remain intact. During online index reorganization with the **CLEANUP** and **RECLAIM EXTENTS** options, space is reclaimed and made available for use for all objects in the table space.

An online index reorganization operation can hold the following locks:

- To ensure access to table spaces, an IX-lock is acquired on the table spaces affected by the reorganization operation. This includes table spaces that hold the table, as well as partition, and index objects.

- To prevent the affected table from being altered during reorganization, an X alter table lock is acquired.
- A table lock is acquired and held throughout the reorganization operation. The type of lock depends on the table type, access mode, and reorganization option:
  - For nonpartitioned tables:
    - If **ALLOW READ ACCESS** is specified, a U-lock is acquired on the table.
    - If **ALLOW WRITE ACCESS** is specified, an IN-lock is acquired on the table.
    - If **CLEANUP** is specified, an S-lock is acquired on the table for READ access, and IX-lock for WRITE access.
  - For partitioned tables, reorganization with **ALLOW READ ACCESS** or **ALLOW WRITE ACCESS** is supported at partition level only:
    - If **ALLOW READ ACCESS** is specified, a U-lock is acquired on the partition.
    - If **ALLOW WRITE ACCESS** is specified, an IS-lock is acquired on the partition.
    - If **CLEANUP** is specified, an S-lock is acquired on the partition for READ access, and an IX-lock for WRITE access.
    - An IS-lock is acquired on the table regardless of which access mode or option is specified.
- An exclusive Z-lock on the table or partition is requested at the end of index reorganization. If a partitioned table contains nonpartitioned indexes, then the Z-lock is acquired on the table as well as the partition. This lock suspends table and partition access to allow for the replacement of the original indexes by the new indexes. This lock is held until transactions that are committed during reorganization are reflected in the new indexes.
- The IS table lock and NS row lock are acquired on the system catalog table SYSIBM.SYSTABLES.
- For a partition level reorganization, IS table lock and NS row lock are also acquired on the system catalog table SYSIBM.SYSDATAPARTITIONS.
- Certain internal locks might also be acquired during an online index reorganization operation.
- Online index reorganization might have impact on concurrency if the reorganization operation fails. For example, the reorganization might fail due to insufficient memory, lack of disk space, or a lock timeout. The reorganization transaction performs certain updates before ending. To perform updates, reorganization must wait on existing transaction to be committed. This delay might block other transactions in the process. Starting in Db2 Version 9.7 Fix Pack 1, reorganization requests a special drain lock on the index object. Reorganization operations wait for existing transactions to finish; however, new requests to access the index object are allowed.

Monitoring an index reorganization operation:

You can use the **db2pd** command to monitor the progress of reorganization operations on a database.

Procedure

Issue the **db2pd** command with the **-reorgs index** parameter:

```
db2pd -reorgs index
```

Results

The following is an example of output obtained using the **db2pd** command with the**-reorgs index** parameter, which reports the index reorganization progress for a range-partitioned table with two partitions.

Note: The first output reports the Index Reorg Stats of the non-partitioned indexes. The following outputs report the Index Reorg Stats of the partitioned indexes on each partition. The index reorganization statistics of only one partition is reported in each output.

```
Index Reorg Stats:
Retrieval Time: 02/08/2010 23:04:21
TbspaceID: -6       TableID: -32768
Schema: TEST1    TableName: BIGRPT
Access: Allow none
Status: Completed
Start Time: 02/08/2010 23:03:55   End Time: 02/08/2010 23:04:04
Total Duration: 00:00:08
Prev Index Duration:  -
Cur Index Start: -
Cur Index: 0            Max Index: 2            Index ID: 0
Cur Phase: 0          ( -     )   Max Phase: 0
Cur Count: 0                      Max Count: 0
Total Row Count: 750000

Retrieval Time: 02/08/2010 23:04:21
TbspaceID: 2        TableID: 5
Schema: TEST1    TableName: BIGRPT
PartitionID: 0      MaxPartition: 2
Access: Allow none
Status: Completed
Start Time: 02/08/2010 23:04:04   End Time: 02/08/2010 23:04:08
Total Duration: 00:00:04
Prev Index Duration:  -
Cur Index Start: -
Cur Index: 0            Max Index: 2            Index ID: 0
Cur Phase: 0          ( -     )   Max Phase: 0
Cur Count: 0                      Max Count: 0
Total Row Count: 375000

Retrieval Time: 02/08/2010 23:04:21
TbspaceID: 2        TableID: 6
Schema: TEST1    TableName: BIGRPT
PartitionID: 1      MaxPartition: 2
Access: Allow none
Status: Completed
Start Time: 02/08/2010 23:04:08   End Time: 02/08/2010 23:04:12
Total Duration: 00:00:04
Prev Index Duration:  -
Cur Index Start: -
Cur Index: 0            Max Index: 2            Index ID: 0
Cur Phase: 0          ( -     )   Max Phase: 0
Cur Count: 0                      Max Count: 0
Total Row Count: 375000
```

Determining when to reorganize tables and indexes

After many changes to table data, logically sequential data might be on nonsequential physical data pages, especially if many update operations created overflow records. When the data is organized in this way, the database manager must perform additional read operations to access required data. Additional read operations are also required if many rows are deleted.

About this task

Table reorganization defragments data, eliminating wasted space. It also reorders the rows to incorporate overflow records, improving data access and, ultimately, query performance. You can specify that the data can be reordered according to a particular index, so that queries can access the data with a minimal number of read operations.

Many changes to table data can cause index performance to degrade. Index leaf pages can become fragmented or badly clustered, and the index could develop more levels than necessary for optimal performance. All of these issues cause more I/Os and can degrade performance.

Any one of the following factors indicate that you might reorganize a table or index:

- A high volume of insert, update, and delete activity against a table since the table was last reorganized
- Significant changes in the performance of queries that use an index with a high cluster ratio
- Executing the **RUNSTATS** command to refresh statistical information does not improve performance
- Output from the **REORGCHK** command suggests that performance can be improved by reorganizing a table or its indexes

  In some cases, the reorgchk utility might recommend table reorganization, even after a table reorg operation is performed. You should analyze reorgchk utility recommendations and assess the potential benefits against the costs of performing a reorganization.
- If reclaiming space is your primary concern, the **REORG** command with the `CLEANUP` and `RECLAIM EXTENTS` options can be used.

  The **RECLAIMABLE_SPACE** output of the **ADMIN_GET_INDEX_INFO** and **ADMIN_GET_TAB_INFO** functions show how much space, in kilobytes, is available for reclaim. If you issue the **REORG** command with the `CLEANUP` option before running the **ADMIN_GET_INDEX_INFO** and **ADMIN_GET_TAB_INFO** functions, the output of the functions shows the maximum space available for reclamation. Use this information to determine when a **REORG** with `RECLAIM EXTENTS` would help reduce the size of your tables and indexes.

The **REORGCHK** command returns statistical information about data organization and can advise you about whether particular tables or indexes need to be reorganized. When space reclaim is your only concern, the **RECLAIMABLE_SPACE** output of the **ADMIN_GET_INDEX_INFO** and **ADMIN_GET_TAB_INFO** functions outline how much space is available for reclaim. However, running specific queries against the SYSSTAT views at regular intervals or at specific times can build a history that helps you identify trends that have potentially significant performance implications.

To determine whether there is a need to reorganize your tables or indexes, query the SYSSTAT views and monitor the following statistics:

Overflow of rows

The overflow value represents the number of rows that do not fit on their original pages. To monitor the overflow value, query the OVERFLOW column in the SYSSTAT.TABLES view. Row data can overflow when variable length columns cause the record length to expand to the point that a row no longer fits into its assigned location on the data page. Length

changes can also occur if you add a column to the table. In this case, a pointer is kept in the original location in the row, and the value is stored in another location that is indicated by the pointer. This can impact performance because the database manager must follow the pointer to find the contents of the column. This two-step process increases the processing time and might also increase the number of I/Os that are required. Reorganizing the table data eliminates any row overflows.

An overflow can also occur if a row is compressed, then updated. This overflow occurs when the updated row can no longer be compressed or if the updated compressed row is longer.

Fetch statistics

To determine the effectiveness of the prefetchers when the table is accessed in index order, query the following columns in the SYSSTAT.INDEXES catalog view. The statistics in these columns characterize the average performance of the prefetchers against the underlying table.

- The AVERAGE_SEQUENCE_FETCH_PAGES column stores the average number of pages that can be accessed in sequence. Pages that can be accessed in sequence are eligible for prefetching. A small number indicates that the prefetchers are not as effective as they could be, because they cannot read in the full number of pages that is specified by the PREFETCHSIZE value for the table space. A large number indicates that the prefetchers are performing effectively. For a clustered index and table, the number should approach the value of NPAGES, the number of pages that contain rows.
- The AVERAGE_RANDOM_FETCH_PAGES column stores the average number of random table pages that are fetched between sequential page accesses when table rows are fetched by using an index. The prefetchers ignore small numbers of random pages when most pages are in sequence and continue to prefetch to the configured prefetch size. As the table becomes more disorganized, the number of random fetch pages increases. Disorganization is usually caused by insertions that occur out of sequence, either at the end of the table or in overflow pages, and query performance is impacted when an index is used to access a range of values.
- The AVERAGE_SEQUENCE_FETCH_GAP column stores the average gap between table page sequences when table rows are fetched by using an index. Detected through a scan of index leaf pages, each gap represents the average number of table pages that must be randomly fetched between sequences of table pages. This occurs when many pages are accessed randomly, which interrupts the prefetchers. A large number indicates that the table is disorganized or poorly clustered to the index.

Number of index leaf pages that contain record identifiers (RIDs) that are marked deleted but not yet removed

RIDs are not usually physically deleted when they are marked deleted. This means that useful space might be occupied by these logically deleted RIDs. To retrieve the number of leaf pages on which every RID is marked deleted, query the NUM_EMPTY_LEAFS column of the SYSSTAT.INDEXES view. For leaf pages on which not all RIDs are marked deleted, the total number of logically deleted RIDs is stored in the NUMRIDS_DELETED column.

Use this information to estimate how much space you might reclaim by issuing the **REORG INDEXES** command with the **CLEANUP ALL** option. To

reclaim only the space in pages on which all RIDs are marked deleted, issue the **REORG INDEXES** command with the **CLEANUP PAGES** option.

Cluster-ratio and cluster-factor statistics for indexes

In general, only one of the indexes for a table can have a high degree of clustering. A cluster-ratio statistic is stored in the CLUSTERRATIO column of the SYSCAT.INDEXES catalog view. This value, which is 0 - 100, represents the degree of data clustering in the index. If you collect detailed index statistics, a finer cluster-factor statistic of 0 - 1 is stored in the CLUSTERFACTOR column instead, and the value of the CLUSTERRATIO column is -1. Only one of these two clustering statistics can be recorded in the SYSCAT.INDEXES catalog view. To compare CLUSTERFACTOR values with CLUSTERRATIO values, multiply the CLUSTERFACTOR value by 100 to obtain a percentage value.

Index scans that are not index-only access might perform better with a higher density of indexes. A low density leads to more I/O for this type of scan because a data page is less likely to remain in the buffer pool until it is accessed again. Increasing the buffer size might improve the performance of low density indexes. If smart index prefetching is enabled, it can also improve the performance of low density indexes, which reduces the need to issue the **REORG** command on indexes. Smart index prefetching achieves this by switching from sequential detection prefetching to read-ahead prefetching whenever low density indexes exist.

If table data was initially clustered on a certain index and the clustering statistics indicate that the data is now poorly clustered on that index, you might want to reorganize the table to recluster the data. Also, if smart data prefetching is enabled, it can improve the performance of poorly clustered data, which reduces the need to issue the **REORG** command on tables. Smart data prefetching achieves this by switching from sequential detection prefetching to read-ahead prefetching whenever badly clustered data pages exist.

Number of leaf pages

To determine the number of leaf pages that are occupied by an index, query the NLEAF column in the SYSCAT.INDEXES view. This number tells you how many index page I/Os are needed for a complete scan of the index.

Ideally, an index should occupy as little space as possible to reduce the number of I/Os that are required for an index scan. Random update activity can cause page splits that increase the size of an index. During a table **REORG** operation, each index can be rebuilt with the least amount of space.

By default, 10% of free space is left on each index page when an index is built. To increase the free space amount, specify the PCTFREE option when you create the index. The specified PCTFREE value is used whenever you reorganize the index. A free space value that is greater than 10% might reduce the frequency of index reorganization, because the extra space can accommodate additional index insertions.

Number of empty data pages

To calculate the number of empty pages in a table, query the FPAGES and NPAGES columns in the SYSCAT.TABLES view and then subtract the NPAGES value (the number of pages that contain rows) from the FPAGES

value (the total number of pages that are in use). Empty pages can occur when you delete entire ranges of rows.

As the number of empty pages increases, so does the need for table reorganization. Reorganizing a table reclaims empty pages and reduces the amount of space that a table uses. In addition, because empty pages are read into the buffer pool during a table scan, reclaiming unused pages can improve scan performance.

Table reorganization is not recommendedif the following equation evaluates to true: the total number of in-use pages (specified in the FPAGES column) in a table <= (NPARTITIONS x 1 extent size). NPARTITIONS represents the number of data partitions if the table is a partitioned table; otherwise, its value is 1.

In a partitioned database environment, table reorganization is not recommended if the following equation evaluates to true: value of the FPAGES column <= (the number of database partitions in a database partition group of the table) x (NPARTITIONS x 1 extent size).

Before reorganizing tables or indexes, consider the trade-off between the cost of increasingly degraded query performance and the cost of table or index reorganization, which includes processing time, elapsed time, and reduced concurrency.

Costs of table and index reorganization

Performing a table reorganization or an index reorganization with the **REBUILD** option incurs a certain amount of overhead that must be considered when deciding whether to reorganize an object.

The costs of reorganizing tables and reorganizing indexes with the **REBUILD** option include:

- Processing time of the executing utility
- Reduced concurrency (because of locking) while running the reorg utility.
- Extra storage requirements.
  - Offline table reorganization requires more storage space to hold a shadow copy of the table.
  - Online or inplace table reorganization requires more log space.
  - Offline index reorganization requires less log space and does not involve a shadow copy.
  - Online index reorganization requires more log space and more storage space to hold a shadow copy of the index.

In some cases, a reorganized table might be larger than the original table. A table might grow after reorganization in the following situations:

- In a clustering reorg table operation in which an index is used to determine the order of the rows, more space might be required if the table records are of a variable length, because some pages in the reorganized table might contain fewer rows than in the original table.
- The amount of free space left on each page (represented by the PCTFREE value) might have increased since the last reorganization.

Space requirements for an offline table reorganization

Because offline reorganization uses a shadow copy approach, you need enough additional storage to accommodate another copy of the table. The shadow copy is built either in the table space in which the original table resides or in a user-specified temporary table space.

Additional temporary table space storage might be required for sort processing if a table scan sort is used. The additional space required might be as large as the size of the table being reorganized. If the clustering index is of system managed space (SMS) type or unique database managed space (DMS) type, the recreation of this index does not require a sort. Instead, this index is rebuilt by scanning the newly reorganized data. Any other indexes that are recreated will require a sort, potentially involving temporary space up to the size of the table being reorganized.

Offline table reorg operations generate few control log records, and therefore consume a relatively small amount of log space. If the reorg utility does not use an index, only table data log records are created. If an index is specified, or if there is a clustering index on the table, record IDs (RIDs) are logged in the order in which they are placed into the new version of the table. Each RID log record holds a maximum of 8000 RIDs, with each RID consuming 4 bytes. This can contribute to log space problems during an offline table reorg operation. Note that RIDs are only logged if the database is recoverable.

Log space requirements for an online table reorganization

The log space that is required for an online table reorg operation is typically larger than what is required for an offline table reorg. The amount of space that is required is determined by the number of rows being reorganized, the number of indexes, the size of the index keys, and how poorly organized the table is at the outset. It is a good idea to establish a typical benchmark for log space consumption associated with your tables.

Every row in a table is likely moved twice during an online table reorg operation. For each index, each table row must update the index key to reflect the new location, and after all accesses to the old location have completed, the index key is updated again to remove references to the old RID. When the row is moved back, updates to the index key are performed again. All of this activity is logged to make online table reorganization fully recoverable. There is a minimum of two data log records (each including the row data) and four index log records (each including the key data) for each row (assuming one index). Clustering indexes, in particular, are prone to filling up the index pages, causing index splits and merges which must also be logged.

Because the online table reorg utility issues frequent internal COMMIT statements, it usually does not hold a large number of active logs. An exception can occur during the truncation phase, when the utility requests an S table lock. If the utility cannot acquire the lock, it waits, and other transactions might quickly fill up the logs in the meantime.

Reduce the need for table and index reorganization

You can use different strategies to reduce the need for, and the costs that are associated with, table and index reorganization.

Reduce the need for table reorganization

You can reduce the need for table reorganization with the following tables and techniques:

- Use multi-partition tables.
- Create multidimensional clustering (MDC) tables. For MDC tables, clustering is maintained on the columns that you specify with the ORGANIZE BY DIMENSIONS clause of the CREATE TABLE statement. However, the REORGCHK utility might still suggest reorganization of an MDC table if it determines that there are too many unused blocks or that blocks can be compacted.
- In cases where space reuse of empty areas of a table is a motivating factor for the reorganization, create insert time clustering (ITC) tables. For ITC tables, if you have a cyclical access pattern you can release that space back to the system. An example of a cyclical access pattern is deleting all data that was inserted at similar times. In such cases, you can reduce the need for a table reorganization with the RECLAIM EXTENTS table clause of the REORG command. The REORG command with the RECLAIM EXTENTS table clause reclaims free space on which the table is found. The command also consolidates sparse extents implicitly. This consolidation leads to more space reclamation, but a longer duration for utility execution when compared to Db2 Version 10.1.
- Enable the APPEND mode on your tables. If the index key values for new rows are always new high key values. For example, the clustering attribute of the table attempts to place them at the end of the table. In this case, enabling the APPEND mode might be a better choice than using a clustering index.

To further reduce the need for table reorganization, follow these tasks after you create a table:

- Alter the table to specify the percentage of each page that is to be left as free space during a load or a table reorganization operation (PCTFREE).
- Create a clustering index, specifying the PCTFREE option.
- Sort the data before it is loaded into the table.

After you follow these tasks, the clustering index and the PCTFREE setting on the table help preserve the original sorted order. If there is enough space on the table pages, new data can be inserted on the correct pages to maintain the clustering characteristics of the index. However, as more data is inserted and the table pages become full, records are appended to the end of the table, which gradually becomes unclustered.

If you run a table REORG operation or a sort and load operation after you create a clustering index, the index attempts to maintain the order of the data. This action improves the CLUSTERRATIO or CLUSTERFACTOR statistics that are collected by the RUNSTATS utility.

Note: If readahead prefetching is enabled, the table might not require reorganization for clustering purposes even if formula F4 of the **REORGCHK** command states otherwise.

Reduce index rebuild requirements

You can reduce index rebuild requirements with index reorganization:

- Create indexes that specify the PCTFREE or the LEVEL2 PCTFREE option.

- Create indexes with the MINPCTUSED option. Alternatively, consider running the REORG INDEXES command with the CLEANUP ALL option to merge leaf pages.
- Use the RECLAIM EXTENTS option of the REORG INDEXES command to release space back to the table space in an online fashion. This operation provides space reclaim without the need for a full rebuild of the indexes.

Note: If readahead prefetching is enabled it helps reduce index rebuilds with index reorganization, even if formula F4 of the **REORGCHK** command states otherwise.

Space reclamation for column-organized tables

When data is deleted from a column-organized table, the storage extents whose pages held data that was all deleted are candidates for space reclamation. The space reclamation process finds these extents, and returns the pages that they hold to table space storage, where they can later be reused by any table in the table space.

You can start this process manually by specifying the `RECLAIM EXTENTS` option for the **REORG TABLE** command, or you can use an automated approach. If you set the **DB2_WORKLOAD** registry variable to `ANALYTICS`, a default policy is applied, and the **auto_reorg** database configuration parameter is set so that automatic reclamation is active for all column-organized tables. For more information, see "Enabling automatic table and index reorganization" and "Configuring an automated maintenance policy".

You can monitor the progress of a table reorganization operation with the reorganization monitoring infrastructure. The ADMIN_GET_TAB_INFO table function returns an estimate of the amount of reclaimable space on the table, which you can use to determine when a table reorganization operation is necessary.

Note: Because RECLAIMABLE_SPACE output from the ADMIN_GET_TAB_INFO table function is an estimate that applies to the time at which the **RUNSTATS** command was run, this information might be outdated for column-organized tables.

Space reclamation after deletions

When a row in a column-organized table is deleted, the row is logically deleted, not physically deleted. As a result, the space that is used by the deleted row is not available to subsequent transactions and remains unusable until space reclamation occurs. For example, consider the case where a table is created and one million rows are inserted in batch operation A. The size of the table on disk after batch operation A is 5 MB. After some time, batch operation B inserts another 1 million rows. Now the table uses 10 MB on disk. Then, all of the rows that were inserted in batch operation A are deleted, and the table size on disk remains 10 MB. If a third batch operation C inserts another 1 million rows into the table, 5 MB of extra space is required. (With row-organized tables, the rows that are inserted in batch operation C would use the space that was vacated by the deleted rows from batch operation A.) A **REORG TABLE** command is required to reclaim the space that was used by the rows that were inserted in batch operation A.

Space reclamation after updates

When a row in a column-organized table is updated, the row is first deleted, and a new copy of the row is inserted at the end of the table. This means that an

updated row uses space in proportion to the number of times that the row is updated until space reclamation occurs. All rows in the extent where the update took place must be deleted before any space reclamation can occur.

Automatic table and index maintenance

After many changes to table data, a table and its indexes can become fragmented. Logically sequential data might be found on nonsequential pages, forcing additional read operations by the database manager to access data.

The statistical information that is collected by the **RUNSTATS** utility shows the distribution of data within a table. Analysis of these statistics can indicate when and what type of reorganization is necessary.

The automatic reorganization process determines the need for table or index reorganization by using formulas that are part of the **REORGCHK** utility. It periodically evaluates tables and indexes that had their statistics updated to see whether reorganization is required, and schedules such operations whenever they are necessary.

The automatic reorganization feature can be enabled or disabled through the **auto_reorg**, **auto_tbl_maint**, and **auto_maint** database configuration parameters.

In a partitioned database environment, the initiation of automatic reorganization is done on the catalog database partition. These configuration parameters are enabled only on the catalog database partition. The **REORG** operation, however, runs on all of the database partitions on which the target tables are found.

If you are unsure about when and how to reorganize your tables and indexes, you can incorporate automatic reorganization as part of your overall database maintenance plan.

You can also reorganize multidimensional clustering (MDC) and insert time clustering (ITC) tables to reclaim space. The freeing of extents from MDC and ITC tables is only supported for tables in DMS table spaces and automatic storage. Freeing extents from your MDC and ITC tables can be done in an online fashion with the RECLAIM EXTENTS option of the REORG TABLE command.

You can also schedule an alternate means to reclaim space from your indexes. The REORG INDEX command has an index clause in which you can specify space-reclaim-options. When you specify RECLAIM EXTENTS in space-reclaim-options, space is released back to the table space in an online fashion. This operation provides space reclamation without the need for a full rebuild of the indexes. The REBUILD option of the REORG INDEX command also reclaims space, but not necessarily in an online fashion.

Automatic reorganization on data partitioned tables

For Db2 Version 9.7 Fix Pack 1 and earlier releases, automatic reorganization supports reorganization of a data partitioned table for the entire table. For Db2 V9.7 Fix Pack 1 and later releases, automatic reorganization supports reorganizing data partitions of a partitioned table and reorganizing the partitioned indexes on a data partition of a partitioned table.

To avoid placing an entire data partitioned table into ALLOW NO ACCESS mode, automatic reorganization performs **REORG INDEXES ALL** operations at the data

partition level on partitioned indexes that need to be reorganized. Automatic reorganization performs **REORG INDEX** operations on any nonpartitioned index that needs to be reorganized.

Automatic reorganization performs the following **REORG TABLE** operations on data partitioned tables:

- If any nonpartitioned indexes (except system-generated XML path indexes) are defined on the table and there is only one partition that needs to be reorganized, automatic reorganization performs a **REORG TABLE** operation with the **ON DATA PARTITION** clause to specify the partition that needs to be reorganized. Otherwise, automatic reorganization performs a **REORG TABLE** on the entire table without the **ON DATA PARTITION** clause.
- If no nonpartitioned indexes (except system-generated XML path indexes) are defined on the table, automatic reorganization performs a **REORG TABLE** operation with the **ON DATA PARTITION** clause on each partition that needs to be reorganized.

Automatic reorganization on volatile tables

You can enable automatic index reorganization for volatile tables. The automatic reorganization process determines whether index reorganization is required for volatile tables and schedules a REORG INDEX CLEANUP. Index reorganization is performed periodically on volatile tables and releases space that can be reused by the indexes defined on these tables.

Statistics cannot be collected in volatile tables because they are updated frequently. To determine what indexes need to be reorganized, automatic reorganization uses the numInxPseudoEmptyPagesForVolatile attribute instead of REORGCHK. The number of pseudo empty pages is maintained internally, visible through mon_get_index, and does not require a RUNSTATS operation like REORGCHK. This attribute in the AUTO_REORG policy indicates how many empty index pages with pseudo deleted keys an index must have so index reorganization is triggered.

To enable automatic index reorganization in volatile tables:

- The **DB2_WORKLOAD** registry variable must be set to SAP.
- Automatic reorganization must be enabled.
- The numInxPseudoEmptyPagesForVolatile attribute must be set.

Enabling automatic table and index reorganization:

Use automatic table and index reorganization to eliminate the worry of when and how to reorganize your data.

About this task

Having well-organized table and index data is critical to efficient data access and optimal workload performance. After many database operations, such as insert, update, and delete, logically sequential table data might be found on nonsequential data pages. When logically sequential table data is found on nonsequential data pages, additional read operations are required by the database manager to access data. Additional read operations are also required when accessing data in a table from which a significant number of rows are deleted. You can enable the database manager to reorganize system both catalog tables and user tables.

Procedure

To enable your database for automatic reorganization:

1. Set the auto_maint, auto_tbl_maint, and auto_reorg database configuration parameters to ON. You can set the parameters to ON with these commands:
   - **db2 update db cfg for <db_name> using auto_maint on**
   - **db2 update db cfg for <db_name> using auto_tbl_maint on**
   - **db2 update db cfg for <db_name> using auto_reorg on**

     Replace <db_name> with the name of the database on which you want to enable automatic maintenance and reorganization.
2. Connect to the database, <db_name>.
3. Specify a reorganization policy. A reorganization policy is a defined set of rules or guidelines that dictate when automated table and index maintenance takes place. You can set this policy in one of two ways:
   a. Call the AUTOMAINT_SET_POLICY procedure.
   b. Call the AUTOMAINT_SET_POLICYFILE procedure.

   The reorganization policy is either an input argument or file both of which are in an XML format. For more information about both of these procedures, see the Related reference.

Example

For an example of this capability, see the Related concepts.

Enabling automatic index reorganization in volatile tables:

You can enable automatic reorganization to perform index reorganization in volatile tables.

About this task

If you enable automatic index reorganization in volatile tables, automatic reorg checks at every refresh interval whether the indexes on volatile tables require reorganization and schedules the necessary operation using the **REORG** command.

Procedure

To enable automatic index reorganization in volatile tables, perform the following steps:

1. Set the **DB2_WORKLOAD** registry variable to SAP. The following example shows how to set this variable using the **db2set** command:

   ```
   db2set DB2_WORKLOAD=SAP
   ```

   Restart the database so that this setting takes effect.
2. Set the **auto_reorg** database configuration parameter to ON. The following example shows how to set this database configuration parameter using the Db2 CLP command line interface:

   ```
   UPDATE DB CFG FOR SAMPLE USING auto_reorg ON
   ```

   Ensure that the **auto_maint** and **auto_tbl_maint** database configuration parameters are also set to ON. By the default, **auto_maint** and **auto_tbl_maint** are set to ON.

3. Set the numInxPseudoEmptyPagesForVolatileTables attribute in the AUTO_REORG policy by calling the AUTOMAINT_SET_POLICY or AUTOMAINT_SET_POLICYFILE procedure. This attribute indicates the minimum number of empty index pages with pseudo deleted keys required to perform the index reorganization. The following example shows how to set this attribute:

```
CALL SYSPROC.AUTOMAINT_SET_POLICY
  ('AUTO_REORG',
  BLOB(' <?xml version="1.0" encoding="UTF-8"?>
<DB2AutoReorgPolicy
xmlns="http://www.ibm.com/xmlns/prod/db2/autonomic/config" >

 <ReorgOptions  dictionaryOption="Keep" indexReorgMode="Online"
    useSystemTempTableSpace="false"
 numInxPseudoEmptyPagesForVolatileTables="20" />

 <ReorgTableScope  maxOfflineReorgTableSize="0">
  <FilterClause>TABSCHEMA NOT LIKE 'SYS%'</FilterClause>
 </ReorgTableScope>
</DB2AutoReorgPolicy>')
  )
```

   You can monitor the values for the PSEUDO_EMPTY_PAGES, EMPTY_PAGES_DELETED, and EMPTY_PAGES_REUSED column by querying the MON_GET_INDEX table function to help you determine an appropriate value for the numInxPseudoEmptyPagesForVolatileTables attribute.

Scenario: ExampleBANK reclaiming table and index space

This scenario presents ExampleBANK, a banking institution with a large customer base spanning many branches, as an organization wanting to reclaim table and index space.

ExampleBANK reclaims table and index space to ensure that the size of their tables and indexes is consistently managed without rendering them unavailable to users at any time. Organizations that handle large amounts of constantly changing data, like ExampleBANK, need a way to manage their table and index size. Reclaiming table and index space helps achieve this objective.

Scenario: ExampleBANK reclaiming table and index space - Space management policies:

The database administrator at ExampleBANK, Olivia, struggled for years to effectively manage the size of databases.

During the normal course of business operation, batch deletes are done on tables to get rid of data that is no longer required. The tables and associated indexes have free unused space that cannot be used by any other object in the same table space after the batch deletes are complete. ExampleBank has a space management policy in place to free this unused space. Each month Olivia takes the affected databases offline so the tables and indexes can be reorganized. The reorganization of the objects frees the space. To minimize downtime, the work is done during off-peak hours.

This table and index space management policy takes time and manual intervention. Also, because Olivia takes the database offline to complete this task, the affected tables and indexes are not available to users during the reorganization.

Olivia is told about new command and statement parameters to reclaim space from tables and indexes. A new way to manage the space needed for tables and indexes is presented.

Scenario: ExampleBANK reclaiming table and index space - Creating an insert time clustering table:

Insert time clustering (ITC) tables can help Olivia, and ExampleBANK, manage database size more effectively without manual intervention or database downtime.

Olivia creates an insert time clustering table as a test. The ORGANIZE BY INSERT TIME clause ensures that the table is created as an ITC table:

```
DB2 CREATE TABLE T1(c1 int, c2 char(100), ...) IN TABLESPACE1
      ORGANIZE BY INSERT TIME;
DB2 CREATE INDEX INX1 ON T1(C1);
```

Scenario: ExampleBANK reclaiming table and index space - Evaluating the effectiveness of reclaiming space from a table:

Time passes and normal operations are run on the ITC table.

At some point a batch delete is run on this table, and large portions of the object become empty. Olivia wants to make this trapped space available to other objects in TABLESPACE1. Olivia can evaluate the effectiveness of space reclaim by using the ADMIN_GET_TAB_INFO function. This function can collect information about the physical space that is occupied by the data portion of the table T1.

Olivia collects the number of extents in the table T1 that are candidates for cleanup:

```
SELECT T.SPARSE_BLOCKS, T.RECLAIMABLE_SPACE FROM TABLE
       (SYSPROC.ADMIN_GET_TAB_INFO('OLIVIA,','T1')) T

SPARSE_BLOCKS        RECLAIMABLE_SPACE
-------------------- -----------------

7834                 14826781647

  1 record(s) selected.
```

Olivia notices that there are a significant number of blocks in this table that are sparsely populated with data. A significant amount of space is available to be reclaimed. By running a reorganization on this table, Olivia consolidates the remaining data in these blocks into a smaller group of blocks. A reorganization also releases any fully empty blocks back to the table space. Using the **REORG** command, Olivia then releases the reclaimable space now empty after the batch delete process back to the system:

```
REORG TABLE T1 RECLAIM EXTENTS
```

Note: The table remains fully available to all users while the REORG command is processed.

Olivia then repeats the command to determine how much space was released to the table space:

```
SELECT T.SPARSE_BLOCKS, T.RECLAIMABLE_SPACE FROM TABLE
       (SYSPROC.ADMIN_GET_TAB_INFO('OLIVIA,','T1')) T

SPARSE_BLOCKS        RECLAIMABLE_SPACE
-------------------- -----------------
```

```
1                   30433

  1 record(s) selected.
```

The result is 14,826,751,224 KB of space that is formerly occupied by data is reclaimed. Since the RECLAIM EXTENTS operation is an online operation, Olivia notes that the sparse blocks and reclaimable space are not zero when the operation is complete. Other activity on the table occurred while the RECLAIM EXTENTS operation completed.

Scenario: ExampleBANK reclaiming table and index space - Evaluating the effectiveness of reclaiming space from an index:

Olivia notes the space reclaimed from the data portion of the table T1 after a batch delete. Olivia knows that there is some cleanup left to be done in the indexes for this table.

Reclaiming index space can recover space occupied by indexes while the same indexes are still available for use by users.

As with reclaiming space from the table, the space in question is reclaimed back to the table space for reuse by other objects.

Olivia uses the ADMIN_GET_INDEX_INFO function to see how much space can be reclaimed:

```
SELECT SUBSTR(INDNAME,1,10) AS INDNAME, IID, INDEX_OBJECT_L_SIZE, INDEX_OBJECT_P_SIZE,
  RECLAIMABLE_SPACE FROM TABLE(SYSPROC.ADMIN_GET_INDEX_INFO('','OLIVIA','T1'))
  AS INDEXINFO WHERE INDNAME='INX1'

       INDNAME    IID    INDEX_OBJECT_L_SIZE  INDEX_OBJECT_P_SIZE  RECLAIMABLE_SPACE
       ---------- ------ -------------------- -------------------- --------------------
       INX1            3              1106752              1106752               846592

         1 record(s) selected.
```

Olivia uses the **REORG** command with the new parameters added for reclaiming index space to finish the cleanup in the index:

```
REORG INDEXES ALL FOR TABLE T1 ALLOW WRITE ACCESS CLEANUP ALL RECLAIM EXTENTS
```

Olivia then repeats the command to determine how much space was released to the table space:

```
SELECT SUBSTR(INDNAME,1,10) AS INDNAME, IID, INDEX_OBJECT_L_SIZE, INDEX_OBJECT_P_SIZE,
  RECLAIMABLE_SPACE FROM TABLE(SYSPROC.ADMIN_GET_INDEX_INFO('','OLIVIA','T1'))
  AS INDEXINFO WHERE INDNAME='INX1'

       INDNAME    IID    INDEX_OBJECT_L_SIZE  INDEX_OBJECT_P_SIZE  RECLAIMABLE_SPACE
       ---------- ------ -------------------- -------------------- --------------------
       INX1            3               259776               259776                    0

         1 record(s) selected.
```

The result is an estimated 846,592 KB of space reclaimed. If the physical size after space is reclaimed is subtracted from the original physical size, Olivia notes that the actual space reclaimed is 846,976 KB.

Scenario: ExampleBANK reclaiming table and index space - Converting an existing table to an insert time clustering table:

Olivia sees the benefit of using insert time clustering tables. Olivia now wants to use this solution on existing tables in the production database. This change is accomplished by using the online table move utility.

Olivia has a table that exists on a system with the following schema. In this scenario, assume that the table actually has a column which is useful for placing data in approximate insert time order (C4).

```
CREATE TABLE EXMP.T1 (C1 INT, C2 CHAR(50), C3 BIGINT, C4 DATE) IN TABLESPACE1

CREATE INDEX INX1 ON EXMP.T1(C4)
```

Olivia now creates the target table for the conversion:

```
DB2 CREATE TABLE EXMP.NEWT1(C1 INT, C2 CHAR(50), C3 BIGINT, C4 DATE)
  IN TABLESPACE1 ORGANIZE BY INSERT TIME
```

The schema is identical to the original table but by using the ORGANIZE BY INSERT TIME keywords, Olivia ensures that this table is clustered by time.

Olivia uses the online table move stored procedure to perform the conversion.

Since a clustering index exists on column C4, it gives Olivia a good approximation of insert time ordering. For tables that do not have such a column, the space reclamation benefits of moving to an insert time clustering table is not apparent for some time. This benefit is not immediately apparent because newer data is grouped together with older data.

```
DB2 CALL SYSPROC.ADMIN_MOVE_TABLE('EXMP', 'T1', 'NEWT1', NULL, 'MOVE')
```

EXMP.T1 is now in a time clustering table format. It is ready to have extents reclaimed after subsequent batch deletions.

Scenario: ExampleBANK reclaiming table and index space - Improving row overflow performance:

During normal SQL processing, a row value update might result in that row no longer fitting in the original location in the database. When this scenario occurs, the database manager splits the row into two pieces. In the original location, the row is replaced with a pointer. The location that the new pointer indicates is where the larger and new copy of the row can be found. Any subsequent access to the updated row now follows this pointer, causing performance degradation.

Olivia checks whether the table T1 is suffering from the additional performance cost that is associated with overflow pairs:

```
SELECT T.TABNAME, T.OVERFLOW_ACCESSES FROM TABLE(SYSPROC.MON_GET_TABLE('','',0))
   WHERE TABNAME = 'T1' ORDER BY OVERFLOW_ACCESSES

TABNAME                                         OVERFLOW_ACCESSES
----------------------------------------------- --------------------
T1                                                               172

  1 record(s) selected.
```

Olivia notes that T1 might benefit from a CLEANUP reorganization to reduce the number of overflow accesses. Olivia uses the following command on each table:

```
REORG TABLE T1 INPLACE CLEANUP OVERFLOWS
```

Olivia can then rerun the original monitor command after this reorganization operation. Olivia notices the number of new pointer or overflow accesses is reduced to 0.

```
SELECT T.TABNAME, T.OVERFLOW_ACCESSES FROM TABLE(SYSPROC.MON_GET_TABLE('','',0))
   WHERE TABNAME = 'T1' ORDER BY OVERFLOW_ACCESSES

TABNAME                                                  OVERFLOW_ACCESSES
-------------------------------------------------------- --------------------
T1                                                                          0

  1 record(s) selected.
```

Scenario: ExampleBANK reclaiming table and index space - General index maintenance:

Olivia notes that, for some indexes and tables, space consumption and behavior are not closely tracked. Periodically, a script can check whether any space in the affected table spaces can be cleaned up and reclaimed.

Olivia uses the **REORGCHK** command to determine whether an index cleanup would be beneficial:

```
REORGCHK ON TABLE USER1.TBL1;

Doing RUNSTATS ....


Table statistics:

F1: 100 * OVERFLOW / CARD < 5
F2: 100 * (Effective Space Utilization of Data Pages) > 70
F3: 100 * (Required Pages / Total Pages) > 80

SCHEMA.NAME                     CARD     OV     NP     FP ACTBLK    TSIZE  F1  F2  F3 REORG
-------------------------------------------------------------------------------------------
Table: USER1.TBL1
                                3400      0    201    295      -   775200   0  67  70 -**
-------------------------------------------------------------------------------------------

Index statistics:

F4: CLUSTERRATIO or normalized CLUSTERFACTOR > 80
F5: 100 * (Space used on leaf pages / Space available on non-empty leaf pages) >
  MIN(50, (100 - PCTFREE))
F6: (100 - PCTFREE) * (Amount of space available in an index with one less level /
  Amount of space required for all keys) < 100
F7: 100 * (Number of pseudo-deleted RIDs / Total number of RIDs) < 20
F8: 100 * (Number of pseudo-empty leaf pages / Total number of leaf pages) < 20

SCHEMA.NAME                 INDCARD  LEAF ELEAF LVLS  NDEL    KEYS LEAF_RECSIZE
-------------------------------------------------------------------------------
Table: USER1.TBL1
Index: USER1.TBL1_INX1
                              3400   244     0    3  1595    2882          205
-------------------------------------------------------------------------------
(continued)
NLEAF_RECSIZE LEAF_PAGE_OVERHEAD NLEAF_PAGE_OVERHEAD  PCT_PAGES_SAVED  F4  F5  F6
-------------------------------------------------------------------------------


          205               132                132               0 100  64   9
-------------------------------------------------------------------------------
(continued)
F7  F8 REORG
------------


31   0 ---*-
------------

CLUSTERRATIO or normalized CLUSTERFACTOR (F4) will indicate REORG is necessary
for indexes that are not in the same sequence as the base table. When multiple
indexes are defined on a table, one or more indexes may be flagged as needing
```

```
REORG.  Specify the most important index for REORG sequencing.

Tables defined using the ORGANIZE BY clause and the corresponding dimension
indexes have a '*' suffix to their names. The cardinality of a dimension index
is equal to the Active blocks statistic of the table.
```

Formula F7 in the output shows that, for the TBL1_INX1 index, an index cleanup using the **REORG** command would be beneficial. Olivia issues the command to clean up the indexes:

```
REORG INDEXES ALL FOR TABLE USER1.TBL1 ALLOW WRITE ACCESS CLEANUP;
```

To determine how much space can be reclaimed now that the **REORG INDEXES CLEANUP** command freed up space, Olivia uses the ADMIN_GET_INDEX_INFO routine:

```
SELECT RECLAIMABLE_SPACE
  FROM TABLE(sysproc.admin_get_index_info('T','USER1', 'TBL1'))
  AS t";

RECLAIMABLE_SPACE
-----------------
            14736

1 record(s) selected.
```

If Olivia considers this value, in KB, to be significant, she can run the **REORG INDEX RECLAIM EXTENTS** command:

```
REORG INDEXES ALL FOR TABLE USER1.TBL1 ALLOW WRITE ACCESS RECLAIM EXTENTS;
```

Olivia can schedule this work at regular intervals to ensure that the indexes in question do not hold more space than required. This regularly scheduled work does not prohibit others from using the indexes in question.

Scenario: ExampleBANK reclaiming table and index space - AUTO_REORG policy:

Olivia can opt to automatically manage index reorganization in place of performing manual checks.

In the samples directory SQLLIB/samples/automaintcfg, Olivia finds a file called DB2AutoReorgPolicySample.xml. In this file, Olivia notes the following line:

```
<ReorgOptions dictionaryOption="Keep" indexReorgMode="Offline"
 useSystemTempTableSpace="false" />
```

Olivia decides that the reclaimExtentsSizeForIndexObjects threshold must exceed 51,200 KB (50 MB) before any automatic reorganization with the `RECLAIM EXTENTS` option is run. Olivia copies DB2AutoReorgPolicySample.xml to a file called autoreorg_policy.xml and changes the line in the sample to the following value:

```
<ReorgOptions dictionaryOption="Keep" indexReorgMode="Online"
 useSystemTempTableSpace="false" reclaimExtentsSizeForIndexObjects="51200"  />
```

Olivia then sets the policy:

```
cp $HOME/autoreorg_policy.xml $HOME/sqllib/tmp/.
db2 "call sysproc.automaint_set_policyfile( 'AUTO_REORG', 'autoreorg_policy.xml')"
```

As a final step, Olivia ensures that AUTO REORG is enabled. For more details, see the related tasks.

Now that the policy is defined and AUTO REORG is enabled, automatic reorganization starts when the reclaimExtentsSizeForIndexObjects threshold exceeds 51,200 KB (50 MB).

Application design

Database application design is one of the factors that affect application performance. Review this section for details about application design considerations that can help you to maximize the performance of database applications.

Application processes, concurrency, and recovery

All SQL programs execute as part of an *application process* or agent. An application process involves the execution of one or more programs, and is the unit to which the database manager allocates resources and locks. Different application processes might involve the execution of different programs, or different executions of the same program.

More than one application process can request access to the same data at the same time. *Locking* is the mechanism that is used to maintain data integrity under such conditions, preventing, for example, two application processes from updating the same row of data simultaneously.

The database manager acquires locks to prevent uncommitted changes made by one application process from being accidentally perceived by any other process. The database manager releases all locks it has acquired and retained on behalf of an application process when that process ends. However, an application process can explicitly request that locks be released sooner. This is done using a *commit* operation, which releases locks that were acquired during a unit of work and also commits database changes that were made during the unit of work.

A *unit of work* (UOW) is a recoverable sequence of operations within an application process. A unit of work is initiated when an application process starts, or when the previous UOW ends because of something other than the termination of the application process. A unit of work ends with a commit operation, a rollback operation, or the end of an application process. A commit or rollback operation affects only the database changes that were made within the UOW that is ending.

The database manager provides a means of backing out of uncommitted changes that were made by an application process. This might be necessary in the event of a failure on the part of an application process, or in the case of a deadlock or lock timeout situation. An application process can explicitly request that its database changes be cancelled. This is done using a *rollback* operation.

As long as these changes remain uncommitted, other application processes are unable to see them, and the changes can be rolled back. This is not true, however, if the prevailing isolation level is uncommitted read (UR). After they are committed, these database changes are accessible to other application processes and can no longer be rolled back.

Both Db2 call level interface (CLI) and embedded SQL allow for a connection mode called *concurrent transactions*, which supports multiple connections, each of which is an independent transaction. An application can have multiple concurrent connections to the same database.

Locks that are acquired by the database manager on behalf of an application process are held until the end of a UOW, except when the isolation level is cursor stability (CS, in which the lock is released as the cursor moves from row to row) or uncommitted read (UR).

An application process is never prevented from performing operations because of its own locks. However, if an application uses concurrent transactions, the locks from one transaction might affect the operation of a concurrent transaction.

The initiation and the termination of a UOW define *points of consistency* within an application process. For example, a banking transaction might involve the transfer of funds from one account to another. Such a transaction would require that these funds be subtracted from the first account, and then added to the second account. Following the subtraction step, the data is inconsistent. Only after the funds have been added to the second account is consistency reestablished. When both steps are complete, the commit operation can be used to end the UOW, thereby making the changes available to other application processes. If a failure occurs before the UOW ends, the database manager will roll back any uncommitted changes to restore data consistency.

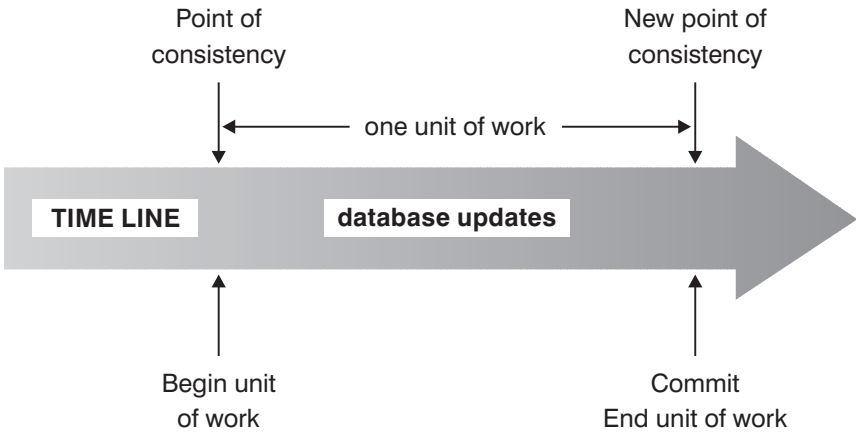


Figure 21. Unit of work with a COMMIT statement

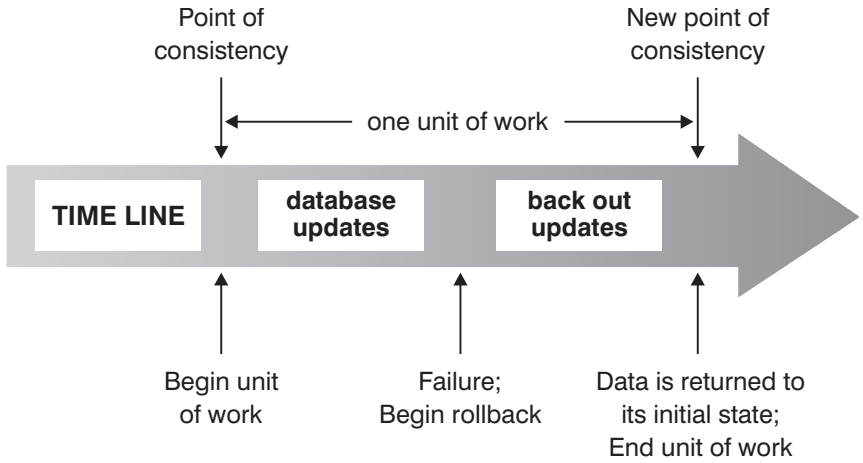


Figure 22. Unit of work with a ROLLBACK statement

Concurrency issues

Because many users access and change data in a relational database, the database manager must allow users to make these changes while ensuring that data integrity is preserved.

Concurrency refers to the sharing of resources by multiple interactive users or application programs at the same time. The database manager controls this access to prevent undesirable effects, such as:

- **Lost updates**. Two applications, A and B, might both read the same row and calculate new values for one of the columns based on the data that these applications read. If A updates the row and then B also updates the row, A's update lost.
- **Access to uncommitted data**. Application A might update a value, and B might read that value before it is committed. Then, if A backs out of that update, the calculations performed by B might be based on invalid data.
- **Non-repeatable reads**. Application A might read a row before processing other requests. In the meantime, B modifies or deletes the row and commits the change. Later, if A attempts to read the original row again, it sees the modified row or discovers that the original row has been deleted.
- **Phantom reads**. Application A might execute a query that reads a set of rows based on some search criterion. Application B inserts new data or updates existing data that would satisfy application A's query. Application A executes its query again, within the same unit of work, and some additional (“phantom”) values are returned.

Concurrency is not an issue for global temporary tables, because they are available only to the application that declares or creates them.

Concurrency control in federated database systems

A *federated database system* supports applications and users submitting SQL statements that reference two or more database management systems (DBMSs) in a single statement. To reference such data sources (each consisting of a DBMS and data), the Db2 server uses nicknames. *Nicknames* are aliases for objects in other DBMSs. In a federated system, the Db2 server relies on the concurrency control protocols of the database manager that hosts the requested data.

A Db2 federated system provides *location transparency* for database objects. For example, if information about tables and views is moved, references to that information (through nicknames) can be updated without changing the applications that request this information. When an application accesses data through nicknames, the Db2 server relies on concurrency control protocols at the data source to ensure that isolation levels are enforced. Although the Db2 server tries to match the isolation level that is requested at the data source with a logical equivalent, results can vary, depending on data source capabilities.

Isolation levels:

The *isolation level* that is associated with an application process determines the degree to which the data that is being accessed by that process is locked or isolated from other concurrently executing processes. The isolation level is in effect for the duration of a unit of work.

The isolation level of an application process therefore specifies:

- The degree to which rows that are read or updated by the application are available to other concurrently executing application processes
- The degree to which the update activity of other concurrently executing application processes can affect the application

The isolation level for static SQL statements is specified as an attribute of a package and applies to the application processes that use that package. The isolation level is specified during the program preparation process by setting the ISOLATION bind or precompile option. For dynamic SQL statements, the default isolation level is the isolation level that was specified for the package preparing the statement. Use the SET CURRENT ISOLATION statement to specify a different isolation level for dynamic SQL statements that are issued within a session. For more information, see "CURRENT ISOLATION special register". For both static SQL statements and dynamic SQL statements, the *isolation-clause* in a *select-statement* overrides both the special register (if set) and the bind option value. For more information, see "Select-statement".

Isolation levels are enforced by locks, and the type of lock that is used limits or prevents access to the data by concurrent application processes. Declared temporary tables and their rows cannot be locked because they are only accessible to the application that declared them.

The database manager supports three general categories of locks:

Share (S)
Under an S lock, concurrent application processes are limited to read-only operations on the data.

Update (U)
Under a U lock, concurrent application processes are limited to read-only operations on the data, if these processes have not declared that they might update a row. The database manager assumes that the process currently looking at a row might update it.

Exclusive (X)
Under an X lock, concurrent application processes are prevented from accessing the data in any way. This does not apply to application processes with an isolation level of uncommitted read (UR), which can read but not modify the data.

Regardless of the isolation level, the database manager places exclusive locks on every row that is inserted, updated, or deleted. Thus, all isolation levels ensure that any row that is changed by an application process during a unit of work is not changed by any other application process until the unit of work is complete.

The database manager supports four isolation levels.

- "Repeatable read (RR)" on page 158
- "Read stability (RS)" on page 158
- "Cursor stability (CS)" on page 159
- "Uncommitted read (UR)" on page 160

Note: Some host database servers support the *no commit (NC)* isolation level. On other database servers, this isolation level behaves like the uncommitted read isolation level.

Note: The *lost updates (LU)* concurrency issue is not allowed by any of the Db2 product's four isolation levels.

A detailed description of each isolation level follows, in decreasing order of performance impact, but in increasing order of the care that is required when accessing or updating data.

Repeatable read (RR)

The *repeatable read* isolation level locks all the rows that an application references during a unit of work (UOW). If an application issues a SELECT statement twice within the same unit of work, the same result is returned each time. Under RR, lost updates, access to uncommitted data, non-repeatable reads, and phantom reads are not possible.

Under RR, an application can retrieve and operate on the rows as many times as necessary until the UOW completes. However, no other application can update, delete, or insert a row that would affect the result set until the UOW completes. Applications running under the RR isolation level cannot see the uncommitted changes of other applications. This isolation level ensures that all returned data remains unchanged until the time the application sees the data, even when temporary tables or row blocking is used.

Every referenced row is locked, not just the rows that are retrieved. For example, if you scan 10 000 rows and apply predicates to them, locks are held on all 10 000 rows, even if, say, only 10 rows qualify. Another application cannot insert or update a row that would be added to the list of rows referenced by a query if that query were to be executed again. This prevents phantom reads.

Because RR can acquire a considerable number of locks, this number might exceed limits specified by the **locklist** and **maxlocks** database configuration parameters. To avoid lock escalation, the optimizer might elect to acquire a single table-level lock for an index scan, if it appears that lock escalation is likely. If you do not want table-level locking, use the read stability isolation level.

While evaluating referential constraints, the Db2 server might occasionally upgrade the isolation level used on scans of the foreign table to RR, regardless of the isolation level that was previously set by the user. This results in additional locks being held until commit time, which increases the likelihood of a deadlock or a lock timeout. To avoid these problems, create an index that contains only the foreign key columns, which the referential integrity scan can use instead.

Read stability (RS)

The *read stability* isolation level locks only those rows that an application retrieves during a unit of work. RS ensures that any qualifying row read during a UOW cannot be changed by other application processes until the UOW completes, and that any change to a row made by another application process cannot be read until the change is committed by that process. Under RS, access to uncommitted data and non-repeatable reads are not possible. However, phantom reads are possible. Phantom reads might also be introduced by concurrent updates to rows where the old value did not satisfy the search condition of the original application but the new updated value does.

For example, a phantom row can occur in the following situation:

1. Application process P1 reads the set of rows n that satisfy some search condition.
2. Application process P2 then inserts one or more rows that satisfy the search condition and commits those new inserts.
3. P1 reads the set of rows again with the same search condition and obtains both the original rows and the rows inserted by P2.

In a Db2 pureScale environment, an application running at this isolation level might reject a previously committed row value if the row is updated concurrently on a different member. To override this behavior, specify the `WAIT_FOR_OUTCOME` option.

This isolation level ensures that all returned data remains unchanged until the time the application sees the data, even when temporary tables or row blocking is used.

The RS isolation level provides both a high degree of concurrency and a stable view of the data. To that end, the optimizer ensures that table-level locks are not obtained until lock escalation occurs.

The RS isolation level is suitable for an application that:

- Operates in a concurrent environment
- Requires qualifying rows to remain stable for the duration of a unit of work
- Does not issue the same query more than once during a unit of work, or does not require the same result set when a query is issued more than once during a unit of work

Cursor stability (CS)

The *cursor stability* isolation level locks any row being accessed during a transaction while the cursor is positioned on that row. This lock remains in effect until the next row is fetched or the transaction terminates. However, if any data in the row was changed, the lock is held until the change is committed.

Under this isolation level, no other application can update or delete a row while an updatable cursor is positioned on that row. Under CS, access to the uncommitted data of other applications is not possible. However, non-repeatable reads and phantom reads are possible.

CS is the default isolation level. It is suitable when you want maximum concurrency and need to see only committed data. Scans performed under this isolation level behaves according to the configuration parameter **`cur_commit`** (Currently Committed).

In a Db2 pureScale environment, an application running at this isolation level may return or reject a previously committed row value if the row is concurrently updated on a different member. The `WAIT FOR OUTCOME` option of the concurrent access resolution setting can be used to override this behavior.

Note: Under the *currently committed* semantics introduced in Version 9.7, only committed data is returned, as was the case previously, but now readers do not wait for updaters to release row locks. Instead, readers return data that is based on the currently committed version; that is, data prior to the start of the write operation.

Uncommitted read (UR)

The *uncommitted read* isolation level allows an application to access the uncommitted changes of other transactions. Moreover, UR does not prevent another application from accessing a row that is being read, unless that application is attempting to alter or drop the table.

Under UR, access to uncommitted data, non-repeatable reads, and phantom reads are possible. This isolation level is suitable if you run queries against read-only tables, or if you issue SELECT statements only, and seeing data that has not been committed by other applications is not a problem.

UR works differently for read-only and updatable cursors.

- Read-only cursors can access most of the uncommitted changes of other transactions.
- Tables, views, and indexes that are being created or dropped by other transactions are not available while the transaction is processing. Any other changes by other transactions can be read before they are committed or rolled back. Updatable cursors operating under UR behave as though the isolation level were CS.

If an uncommitted read application uses ambiguous cursors, it might use the CS isolation level when it runs. The ambiguous cursors can be escalated to CS if the value of the BLOCKING option on the **PREP** or **BIND** command is UNAMBIG (the default). To prevent this escalation:

- Modify the cursors in the application program to be unambiguous. Change the SELECT statements to include the FOR READ ONLY clause.
- Let the cursors in the application program remain ambiguous, but precompile the program or bind it with the BLOCKING ALL and STATICREADONLY YES options to enable the ambiguous cursors to be treated as read-only when the program runs.

Comparison of isolation levels

Table 6 summarizes the supported isolation levels.

Table 6. Comparison of isolation levels

	UR	CS	RS	RR
Can an application see uncommitted changes made by other application processes?	Yes	No	No	No
Can an application update uncommitted changes made by other application processes?	No	No	No	No
Can the re-execution of a statement be affected by other application processes? [1]	Yes	Yes	Yes	No [2]
Can updated rows be updated by other application processes? [3]	No	No	No	No
Can updated rows be read by other application processes that are running at an isolation level other than UR?	No	No	No	No
Can updated rows be read by other application processes that are running at the UR isolation level?	Yes	Yes	Yes	Yes

Table 6. Comparison of isolation levels (continued)

	UR	CS	RS	RR
Can accessed rows be updated by other application processes? [4]	Yes	Yes	No	No
Can accessed rows be read by other application processes?	Yes	Yes	Yes	Yes
Can the current row be updated or deleted by other application processes? [5]	Yes/No [6]	Yes/No [6]	No	No

Note:

1. An example of the *phantom read phenomenon* is as follows: Unit of work UW1 reads the set of *n* rows that satisfies some search condition. Unit of work UW2 inserts one or more rows that satisfy the same search condition and then commits. If UW1 subsequently repeats its read with the same search condition, it sees a different result set: the rows that were read originally plus the rows that were inserted by UW2.
2. If your label-based access control (LBAC) credentials change between reads, results for the second read might be different because you have access to different rows.
3. The isolation level offers no protection to the application if the application is both reading from and writing to a table. For example, an application opens a cursor on a table and then performs an insert, update, or delete operation on the same table. The application might see inconsistent data when more rows are fetched from the open cursor.
4. An example of the *non-repeatable read phenomenon* is as follows: Unit of work UW1 reads a row. Unit of work UW2 modifies that row and commits. If UW1 subsequently reads that row again, it might see a different value.
5. An example of the *dirty read phenomenon* is as follows: Unit of work UW1 modifies a row. Unit of work UW2 reads that row before UW1 commits. If UW1 subsequently rolls the changes back, UW2 has read nonexisting data.
6. Under UR or CS, if the cursor is not updatable, the current row can be updated or deleted by other application processes in some cases. For example, buffering might cause the current row at the client to be different from the current row at the server. Moreover, when using currently committed semantics under CS, a row that is being read might have uncommitted updates pending. In this case, the currently committed version of the row is always returned to the application.

Summary of isolation levels

Table 7 lists the concurrency issues that are associated with different isolation levels.

Table 7. Summary of isolation levels

Isolation level	Access to uncommitted data	Non-repeatable reads	Phantom reads
Repeatable read (RR)	Not possible	Not possible	Not possible
Read stability (RS)	Not possible	Not possible	Possible
Cursor stability (CS)	Not possible	Possible	Possible
Uncommitted read (UR)	Possible	Possible	Possible

The isolation level affects not only the degree of isolation among applications but also the performance characteristics of an individual application, because the processing and memory resources that are required to obtain and free locks vary

with the isolation level. The potential for deadlocks also varies with the isolation level. Table 8 provides a simple heuristic to help you choose an initial isolation level for your application.

Table 8. Guidelines for choosing an isolation level

Application type	High data stability required	High data stability not required
Read-write transactions	RS	CS
Read-only transactions	RR or RS	UR

Specifying the isolation level:

Because the isolation level determines how data is isolated from other processes while the data is being accessed, you should select an isolation level that balances the requirements of concurrency and data integrity.

About this task

The isolation level that you specify is in effect for the duration of the unit of work (UOW). The following heuristics are used to determine which isolation level will be used when compiling an SQL or XQuery statement:

- For static SQL:
  - If an *isolation-clause* is specified in the statement, the value of that clause is used.
  - If an *isolation-clause* is not specified in the statement, the isolation level that was specified for the package when the package was bound to the database is used.
- For dynamic SQL:
  - If an *isolation-clause* is specified in the statement, the value of that clause is used.
  - If an *isolation-clause* is not specified in the statement, and a SET CURRENT ISOLATION statement has been issued within the current session, the value of the CURRENT ISOLATION special register is used.
  - If an *isolation-clause* is not specified in the statement, and a SET CURRENT ISOLATION statement has not been issued within the current session, the isolation level that was specified for the package when the package was bound to the database is used.
- For static or dynamic XQuery statements, the isolation level of the environment determines the isolation level that is used when the XQuery expression is evaluated.

Note: Many commercially-written applications provide a method for choosing the isolation level. Refer to the application documentation for information.

The isolation level can be specified in several different ways.

Procedure

- **At the statement or subselect level:**

  Note: Isolation levels for XQuery statements cannot be specified at the statement level.

Use the WITH clause. The WITH UR option applies to read-only operations only. In other cases, the statement is automatically changed from UR to CS.

This isolation level overrides the isolation level that is specified for the package in which the statement appears. You can specify an isolation level for the following SQL statements:

- DECLARE CURSOR
- Searched DELETE
- INSERT
- SELECT
- SELECT INTO
- Searched UPDATE

- **For dynamic SQL within the current session:**

  Use the SET CURRENT ISOLATION statement to set the isolation level for dynamic SQL issued within a session. Issuing this statement sets the CURRENT ISOLATION special register to a value that specifies the isolation level for any dynamic SQL statements that are issued within the current session. Once set, the CURRENT ISOLATION special register provides the isolation level for any subsequent dynamic SQL statement that is compiled within the session, regardless of which package issued the statement. This isolation level is in effect until the session ends or until the SET CURRENT ISOLATION...RESET statement is issued.

- **At precompile or bind time:**

  For an application written in a supported compiled language, use the ISOLATION option of the **PREP** or **BIND** commands. You can also use the sqlaprep or sqlabndx API to specify the isolation level.

  - If you create a bind file at precompile time, the isolation level is stored in the bind file. If you do not specify an isolation level at bind time, the default is the isolation level that was used during precompilation.
  - If you do not specify an isolation level, the default level of cursor stability (CS) is used.

  To determine the isolation level of a package, execute the following query:

```
select isolation from syscat.packages
  where pkgname = 'pkgname'
    and pkgschema = 'pkgschema'
```

  where *pkgname* is the unqualified name of the package and *pkgschema* is the schema name of the package. Both of these names must be specified in uppercase characters.

- **When working with JDBC or SQLJ at run time:**

  Note: JDBC and SQLJ are implemented with CLI on Db2 servers, which means that the `db2cli.ini` settings might affect what is written and run using JDBC and SQLJ.

  To create a package (and specify its isolation level) in SQLJ, use the SQLJ profile customizer (**db2sqljcustomize** command).

- **From CLI or ODBC at run time:**

  Use the **CHANGE ISOLATION LEVEL** command. With Db2 Call-level Interface (CLI), you can change the isolation level as part of the CLI configuration. At run time, use the SQLSetConnectAttr function with the SQL_ATTR_TXN_ISOLATION attribute to set the transaction isolation level for the current connection referenced by the *ConnectionHandle* argument. You can also use the TXNISOLATION keyword in the `db2cli.ini` file.

- **On database servers that support REXX:**

  When a database is created, multiple bind files that support the different isolation levels for SQL in REXX are bound to the database. Other command line processor (CLP) packages are also bound to the database when a database is created.

  REXX and the CLP connect to a database using the default CS isolation level. Changing this isolation level does not change the connection state.

  To determine the isolation level that is being used by a REXX application, check the value of the SQLISL predefined REXX variable. The value is updated each time that the **CHANGE ISOLATION LEVEL** command executes.

Results

Currently committed semantics:

Under *currently committed* semantics, only committed data is returned to readers. However, readers do not wait for writers to release row locks. Instead, readers return data that is based on the currently committed version of data: that is, the version of the data before the start of the write operation.

Lock timeouts and deadlocks can occur under the cursor stability (CS) isolation level with row-level locking, especially with applications that are not designed to prevent such problems. Some high-throughput database applications cannot tolerate waiting on locks that are issued during transaction processing. Also, some applications cannot tolerate processing uncommitted data but still require non-blocking behavior for read transactions.

Currently committed semantics are turned on by default for new databases. You do not have to make application changes to take advantage of the new behavior. To override the default behavior, Set the **cur_commit** database configuration parameter to DISABLED. Overriding the behavior might be useful, for example, if applications require the blocking of writers to synchronize internal logic. During database upgrade from V9.5 or earlier, the **cur_commit** configuration parameter is set to DISABLED to maintain the same behavior as in previous releases. If you want to use currently committed on cursor stability scans, you need to set the **cur_commit** configuration parameter to ON after the upgrade.

Currently committed semantics apply only to read-only scans that do not involve catalog tables and internal scans that are used to evaluate or enforce constraints. Because currently committed semantics are decided at the scan level, the access plan of a writer might include currently committed scans. For example, the scan for a read-only subquery can involve currently committed semantics.

Because currently committed semantics obey isolation level semantics, applications running under currently committed semantics continue to respect isolation levels.

Currently committed semantics require increased log space for writers. Additional space is required for logging the first update of a data row during a transaction. This data is required for retrieving the currently committed image of the row. Depending on the workload, this can have an insignificant or measurable impact on the total log space used. The requirement for additional log space does not apply when **cur_commit** database configuration parameter is set to DISABLED.

Restrictions

The following restrictions apply to currently committed semantics:

- The target table object in a section that is to be used for data update or deletion operations does not use currently committed semantics. Rows that are to be modified must be lock protected to ensure that they do not change after they have satisfied any query predicates that are part of the update operation.
- A transaction that makes an uncommitted modification to a row forces the currently committed reader to access appropriate log records to determine the currently committed version of the row. Although log records that are no longer in the log buffer can be physically read, currently committed semantics do not support the retrieval of log files from the log archive. This affects only databases that you configure to use infinite logging.
- The following scans do not use currently committed semantics:
  - Catalog table scans
  - Scans that are used to enforce referential integrity constraints
  - Scans that reference LONG VARCHAR or LONG VARGRAPHIC columns
  - Range-clustered table (RCT) scans
  - Scans that use spatial or extended indexes
- In a Db2 pureScale environment, currently committed semantics only apply to lock conflicts between applications running on the same member. If the lock conflict is with an application on a remote member, the requester of the lock waits for the lock to be released before processing the row.

Example

Consider the following scenario, in which deadlocks are avoided by using currently committed semantics. In this scenario, two applications update two separate tables, as shown in step 1, but do not yet commit. Each application then attempts to use a read-only cursor to read from the table that the other application updated, as shown in step 2. These applications are running under the CS isolation level.

Step	Application A	Application B
1	update T1 set col1 = ? where col2 = ?	update T2 set col1 = ? where col2 = ?
2	select col1, col3, col4 from T2 where col2 >= ?	select col1, col5, from T1 where col5 = ? and col2 = ?
3	commit	commit

Without currently committed semantics, these applications running under the cursor stability isolation level might create a deadlock, causing one of the applications to fail. This happens when each application must read data that is being updated by the other application.

Under currently committed semantics, if one of the applications that is running a query in step 2 requires the data that is being updated by the other application, the first application does not wait for the lock to be released. As a result, a deadlock is impossible. The first application locates and uses the previously committed version of the data instead.

Option to disregard uncommitted insertions:

The **DB2_SKIPINSERTED** registry variable controls whether or not uncommitted data insertions can be ignored for statements that use the cursor stability (CS) or the read stability (RS) isolation level.

Uncommitted insertions are handled in one of two ways, depending on the value of the **DB2_SKIPINSERTED** registry variable.

- When the value is ON, the Db2 server ignores uncommitted insertions, which in many cases can improve concurrency and is the preferred behavior for most applications. Uncommitted insertions are treated as though they had not yet occurred.
- When the value is OFF (the default), the Db2 server waits until the insert operation completes (commits or rolls back) and then processes the data accordingly. This is appropriate in certain cases. For example:
  - Suppose that two applications use a table to pass data between themselves, with the first application inserting data into the table and the second one reading it. The data must be processed by the second application in the order presented, such that if the next row to be read is being inserted by the first application, the second application must wait until the insert operation commits.
  - An application avoids UPDATE statements by deleting data and then inserting a new image of the data.

Evaluate uncommitted data through lock deferral:

To improve concurrency, the database manager in some situations permits the deferral of row locks for CS or RS isolation scans until a row is known to satisfy the predicates of a query.

By default, when row-level locking is performed during a table or index scan, the database manager locks each scanned row whose commitment status is unknown before determining whether the row satisfies the predicates of the query.

To improve the concurrency of such scans, enable the **DB2_EVALUNCOMMITTED** registry variable so that predicate evaluation can occur on uncommitted data. A row that contains an uncommitted update might not satisfy the query, but if predicate evaluation is deferred until after the transaction completes, the row might indeed satisfy the query.

Uncommitted deleted rows are skipped during table scans, and the database manager skips deleted keys during index scans if the **DB2_SKIPDELETED** registry variable is enabled.

The **DB2_EVALUNCOMMITTED** registry variable setting applies at compile time for dynamic SQL or XQuery statements, and at bind time for static SQL or XQuery statements. This means that even if the registry variable is enabled at run time, the lock avoidance strategy is not deployed unless **DB2_EVALUNCOMMITTED** was enabled at bind time. If the registry variable is enabled at bind time but not enabled at run time, the lock avoidance strategy is still in effect. For static SQL or XQuery statements, if a package is rebound, the registry variable setting that is in effect at bind time is the setting that applies. An implicit rebind of static SQL or XQuery statements will use the current setting of the **DB2_EVALUNCOMMITTED** registry variable.

Applicability of evaluate uncommitted for different access plans

Table 9. RID Index Only Access

Predicates	Evaluate Uncommitted
None	No
SARGable	Yes

Table 10. Data Only Access (relational or deferred RID list)

Predicates	Evaluate Uncommitted
None	No
SARGable	Yes

Table 11. RID Index + Data Access

Predicates		Evaluate Uncommitted	
Index	**Data**	**Index access**	**Data access**
None	None	No	No
None	SARGable	No	No
SARGable	None	Yes	No
SARGable	SARGable	Yes	No

Table 12. Block Index + Data Access

Predicates		Evaluate Uncommitted	
Index	**Data**	**Index access**	**Data access**
None	None	No	No
None	SARGable	No	Yes
SARGable	None	Yes	No
SARGable	SARGable	Yes	Yes

Example

The following example provides a comparison between the default locking behavior and the evaluate uncommitted behavior. The table is the ORG table from the SAMPLE database.

```
DEPTNUMB DEPTNAME       MANAGER DIVISION   LOCATION
-------- -------------- ------- ---------- -------------
      10 Head Office        160 Corporate  New York
      15 New England         50 Eastern    Boston
      20 Mid Atlantic        10 Eastern    Washington
      38 South Atlantic      30 Eastern    Atlanta
      42 Great Lakes        100 Midwest    Chicago
      51 Plains             140 Midwest    Dallas
      66 Pacific            270 Western    San Francisco
      84 Mountain           290 Western    Denver
```

The following transactions occur under the default cursor stability (CS) isolation level.

Table 13. Transactions against the ORG table under the CS isolation level

SESSION 1	SESSION 2
`connect to sample`	`connect to sample`
`+c update org set  deptnumb=5 where manager=160`	
	`select * from org where deptnumb >= 10`

The uncommitted UPDATE statement in Session 1 holds an exclusive lock on the first row in the table, preventing the query in Session 2 from returning a result set, even though the row being updated in Session 1 does not currently satisfy the query in Session 2. The CS isolation level specifies that any row that is accessed by a query must be locked while the cursor is positioned on that row. Session 2 cannot obtain a lock on the first row until Session 1 releases its lock.

Waiting for a lock in Session 2 can be avoided by using the evaluate uncommitted feature, which first evaluates the predicate and then locks the row. As such, the query in Session 2 would not attempt to lock the first row in the table, thereby increasing application concurrency. Note that this also means that predicate evaluation in Session 2 would occur with respect to the uncommitted value of `deptnumb=5` in Session 1. The query in Session 2 would omit the first row in its result set, despite the fact that a rollback of the update in Session 1 would satisfy the query in Session 2.

If the order of operations were reversed, concurrency could still be improved with the evaluate uncommitted feature. Under default locking behavior, Session 2 would first acquire a row lock prohibiting the searched UPDATE in Session 1 from executing, even though the Session 1 UPDATE statement would not change the row that is locked by the Session 2 query. If the searched UPDATE in Session 1 first attempted to examine rows and then locked them only if they qualified, the Session 1 query would be non-blocking.

Restrictions

- The **DB2_EVALUNCOMMITTED** registry variable must be enabled.
- The isolation level must be CS or RS.
- Row-level locking is in effect.
- SARGable evaluation predicates exist.
- Evaluate uncommitted is not applicable to scans on the system catalog tables.
- For multidimensional clustering (MDC) or insert time clustering (ITC) tables, block-level locking can be deferred for an index scan; however, block-level locking cannot be deferred for table scans.
- Lock deferral will not occur on a table that is executing an inplace table reorganization.
- For Iscan-Fetch plans, row-level locking is not deferred to the data access; rather, the row is locked during index access before moving to the row in the table.
- Deleted rows are unconditionally skipped during table scans, but deleted index keys are skipped only if the **DB2_SKIPDELETED** registry variable is enabled.

Writing and tuning queries for optimal performance

There are several ways in which you can minimize the impact of SQL statements on Db2 database performance.

You can minimize this impact by:

- Writing SQL statements that the Db2 optimizer can more easily optimize. The Db2 optimizer might not be able to efficiently run SQL statements that contain non-equality join predicates, data type mismatches on join columns, unnecessary outer joins, and other complex search conditions.
- Correctly configuring the Db2 database to take advantage of Db2 optimization functionality. The Db2 optimizer can select the optimal query access plan if you have accurate catalog statistics and choose the best optimization class for your workload.
- Using the Db2 explain functionality to review potential query access plans and determine how to tune queries for best performance.

Best practices apply to general workloads, warehouse workloads, and SAP workloads.

Although there are a number of ways to deal with specific query performance issues after an application is written, good fundamental writing and tuning practices can be widely applied early on to help improve Db2 database performance.

Query performance is not a one-time consideration. You should consider it throughout the design, development, and production phases of the application development life cycle.

SQL is a very flexible language, which means that there are many ways to get the same correct result. This flexibility also means that some queries are better than others in taking advantage of the Db2 optimizer's strengths.

During query execution, the Db2 optimizer chooses a query access plan for each SQL statement. The optimizer models the execution cost of many alternative access plans and chooses the one with the minimum estimated cost. If a query contains many complex search conditions, the Db2 optimizer can rewrite the predicate in some cases, but there are some cases where it cannot.

The time to prepare or compile an SQL statement can be long for complex queries, such as those used in business intelligence (BI) applications. You can help minimize statement compilation time by correctly designing and configuring your database. This includes choosing the correct optimization class and setting other registry variables correctly.

The optimizer also requires accurate inputs to make accurate access plan decisions. This means that you need to gather accurate statistics, and potentially use advanced statistical features, such as statistical views and column group statistics.

You can use the Db2 tools, especially the Db2 explain facility, to tune queries. The Db2 compiler can capture information about the access plans and environments of static or dynamic queries. Use this captured information to understand how individual statements are run so that you can tune them and your database manager configuration to improve performance.

Writing SQL statements:

SQL is a powerful language that enables you to specify relational expressions in syntactically different but semantically equivalent ways. However, some semantically equivalent variations are easier to optimize than others. Although the Db2 optimizer has a powerful query rewrite capability, it might not always be able to rewrite an SQL statement into the most optimal form.

Certain SQL constructs can limit the access plans that are considered by the query optimizer, and these constructs should be avoided or replaced whenever possible.

Avoiding complex expressions in search conditions:

Avoid using complex expressions in search conditions where the expressions prevent the optimizer from using the catalog statistics to estimate an accurate selectivity.

The expressions might also limit the choices of access plans that can be used to apply the predicate. During the query rewrite phase of optimization, the optimizer can rewrite a number of expressions to allow the optimizer to estimate an accurate selectivity; it cannot handle all possibilities.

Avoiding join predicates on expressions:

Using expressions on join predicates can limit the join method used.

A hash join method will be considered for join predicates with an expression, as long as none of the following types of functions are used:

- A function that reads or modifies SQL data
- A function that is non-deterministic
- A function that has an external action
- A function that uses a scratchpad

If columns from both operands of the associated join are used on the same side of the join condition, hash joins are also considered, but cannot be chosen in most cases.

If a hash join cannot be used, then a potentially slower nested loop join will be used instead, and in these cases the cardinality estimate might be inaccurate.

An example of a join with an expression that is considered for a hash join:

```
WHERE UPPER(CUST.LASTNAME) = TRANS.NAME
```

An example of a join with expression that would not benefit from a hash join, but would use a nested loop join instead:

```
WHERE RAND() > 0.5
```

Avoiding expressions over columns in local predicates:

Instead of applying an expression over columns in a local predicate, use the inverse of the expression.

Consider the following examples:

```
XPRESSN(C) = 'constant'
INTEGER(TRANS_DATE)/100 = 200802
```

You can rewrite these statements as follows:

```
C = INVERSEXPRESSN('constant')
TRANS_DATE BETWEEN 20080201 AND 20080229
```

Applying expressions over columns prevents the use of index start and stop keys, leads to inaccurate selectivity estimates, and requires extra processing at query execution time.

These expressions also prevent query rewrite optimizations such as recognizing when columns are equivalent, replacing columns with constants, and recognizing when at most one row will be returned. Further optimizations are possible after it can be proven that at most one row will be returned, so the lost optimization opportunities are further compounded. Consider the following query:

```
SELECT LASTNAME, CUST_ID, CUST_CODE FROM CUST
WHERE (CUST_ID * 100) + INT(CUST_CODE) = 123456 ORDER BY 1,2,3
```

You can rewrite it as follows:

```
SELECT LASTNAME, CUST_ID, CUST_CODE FROM CUST
WHERE CUST_ID = 1234 AND CUST_CODE = '56' ORDER BY 1,2,3
```

If there is a unique index defined on CUST_ID, the rewritten version of the query enables the query optimizer to recognize that at most one row will be returned. This avoids introducing an unnecessary SORT operation. It also enables the CUST_ID and CUST_CODE columns to be replaced by 1234 and '56', avoiding copying values from the data or index pages. Finally, it enables the predicate on CUST_ID to be applied as an index start or stop key.

It might not always be apparent when an expression is present in a predicate. This can often occur with queries that reference views when the view columns are defined by expressions. For example, consider the following view definition and query:

```
CREATE VIEW CUST_V AS
  (SELECT LASTNAME, (CUST_ID * 100) + INT(CUST_CODE) AS CUST_KEY
    FROM CUST)

SELECT LASTNAME FROM CUST_V WHERE CUST_KEY = 123456
```

The query optimizer merges the query with the view definition, resulting in the following query:

```
SELECT LASTNAME FROM CUST
WHERE (CUST_ID * 100) + INT(CUST_CODE) = 123456
```

This is the same problematic predicate described in a previous example. You can observe the result of view merging by using the explain facility to display the optimized SQL.

If the inverse function is difficult to express, consider using a generated column. For example, if you want to find a last name that fits the criteria expressed by `LASTNAME IN ('Woo', 'woo', 'WOO', 'WOo',...)`, you can create a generated column `UCASE(LASTNAME) = 'WOO'` as follows:

```
CREATE TABLE CUSTOMER (
  LASTNAME VARCHAR(100),
  U_LASTNAME VARCHAR(100) GENERATED ALWAYS AS (UCASE(LASTNAME))
)

CREATE INDEX CUST_U_LASTNAME ON CUSTOMER(U_LASTNAME)
```

Support for case-insensitive search, which was introduced in Db2 Database for Linux, UNIX, and Windows Version 9.5 Fix Pack 1, is designed to resolve the situation in this particular example. You can use _Sx attribute on a locale-sensitive UCA-based collation to control the strength of the collations. For example, a locale-sensitive UCA-based collation with the attributes _LFR_S1 is a French collation that ignores case and accent.

Avoiding no-op expressions in predicates to change the optimizer estimate:

A "no-op" coalesce() predicate of the form `COALESCE(X, X) = X` introduces an estimation error into the planning of any query that uses it. Currently the Db2 query compiler does not have the capability of dissecting that predicate and determining that all rows actually satisfy it.

As a result, the predicate artificially reduces the estimated number of rows coming from some part of a query plan. This smaller row estimate usually reduces the row and cost estimates for the rest of query planning, and sometimes results in a different plan being chosen, because relative estimates between different candidate plans have changed.

Why can this do-nothing predicate sometimes improve query performance? The addition of the “no-op” coalesce() predicate introduces an error that masks something else that is preventing optimal performance.

What some performance enhancement tools do is a brute-force test: the tool repeatedly introduces the predicate into different places in a query, operating on different columns, to try to find a case where, by introducing an error, it stumbles onto a better-performing plan. This is also true of a query developer hand-coding the "no-op" predicate into a query. Typically, the developer will have some insight on the data to guide the placement of the predicate.

Using this method to improve query performance is a short-term solution that does not address root cause and might have the following implications:

- Potential areas for performance improvements are hidden.
- There are no guarantees that this workaround will provide permanent performance improvements, because the Db2 query compiler might eventually handle the predicate better, or other random factors might affect it.
- There might be other queries that are affected by the same root cause and the performance of your system in general might suffer as a result.

If you have followed best practices recommendations, but you believe that you are still getting less than optimal performance, you can provide explicit optimization guidelines to the Db2 optimizer, rather than introducing a “no-op” predicate. See “Optimization profiles and guidelines”.

Avoiding non-equality join predicates:

Join predicates that use comparison operators other than equality should be avoided because the join method is limited to nested loop.

Additionally, the optimizer might not be able to compute an accurate selectivity estimate for the join predicate. However, non-equality join predicates cannot always be avoided. When they are necessary, ensure that an appropriate index exists on either table, because the join predicates will be applied to the inner table of the nested-loop join.

One common example of non-equality join predicates is the case in which dimension data in a star schema must be versioned to accurately reflect the state of a dimension at different points in time. This is often referred to as a *slowly changing dimension*. One type of slowly changing dimension involves including effective start and end dates for each dimension row. A join between the fact table and the dimension table requires checking that a date associated with the fact falls within

the dimension's start and end date, in addition to joining on the dimension primary key. This is often referred to as a *type 6 slowly changing dimension*. The range join back to the fact table to further qualify the dimension version by some fact transaction date can be expensive. For example:

```
SELECT...
  FROM PRODUCT P, SALES F
  WHERE
    P.PROD_KEY = F.PROD_KEY AND
    F.SALE_DATE BETWEEN P.START_DATE AND
    P.END_DATE
```

In this situation, ensure that there is an index on (SALES.PROD_KEY, SALES.SALE_DATE).

Consider creating a statistical view to help the optimizer compute a better selectivity estimate for this scenario. For example:

```
CREATE VIEW V_PROD_FACT AS
  SELECT P.*
    FROM PRODUCT P, SALES F
    WHERE
      P.PROD_KEY = F.PROD_KEY AND
      F.SALE_DATE BETWEEN P.START_DATE AND
      P.END_DATE

ALTER VIEW V_PROD_FACT ENABLE QUERY OPTIMIZATION

RUNSTATS ON TABLE DB2USER.V_PROD_FACT WITH DISTRIBUTION
```

Specialized star schema joins, such as star join with index ANDing and hub joins, are not considered if there are any non-equality join predicates in the query block. (See “Ensuring that queries fit the required criteria for the star schema join”.)

Avoiding unnecessary outer joins:

The semantics of certain queries require outer joins (either left, right, or full). However, if the query semantics do not require an outer join, and the query is being used to deal with inconsistent data, it is best to deal with the inconsistent data problems at their root cause.

For example, in a data mart with a star schema, the fact table might contain rows for transactions but no matching parent dimension rows for some dimensions, due to data consistency problems. This could occur because the extract, transform, and load (ETL) process could not reconcile some business keys for some reason. In this scenario, the fact table rows are left outer joined with the dimensions to ensure that they are returned, even when they do not have a parent. For example:

```
SELECT...
  FROM DAILY_SALES F
    LEFT OUTER JOIN CUSTOMER C ON F.CUST_KEY = C.CUST_KEY
    LEFT OUTER JOIN STORE S ON F.STORE_KEY = S.STORE_KEY
  WHERE
    C.CUST_NAME = 'SMITH'
```

The left outer join can prevent a number of optimizations, including the use of specialized star-schema join access methods. However, in some cases the left outer join can be automatically rewritten to an inner join by the query optimizer. In this example, the left outer join between CUSTOMER and DAILY_SALES can be converted to an inner join because the predicate `C.CUST_NAME = 'SMITH'` will remove any rows with null values in this column, making a left outer join semantically unnecessary. So the loss of some optimizations due to the presence of

outer joins might not adversely affect all queries. However, it is important to be aware of these limitations and to avoid outer joins unless they are absolutely required.

Using the OPTIMIZE FOR N ROWS clause with the FETCH FIRST N ROWS ONLY clause:

The OPTIMIZE FOR *n* ROWS clause indicates to the optimizer that the application intends to retrieve only *n* rows, but the query will return the complete result set. The FETCH FIRST *n* ROWS ONLY clause indicates that the query should return only *n* rows.

The Db2 data server does not automatically assume OPTIMIZE FOR *n* ROWS when FETCH FIRST *n* ROWS ONLY is specified for the outer subselect. Try specifying OPTIMIZE FOR *n* ROWS along with FETCH FIRST *n* ROWS ONLY, to encourage query access plans that return rows directly from the referenced tables, without first performing a buffering operation such as inserting into a temporary table, sorting, or inserting into a hash join hash table.

Applications that specify OPTIMIZE FOR *n* ROWS to encourage query access plans that avoid buffering operations, yet retrieve the entire result set, might experience poor performance. This is because the query access plan that returns the first *n* rows fastest might not be the best query access plan if the entire result set is being retrieved.

Ensuring that queries fit the required criteria for the star schema join:

The optimizer considers three specialized join methods for queries based on star schema: star join, Cartesian hub join, and zigzag join. These join methods can help to significantly improve performance for such queries.

A query must meet the following criteria to be recognized as a star schema for the purposes of a zigzag join, star join, or Cartesian hub join plan.

- It must be a star-shaped query with one fact table and at least two dimension tables. If the query includes more than one fact table with common associated dimension tables (a multiple fact table query), the query optimizer will split the query into a query with multiple stars in it. The common dimension tables that join with more than one fact table are then used multiple times in the query. The explain output will show multiple zigzag join operators for these multiple fact table queries.
- The dimension tables must have a primary key, a unique constraint, or a unique index defined on them; the primary key can be a composite key. If the dimension tables do not have a primary key, a unique constraint, or a unique index, then an older star detection method is used to detect a star for Cartesian hub and star join methods. In that case, the Cartesian hub and star join must meet the criteria described in "Alternative Cartesian hub join and star join criteria" on page 175.
- The dimension tables and the fact table must be joined using equijoin predicates on all columns that are part of the primary keys for the dimension tables.
- For Cartesian hub joins and zigzag joins, there must be a multicolumn index on the fact table; columns that participate in the join are part of that index, which must have enough join columns from the fact table that at least two dimension tables are covered.
- For Cartesian hub joins and star index ANDing joins, a dimension table or a snowflake must filter the fact table. (Filtering is based on the optimizer's

estimates.) There are also cases where a star join will still occur if the dimension table is joined with the fact table not as a filter, but as a simple look-up type of join.

For example, suppose that there are three dimension tables D1, D2, and D3. Dimension table D1 has primary key A, and it joins with the fact table on column A; dimension table D2 has primary key (B,C), and it joins with the fact table on columns B and C; finally, dimension table D3 has primary key D, and it joins with the fact table on column D. Supported index usage is as follows:

- Any one of the following indexes would suffice, because each of these indexes covers at least two dimension tables: (A,D), (A,B,C), or (C,B,D).
- Index (A,B,C,D) is also suitable, because it covers three dimension tables.
- Index (A,B) cannot be used, because it does not completely cover dimension table D2.
- Index (B,A,C) cannot be used, because columns B and C, which join with the primary key of D2, do not appear in contiguous positions in the index.

A dimension table cannot participate in any of these joins methods (zigzag join, star join, or Cartesian hub join) if any of the following occurs:

- Sampling
- Non-deterministic functions
- Functions with side effects

Additionally, a zigzag join within a snowflake that causes duplication of keys from the dimension table is not supported.

Dimension tables that do not have these characteristics will be considered for inclusion in a zigzag join or star join by the optimizer using cost analysis; dimension tables that do have one or more of these characteristics can be joined using other join methods.

Nested zigzag joins are not supported; this means that a zigzag join plan may not be a dimension/snowflake to another zigzag join plan. Similarly, nested star joins are not supported; this means that a star join plan may not be a dimension/snowflake to another star join plan. In addition, zigzag join plans and star join plans cannot be nested inside one another.

Federated nicknames are excluded from the zigzag joins and star joins. Nicknames can be joined using other join methods. If the entire star is pushed down to a remote Db2 data server, the remote data server can use the zigzag join or star join method to execute the query.

Alternative Cartesian hub join and star join criteria

When there is no primary key, unique constraint, or unique index defined on the dimension tables then to be recognized as a star schema for the purposes of a Cartesian hub join or star join, a query must meet the following criteria:

- For each query block
  - At least three different tables must be joined
  - All join predicates must be equality predicates
  - No subqueries can exist
  - No correlations or dependencies can exist between tables or outside of the query block

- A fact table
  - Is the largest table in the query block
  - Has at least 10 000 rows
  - Is a base table
  - Must be joined to at least two dimension tables or two groups called snowflakes
- A dimension table
  - Is not the fact table
  - Can be joined individually to the fact table or in snowflakes
- A dimension table or a snowflake
  - Must filter the fact table. (Filtering is based on the optimizer's estimates.)
  - Must have a join predicate to the fact table that uses a column in a fact table index. This criterion must be met in order for either star join or hub join to be considered, although a hub join will only need to use a single fact table index.

A query block representing a left or right outer join can reference only two tables, so a star-schema join does not qualify.

Explicitly declaring referential integrity is not required for the optimizer to recognize a star-schema join.

Ensuring that queries fit the required criteria for the zigzag join:

The query optimizer uses the zigzag join method if: the tables and query fit the prerequisites for zigzag join, and a performance improvement is the result.

About this task

Use this task to ensure that your tables and query meet the prerequisites that are required for a zigzag join. In addition, you can use this task to manipulate the use of zigzag join, if you are not satisfied with the query optimizer choices.

The zigzag join method calculates the Cartesian product of rows from the dimension tables without actually materializing the Cartesian product. This method probes the fact table with a multicolumn index so that the fact table is filtered along two or more dimension tables simultaneously. The probe into the fact table finds matching rows. The zigzag join then returns the next combination of values that is available from the fact table index. This next combination of values, which is known as feedback, is used to skip over probe values that are provided by the Cartesian product of dimension tables that do not find a match in the fact table. Filtering the fact table on two or more dimension tables simultaneously, and skipping probes that are known to be unproductive, makes the zigzag join an efficient method for querying large fact tables.

Procedure

1. Ensure that the tables included in the zigzag join fit the required criteria. Each dimension tables must have the following properties: a primary key, a unique constraint, or a unique index that does not have a random ordering of index keys that are defined on it. To define primary keys, unique constraints, or unique indexes, use commands such as the following example:

```
--  defining a unique constraint on a dimension table using composite keys
create table dim1 (
  d0 int not null,
  d1 int not null,
  c1 int,
  constraint pk1_uniq unique (d0, d1)
);

-- defining a primary key on a dimension table
create table dim2 (
  d2 int primary key not null,
  c2 int
);

-- creating a unique index on a dimension table
create table dim3 (
  d3 int,
  c3 int
);
create unique index uniq_ind on dim3(d3);
```

2. Write a suitable query. The query must have equality join predicates between each dimension tables primary key, unique index, or unique constraint and the fact table columns. For example:

```
select count(*)
  from dim1,dim2,dim3,fact
  where dim1.d0 = fact.f0
    and dim1.d1 = fact.f1
    and dim2.d2 = fact.f2
    and dim3.d3 = fact.f3
    and dim1.c1 = 10
    and dim2.c2 < 20;
```

3. Ensure that there is a suitable multicolumn index on the fact table. The multicolumn index must include: columns that are used in the zigzag query in equality join predicates between the fact table and primary keys, unique indexes, or unique constraints from at least two of the dimension tables. To define such a multicolumn index, use a command such as the following example:

```
create index fact_ind on fact (f0, f1, f2, f3);
```

   If no suitable multicolumn index exists, an informational diagnostic message is displayed in the output of the **db2exfmt** command.

4. Run the query in EXPLAIN mode and then issue the **db2exfmt** command to format the EXPLAIN output. Examine the output to determine whether the zigzag join was used and whether the wanted performance was achieved.
5. Optional: If the zigzag join method was not used or if the wanted performance was not achieved, you might want to create another multicolumn index. Review the "Extended diagnostic information" section of **db2exfmt** command output. If an error message is listed in the output, follow the suggestions (to generate a new index, for instance).
6. Optional: If the wanted performance was not achieved, determine whether there was a gap in the index. Review the gap information (Gap Info) section in the **db2exfmt** output.

```
Gap Info:            Status
  ---------            ------
  Index Column 0:      No Gap
  Index Column 1:      Positioning Gap
  Index Column 2:      No Gap
  Index Column 3:      No Gap
  Index Column 4:      No Gap
```

If the section indicates that the query contains predicates that are inconsistent with a composite index, consider a new index or modifying an existing index to avoid the index gap.

Zigzag join access plan examples:

The following examples show the **db2exfmt** command output for different access plans possible with a zigzag join.

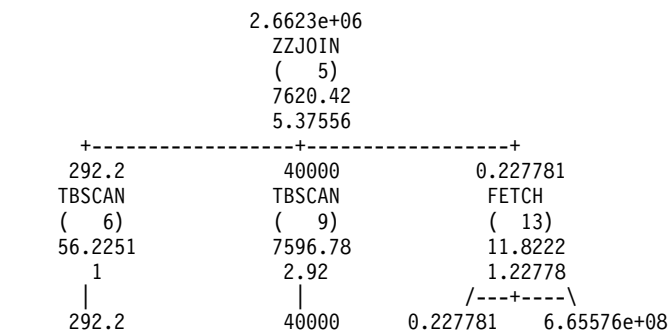
These examples use a star-shaped query with DAILY_SALES as the fact table, and CUSTOMER and PERIOD as the dimension tables. The query asks for the total quantity of the products sold in the month of March 1996 to the customers in age-level 7, such that the results are shown aggregated by the income level description of the customer.

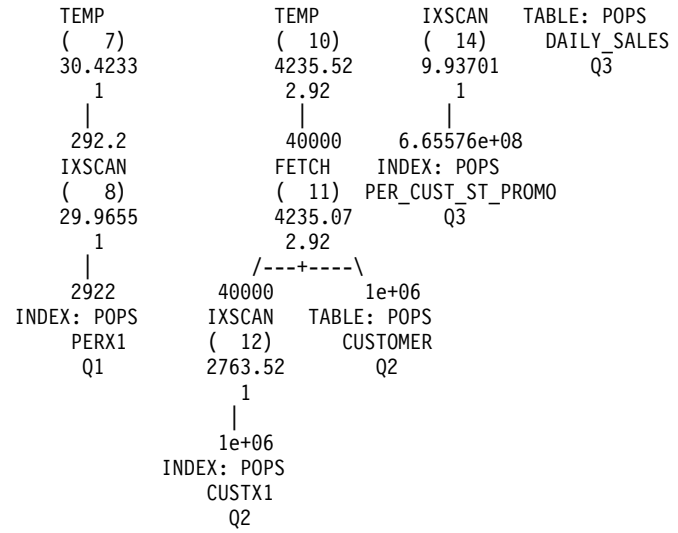
```
select income_level_desc, sum(quantity_sold) "Quantity"
from daily_sales s, customer c, period p
where calendar_date between '1996-03-01' and '1996-03-31'
and p.perkey = s.perkey
and s.custkey = c.custkey
and age_level = 7
group by income_level_desc;
```

Three types of fact table access plans are possible with a zigzag join.

- An index scan-fetch plan: In this plan, the index scan accesses the index over the fact table to retrieve RIDs from the fact table matching the input probe values. These fact table RIDs are then used to fetch the necessary fact table data from the fact table. Any dimension table payload columns are then retrieved from the dimension table and the result row is output by the zigzag join operator.
- A single probe list-prefetch plan: In this plan, a list prefetch plan is executed for every probe row from the combination of dimension tables and snowflakes. The index scan over the fact table finds fact table RIDs matching the input probe values. The SORT, RIDSCAN, and FETCH operators sort RIDs according to data page identifiers and list prefetchers start to get the fact table data. Any dimension table payload columns are then retrieved from the dimension tables and the result row is output by the zigzag join operator.
- An all-probes list-prefetch plan: In this plan, the index scan accesses the fact table index for all the probes from the combination of dimension tables and snowflakes. All such matching RIDs are sorted together in the order of fact table data pages and the list prefetchers start to retrieve the necessary data from the fact table. This method of sorting all RIDs helps achieve better prefetching. These queries will include two separate ZZJOIN operators, one of which represents a back-join between the fact table and the dimension tables.

Example: Index scan-fetch access of the fact table





The TBSCAN(6) and TBSCAN(9) operators show the following information:

```
IS_TEMP_INDEX : True/False
The scan builds an index over the temp for random access of the temp.
     (If the flag is 'true')
The scan builds a fast integer sort structure for random access of the temp.
     (If the flag is 'false')
```

The TBSCAN(6) and TBSCAN(9) operators show the information regarding the feedback predicates applied to the operators, in the form of start-stop key conditions.

```
Predicates:
  ----------
  5) Start Key Predicate,
   Comparison Operator:   Equal (=)
   Subquery Input Required:  No
   Filter Factor:    0.000342231

   Predicate Text:
   --------------
   (Q1.PERKEY = Q3.PERKEY)


  5) Stop Key Predicate,
   Comparison Operator:   Equal (=)
   Subquery Input Required:  No
   Filter Factor:    0.000342231

   Predicate Text:
   --------------
   (Q1.PERKEY = Q3.PERKEY)
```

The ZZJOIN(5) operator shows the collection of all the feedback predicates used in the processing of zigzag join.

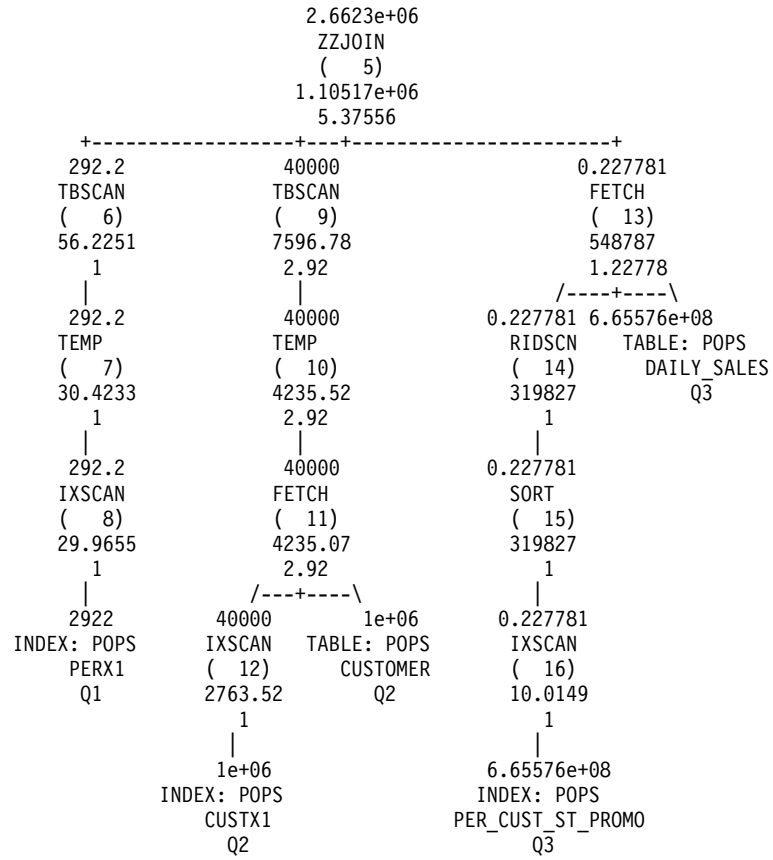
```
  Predicates:
  ----------
  4) Feedback Predicate used in Join,
   Comparison Operator:   Equal (=)
   Subquery Input Required:  No
   Filter Factor:    1e-06

   Predicate Text:
   --------------
   (Q3.CUSTKEY = Q2.CUSTKEY)
```

```
 5) Feedback Predicate used in Join,
  Comparison Operator:   Equal (=)
  Subquery Input Required:  No
  Filter Factor:    0.000342231

  Predicate Text:
  --------------
  (Q1.PERKEY = Q3.PERKEY)
```

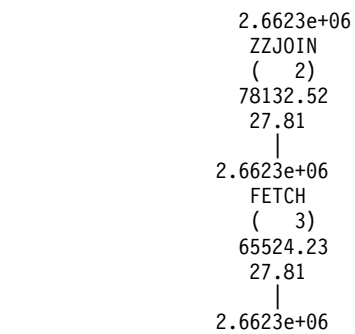
Example: Single probe list-prefetch access of the fact table

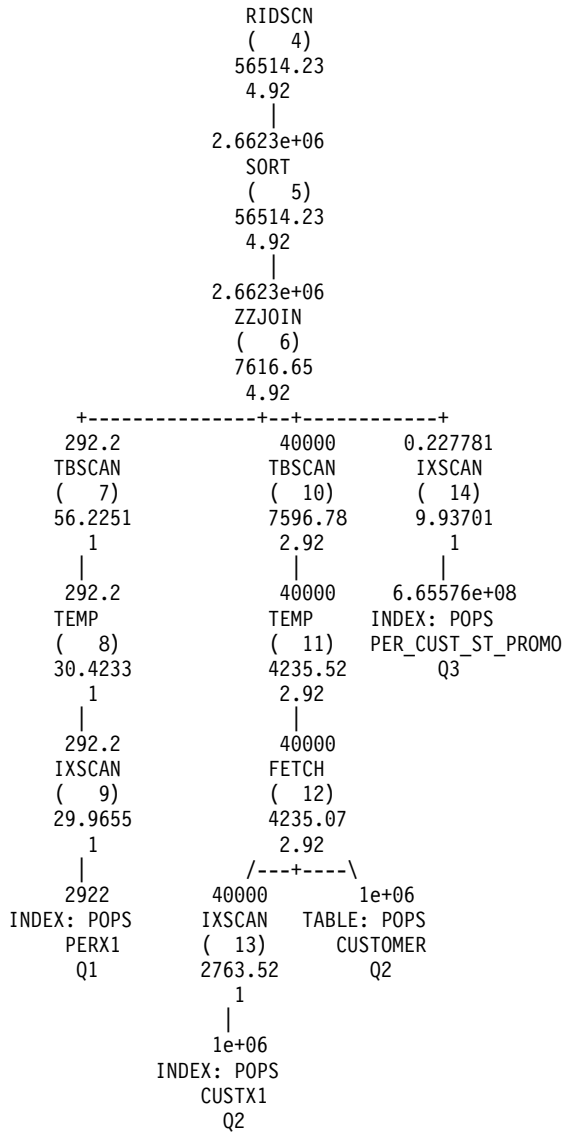


This shows that the difference between the index-scan plan and the single-probe plan is the way in which the fact table is accessed.

All other operators show the same information as the operators in the previous example.

Example: All probes list-prefetch access of the fact table





Compared to the other access plans, the all probes list-prefetch plan shows an additional operator, ZZJOIN (2). This operator is being used to perform back-joins of the fact table with the dimension tables. It shows the following information:

```
Backjoin = True
```

Zigzag joins with index gaps:

Even if there is a gap in the composite indexes, due to a missing join predicate or a missing dimension in the query, the query optimizer recognizes the star shape of a query and can select an appropriate access plan with a zigzag join. However, the performance of access plan may not be optimal. Review the **db2exfmt** command output to find out whether there are any index gaps in your query and consider adding new indexes or modifying existing indexes to improve performance.

Example: Zigzag join with a missing dimension

This example is based on the following query, where d1, d2, d3, d4, and d5 are dimensions and f1 is fact table.

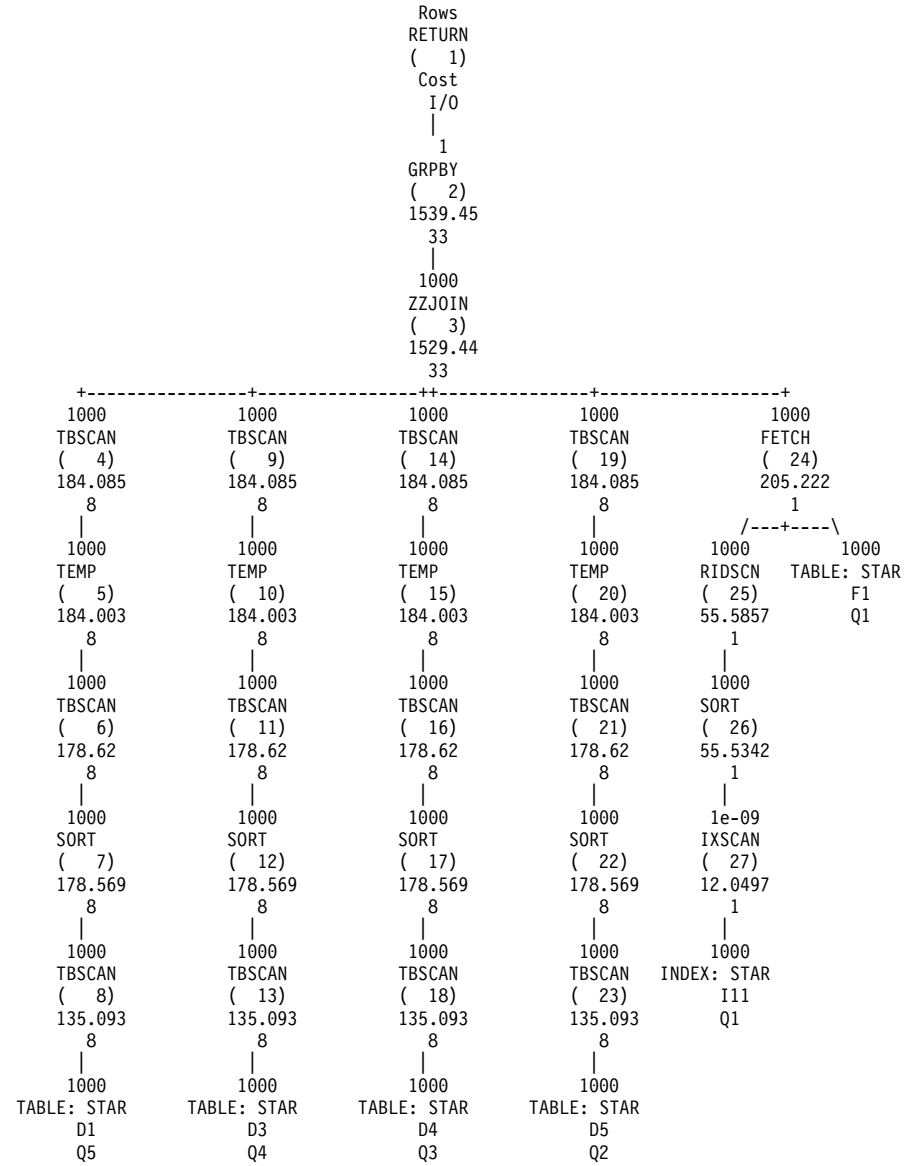
```
select count(*)
from d1, d3, d4, d5, f1
where d1.pk = f1.fk1 and d3.pk = f1.fk3 and d4.pk = f1.fk4 and d5.pk = f1.fk5
```

An index on fact table f1 was created using the following command:

```
create index i11 on f1(fk1, fk2, fk3, fk4, fk5, fk6);
```

The query joins dimensions d1, d3, d4, d5 with fact table f1. Because the dimension d2 is not included in the query, there is no join predicate with the dimension d2 on the column fk2. The query optimizer recognizes fact column fk2 as a gap in the index and is able to use the index for a zigzag join.

The **db2exfmt** command output shows that the index scan is a jump scan, by indicating the JUMPSCAN=TRUE option. The output also shows the index gap information, specifically that the second index column has a positioning gap and the other columns do not.



```
Gap Info:           Status
 ---------           ------
 Index Column 0:     No Gap
 Index Column 1:     Positioning Gap
 Index Column 2:     No Gap
 Index Column 3:     No Gap
 Index Column 4:     No Gap
```

Example: Zigzag join with missing join predicate on fact column

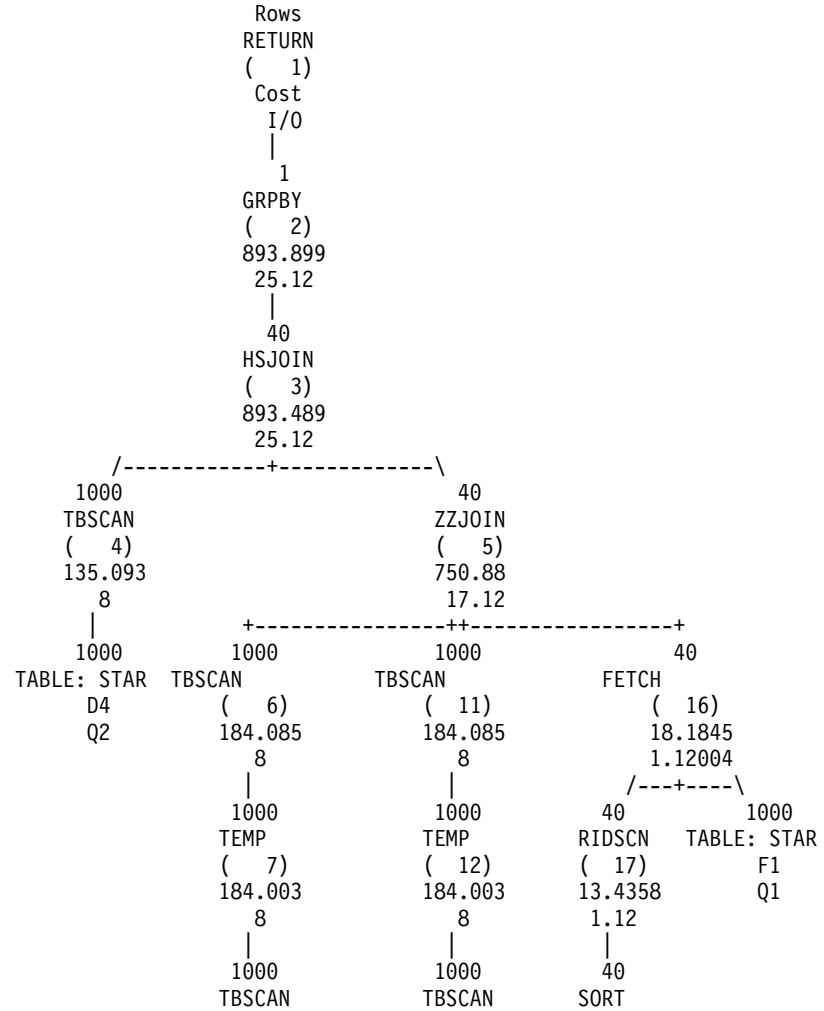
This example is based on the following query:

```
select count(*)
from d2, d3, d4, f1
where d2.pk = f1.fk2 and d3.pk = f1.fk3 and d4.pk = f1.fk4 and fk1=10
```

In this query, dimensions d2, d3 and d4 join with the fact table f1. There is no join predicate on fact column fk1, there is only a local predicate fk1=10.

The query optimizer recognizes the fact column fk1 as a gap, because there is no join predicate on it. The query optimizer is still able to use the index for zigzag join.

The **db2exfmt** command output shows that the index scan is a jump scan, by indicating the JUMPSCAN=TRUE option. The output also shows the index gap information, specifically that the first index column has a positioning gap and the other columns do not.



```
                    (   8)           (  13)       (  18)
                    178.62           178.62       13.3843
                       8                8           1.12
                       |                |            |
                     1000             1000         4e-05
                    SORT             SORT         IXSCAN
                    (   9)           (  14)       (  19)
                    178.569          178.569      12.5738
                       8                8           1.12
                       |                |            |
                     1000             1000         1000
                    TBSCAN           TBSCAN   INDEX: STAR
                    (  10)           (  15)         I11
                    135.093          135.093        Q1
                       8                8
                       |                |
                     1000             1000
                TABLE: STAR      TABLE: STAR
                     D2               D3
                     Q4               Q3

JUMPSCAN: (Jump Scan Plan)
 TRUE

Gap Info:            Status
---------            ------
Index Column 0:      Positioning Gap
Index Column 1:      No Gap
Index Column 2:      No Gap
```

Avoiding redundant predicates:

Avoid redundant predicates, especially when they occur across different tables. In some cases, the optimizer cannot detect that the predicates are redundant. This might result in cardinality underestimation.

For example, within SAP business intelligence (BI) applications, the snowflake schema with fact and dimension tables is used as a query optimized data structure. In some cases, there is a redundant time characteristic column ("SID_0CALMONTH" for month or "SID_0FISCPER" for year) defined on the fact and dimension tables.

The SAP BI online analytical processing (OLAP) processor generates redundant predicates on the time characteristics column of the dimension and fact tables.

These redundant predicates might result in longer query run time.

The following section provides an example with two redundant predicates that are defined in the WHERE condition of a SAP BI query. Identical predicates are defined on the time dimension (DT) and fact (F) table:

```
    AND (     "DT"."SID_0CALMONTH" = 199605
          AND "F". "SID_0CALMONTH" = 199605
           OR "DT"."SID_0CALMONTH" = 199705
          AND "F". "SID_0CALMONTH" = 199705 )
AND NOT (     "DT"."SID_0CALMONTH" = 199803
          AND "F". "SID_0CALMONTH" = 199803 )
```

The Db2 optimizer does not recognize the predicates as identical, and treats them as independent. This leads to underestimation of cardinalities, suboptimal query access plans, and longer query run times.

For that reason, the redundant predicates are removed by the Db2 database platform-specific software layer.

These predicates are transferred to the following ones and only the predicates on the fact table column "SID_0CALMONTH" remain:

```
    AND (     "F". "SID_0CALMONTH" = 199605
           OR "F". "SID_0CALMONTH" = 199705 )
AND NOT (     "F". "SID_0CALMONTH" = 199803 )
```

Apply the instructions in SAP notes 957070 and 1144883 to remove the redundant predicates.

Using constraints to improve query optimization:

Consider defining unique, check, and referential integrity constraints. These constraints provide semantic information that allows the Db2 optimizer to rewrite queries to eliminate joins, push aggregation down through joins, push FETCH FIRST *n* ROWS down through joins, remove unnecessary DISTINCT operations, and perform a number of other optimizations.

Informational constraints can also be used for both check constraints and referential integrity constraints when the application itself can guarantee the relationships. The same optimizations are possible. Constraints that are enforced by the database manager when rows are inserted, updated, or deleted can lead to high system overhead, especially when updating a large number of rows that have referential integrity constraints. If an application has already verified information before updating a row, it might be more efficient to use informational constraints, rather than regular constraints. This type of informational constraint is also known as a NOT ENFORCED TRUSTED constraint.

For example, consider two tables, DAILY_SALES and CUSTOMER. Each row in the CUSTOMER table has a unique customer key (CUST_KEY). DAILY_SALES contains a CUST_KEY column and each row references a customer key in the CUSTOMER table. A referential integrity constraint could be created to represent this 1:N relationship between CUSTOMER and DAILY_SALES. If the application were to enforce the relationship, the constraint could be defined as informational. The following query could then avoid performing the join between CUSTOMER and DAILY_SALES, because no columns are retrieved from CUSTOMER, and every row from DAILY_SALES will find a match in CUSTOMER. The query optimizer will automatically remove the join.

```
SELECT AMT_SOLD, SALE PRICE, PROD_DESC
  FROM DAILY_SALES, PRODUCT, CUSTOMER
  WHERE
    DAILY_SALES.PROD_KEY = PRODUCT.PRODKEY AND
    DAILY_SALES.CUST_KEY = CUSTOMER.CUST_KEY
```

Informational constraints must not be violated, otherwise queries might return incorrect results. In this example, if any rows in DAILY_SALES do not have a corresponding customer key in the CUSTOMER table, the query would incorrectly return those rows.

Another type of informational constraint is the NOT ENFORCED NOT TRUSTED constraint. It can be useful to specify this type of informational constraint if an application cannot verify that the rows of a table will conform to the constraint. The NOT ENFORCED NOT TRUSTED constraint can be used to improve query optimization in cases where the Db2 optimizer can use the data to infer statistics from a statistical view. In these cases the strict matching between the values in the

foreign keys and the primary keys are not needed. If a constraint is NOT TRUSTED and enabled for query optimization, then it will not be used to perform optimizations that depend on the data conforming completely to the constraint, such as join elimination.

When RI (referential integrity) tables are related by informational constraints, the informational constraints might be used in the incremental maintenance of dependant MQT data, staging tables, and query optimization. Violating an informational constraint might result in inaccurate MQT data and query results.

For example, parent and child tables are related by informational constraints, so the order in which they are maintained affects query results and MQT integrity. If there is data in the child table that cannot be related to a row in the parent table, an orphan row has been created. Orphan rows are a violation of the informational constraint relating that parent and child table. The dependent MQT data and staging tables associated with the parent-child tables might be updated with incorrect data, resulting in unpredictable optimization behaviour.

If you have an ENFORCED informational constraint, Db2 will force you to maintain RI tables in the correct order. For example, if you deleted a row in a parent table that would result in an orphan row, Db2 returns an SQL error and rolls back the change.

If you have a NOT ENFORCED informational constraint, you must maintain the integrity of the RI tables by updating tables in the correct order. The order in which parent-child tables are maintained is important to ensure MQT data integrity.

For example, you have set up the following parent and child table with a corresponding MQT:

```
create table parent (i1 int not null primary key, i2 int);
create table child (i1 int not null, i2 int);

alter table child add constraint fk1 foreign key (i2) references parent (i1) not enforced;
enable query optimization;

create table mqt1 as (select p.i1 as c1, p.i2 as c2, c.i1 as c3, count (*) as cnt from parent p, child c
where p.i1 = c.i2 group by p.i1, p.i2, c.i1) data
initially deferred refresh immediate;

refresh table mqt1;

commit;
```

To insert rows into parent-child tables, you must insert rows into the parent table first.

```
insert into parent values (4, 4);
insert into child values(40, 4);
```

If rows are inserted into the child table first, orphan rows exist while there is no row in the parent table that matches the child row's foreign key. This violation, although temporary, might produce unpredictable behaviour during query optimization and MQT processing.

To remove rows from the parent table, you must remove the related rows from the child table first.

```
delete from child;
delete from parent;
```

If rows are removed from the parent table first, orphan rows are created when a child row's foreign key no longer matches a row key in the parent table. This

results in a violation of the informational constraint between the parent and child tables. This violation, although temporary, might produce unpredictable behaviour during query optimization and MQT processing.

Using the REOPT bind option with input variables in complex queries:

Input variables are essential for good statement preparation times in an online transaction processing (OLTP) environment, where statements tend to be simpler and query access plan selection is more straightforward.

Multiple executions of the same query with different input variable values can reuse the compiled access section in the dynamic statement cache, avoiding expensive SQL statement compilations whenever the input values change.

However, input variables can cause problems for complex query workloads, where query access plan selection is more complex and the optimizer needs more information to make good decisions. Moreover, statement compilation time is usually a small component of total execution time, and business intelligence (BI) queries, which do not tend to be repeated, do not benefit from the dynamic statement cache.

If input variables need to be used in a complex query workload, consider using the REOPT(ALWAYS) bind option. The **REOPT** bind option defers statement compilation from PREPARE to OPEN or EXECUTE time, when the input variable values are known. The values are passed to the SQL compiler so that the optimizer can use the values to compute a more accurate selectivity estimate. REOPT(ALWAYS) specifies that the statement should be recompiled for every execution. REOPT(ALWAYS) can also be used for complex queries that reference special registers, such as `WHERE TRANS_DATE = CURRENT DATE - 30 DAYS`, for example. If input variables lead to poor access plan selection for OLTP workloads, and REOPT(ALWAYS) results in excessive overhead due to statement compilation, consider using REOPT(ONCE) for selected queries. REOPT(ONCE) defers statement compilation until the first input variable value is bound. The SQL statement is compiled and optimized using this first input variable value. Subsequent executions of the statement with different values reuse the access section that was compiled on the basis of the first input value. This can be a good approach if the first input variable value is representative of subsequent values, and it provides a better query access plan than one that is based on default values when the input variable values are unknown.

There a number of ways that **REOPT** can be specified:

- For embedded SQL in C/C++ applications, use the **REOPT** bind option. This bind option affects re-optimization behavior for both static and dynamic SQL.
- For CLP packages, rebind the CLP package with the **REOPT** bind option. For example, to rebind the CLP package used for isolation level CS with **REOPT** `ALWAYS`, specify the following command:

  ```
  rebind nullid.SQLC2G13 reopt always
  ```

- For CLI applications, set the **REOPT** value in one of the following ways:
  - Use the **REOPT** keyword setting in the `db2cli.ini` configuration file. The values and corresponding options are:
    - `2` = SQL_REOPT_NONE
    - `3` = SQL_REOPT_ONCE
    - `4` = SQL_REOPT_ALWAYS
  - Use the SQL_ATTR_REOPT connection or statement attribute.

  - Use the SQL_ATTR_CURRENT_PACKAGE_SET connection or statement attribute to specify either the NULLID, NULLIDR1, or NULLIDRA package sets. NULLIDR1 and NULLIDRA are reserved package set names. When used, **REOPT** ONCE or **REOPT** ALWAYS are implied, respectively. These package sets have to be explicitly created with the following commands:

    ```
    db2 bind db2clipk.bnd collection NULLIDR1
    db2 bind db2clipk.bnd collection NULLIDRA
    ```

- For JDBC applications that use the IBM Data Server Driver for JDBC and SQLJ, specify the **-reopt** value when you run the **DB2Binder** utility.
- For SQL PL procedures, use one of the following approaches:
  - Use the SET_ROUTINE_OPTS stored procedure to set the bind options that are to be used for the creation of SQL PL procedures within the current session. For example, call:

    ```
    sysproc.set_routine_opts('reopt always')
    ```

  - Use the **DB2_SQLROUTINE_PREPOPTS** registry variable to set the SQL PL procedure options at the instance level. Values set using the SET_ROUTINE_OPTS stored procedure will override those specified with **DB2_SQLROUTINE_PREPOPTS**.

You can also use optimization profiles to set **REOPT** for static and dynamic statements, as shown in the following example:

```
<STMTPROFILE ID="REOPT example ">
     <STMTKEY>
       <![CDATA[select acct_no from customer where name = ? ]]>
     </STMTKEY>
     <OPTGUIDELINES>
        <REOPT VALUE='ALWAYS'/>
    </OPTGUIDELINES>
</STMTPROFILE>
```

Using parameter markers to reduce compilation time for dynamic queries:

The Db2 data server can avoid recompiling a dynamic SQL statement that has been run previously by storing the access section and statement text in the dynamic statement cache.

A subsequent prepare request for this statement will attempt to find the access section in the dynamic statement cache, avoiding compilation. However, statements that differ only in the literals that are used in predicates will not match. For example, the following two statements are considered different in the dynamic statement cache:

```
SELECT AGE FROM EMPLOYEE WHERE EMP_ID = 26790
SELECT AGE FROM EMPLOYEE WHERE EMP_ID = 77543
```

Even relatively simple SQL statements can result in excessive system CPU usage due to statement compilation, if they are run very frequently. If your system experiences this type of performance problem, consider changing the application to use parameter markers to pass predicate values to the Db2 compiler, rather than explicitly including them in the SQL statement. However, the access plan might not be optimal for complex queries that use parameter markers in predicates. For more information, see “Using the REOPT bind option with input variables in complex queries”.

Setting the DB2_REDUCED_OPTIMIZATION registry variable:

If setting the optimization class does not reduce the compilation time sufficiently for your application, try setting the **DB2_REDUCED_OPTIMIZATION** registry variable.

This registry variable provides more control over the optimizer's search space than setting the optimization class. This registry variable lets you request either reduced optimization features or rigid use of optimization features at the specified optimization class. If you reduce the number of optimization techniques used, you also reduce time and resource use during optimization.

Although optimization time and resource use might be reduced, there is increased risk of producing a less than optimal query access plan.

First, try setting the registry variable to YES. If the optimization class is 5 (the default) or lower, the optimizer disables some optimization techniques that might consume significant prepare time and resources but that do not usually produce a better query access plan. If the optimization class is exactly 5, the optimizer reduces or disables some additional techniques, which might further reduce optimization time and resource use, but also further increase the risk of a less than optimal query access plan. For optimization classes lower than 5, some of these techniques might not be in effect in any case. If they are, however, they remain in effect.

If the YES setting does not provide a sufficient reduction in compilation time, try setting the registry variable to an integer value. The effect is the same as YES, with the following additional behavior for dynamically prepared queries optimized at class 5. If the total number of joins in any query block exceeds the setting, the optimizer switches to greedy join enumeration instead of disabling additional optimization techniques. The result is that the query will be optimized at a level that is similar to optimization class 2.

Improving insert performance

Before data is inserted into a table, an insert search algorithm examines the free space control records (FSCRs) to find a page with enough space for the new data.

However, even when an FSCR indicates that a page has enough free space, that space might not be usable if it has been reserved by an uncommitted delete operation from another transaction.

The **DB2MAXFSCRSEARCH** registry variable specifies the number of FSCRs to search when adding a record to a table. The default is to search five FSCRs. Modifying this value enables you to balance insert speed with space reuse. Use large values to optimize for space reuse. Use small values to optimize for insert speed. Setting the value to -1 forces the database manager to search all FSCRs. If sufficient space is not found while searching FSCRs, the data is appended to the end of the table.

In a pureScale environment, space that is freed up by a delete transaction on one member may not be reused by an insert transaction on a different member if no rows have been deleted on that member.

The APPEND ON option on the ALTER TABLE statement specifies that table data will be appended and that information about free space on pages will not be kept. Such tables must not have a clustering index. This option can improve performance for tables that only grow.

If a clustering index is defined on the table, the database manager attempts to insert records on the same page as other records with similar index key values. If there is no space on that page, surrounding pages are considered. If those pages are unsuitable, the FSCRs are searched, as described previously. In this case, however, a "worst-fit" approach is used instead of a "first-fit" approach. The worst-fit approach tends to choose pages with more free space. This method establishes a new clustering area for rows with similar key values.

If you have defined a clustering index on a table, use the PCTFREE clause on the ALTER TABLE statement before loading or reorganizing the table. The PCTFREE clause specifies the percentage of free space that should remain on a data page after a load or reorg operation. This increases the probability that the cluster index operation will find free space on the appropriate page.

Efficient SELECT statements

Because SQL is a flexible high-level language, you can write several different SELECT statements to retrieve the same data. However, performance might vary for different forms of the statement, as well as for different optimization classes.

Consider the following guidelines for creating efficient SELECT statements:

- Specify only columns that you need. Specifying all columns with an asterisk (*) results in unnecessary processing.
- Use predicates that restrict the answer set to only those rows that you need.
- When you need significantly fewer than the total number of rows that might be returned, specify the OPTIMIZE FOR clause. This clause affects both the choice of access plan and the number of rows that are blocked in the communication buffer.
- To take advantage of row blocking and improve performance, specify the FOR READ ONLY or FOR FETCH ONLY clause. Concurrency improves as well, because exclusive locks are never held on the rows that are retrieved. Additional query rewrites can also occur. Specifying these clauses, as well as the BLOCKING ALL bind option, can similarly improve the performance of queries running against nicknames in a federated database system.
- For cursors that will be used with positioned updates, specify the FOR UPDATE OF clause to enable the database manager to choose more appropriate locking levels initially and to avoid potential deadlocks. Note that FOR UPDATE cursors cannot take advantage of row blocking.
- For cursors that will be used with searched updates, specify the FOR READ ONLY and the USE AND KEEP UPDATE LOCKS clauses to avoid deadlocks and still allow row blocking by forcing U locks on affected rows.
- Avoid numeric data type conversions whenever possible. When comparing values, try to use items that have the same data type. If conversions are necessary, inaccuracies due to limited precision, and performance costs due to runtime conversions might result.

  If possible, use the following data types:

  – Character instead of varying character for short columns
  – Integer instead of float, decimal, or DECFLOAT
  – DECFLOAT instead of decimal
  – Datetime instead of character
  – Numeric instead of character
- To decrease the probability that a sort operation will occur, omit clauses such as DISTINCT or ORDER BY if such operations are not required.

- To check for the existence of rows in a table, select a single row. Either open a cursor and fetch one row, or perform a single-row SELECT INTO operation. Remember to check for the SQLCODE -811 error if more than one row is found.

  Unless you know that the table is very small, do not use the following statement to check for a non-zero value:

  ```
  select count(*) from <table-name>
  ```

  For large tables, counting all the rows impacts performance.
- If update activity is low and tables are large, define indexes on columns that are frequently used in predicates.
- Consider using an IN list if the same column appears in multiple predicates. For large IN lists that are used with host variables, looping a subset of the host variables might improve performance.

The following suggestions apply specifically to SELECT statements that access several tables.

- Use join predicates to join tables. A *join predicate* is a comparison between two columns from different tables in a join.
- Define indexes on the columns in a join predicate to enable the join to be processed more efficiently. Indexes also benefit UPDATE and DELETE statements containing SELECT statements that access several tables.
- If possible, avoid using OR clauses or expressions with join predicates.
- In a partitioned database environment, it is recommended that tables being joined are partitioned on the join column.

Guidelines for restricting SELECT statements

The optimizer assumes that an application must retrieve all of the rows that are identified by a SELECT statement. This assumption is most appropriate in online transaction processing (OLTP) and batch environments.

However, in "browse" applications, queries often define a large potential answer set, but they retrieve only the first few rows, usually the number of rows that are required for a particular display format.

To improve performance for such applications, you can modify the SELECT statement in the following ways:

- Use the FOR UPDATE clause to specify the columns that could be updated by a subsequent positioned UPDATE statement.
- Use the FOR READ or FETCH ONLY clause to make the returned columns read-only.
- Use the OPTIMIZE FOR *n* ROWS clause to give priority to retrieving the first *n* rows from the full result set.
- Use the FETCH FIRST *n* ROWS ONLY clause to retrieve only a specified number of rows.
- Use the DECLARE CURSOR WITH HOLD statement to retrieve rows one at a time.

The following sections describe the performance advantages of each method.

FOR UPDATE clause

The FOR UPDATE clause limits the result set by including only those columns that can be updated by a subsequent positioned UPDATE statement. If you specify the

FOR UPDATE clause without column names, all columns that can be updated in the table or view are included. If you specify column names, each name must be unqualified and must identify a column of the table or view.

You cannot use the FOR UPDATE clause if:

- The cursor that is associated with the SELECT statement cannot be deleted
- At least one of the selected columns is a column that cannot be updated in a catalog table and that has not been excluded in the FOR UPDATE clause.

In CLI applications, you can use the CLI connection attribute SQL_ATTR_ACCESS_MODE for the same purpose.

FOR READ or FETCH ONLY clause

The FOR READ ONLY clause or the FOR FETCH ONLY clause ensures that read-only results are returned. For result tables where updates and deletions are allowed, specifying the FOR READ ONLY clause can improve the performance of fetch operations if the database manager can retrieve blocks of data instead of using exclusive locks. Do not specify the FOR READ ONLY clause in queries that are used in positioned UPDATE or DELETE statements.

In CLI applications, you can use the CLI connection attribute SQL_ATTR_ACCESS_MODE for the same purpose.

OPTIMIZE FOR *n* ROWS clause

The OPTIMIZE FOR clause declares the intent to retrieve only a subset of the result or to give priority to retrieving only the first few rows. The optimizer can then choose access plans that minimize the response time for retrieving the first few rows. In addition, the number of rows that are sent to the client as a single block are limited by the value of *n*. Thus the OPTIMIZE FOR clause affects how the server retrieves qualifying rows from the database, and how it returns those rows to the client.

For example, suppose you regularly query the EMPLOYEE table to determine which employees have the highest salary:

```
select lastname, firstnme, empno, salary
  from employee
  order by salary desc
```

Although you have previously defined a descending index on the SALARY column, this index is likely to be poorly clustered, because employees are ordered by employee number. To avoid many random synchronous I/Os, the optimizer would probably choose the list prefetch access method, which requires sorting the row identifiers of all rows that qualify. This sort causes a delay before the first qualifying rows can be returned to the application. To prevent this delay, add the OPTIMIZE FOR clause to the statement as follows:

```
select lastname, firstnme, empno, salary
  from employee
  order by salary desc
  optimize for 20 rows
```

In this case, the optimizer will likely choose to use the SALARY index directly, because only the 20 employees with the highest salaries are retrieved. Regardless of how many rows might be blocked, a block of rows is returned to the client every twenty rows.

With the OPTIMIZE FOR clause, the optimizer favors access plans that avoid bulk operations or flow interruptions, such as those that are caused by sort operations. You are most likely to influence an access path by using the OPTIMIZE FOR 1 ROW clause. Using this clause might have the following effects:

- Join sequences with composite inner tables are less likely, because they require a temporary table.
- The join method might change. A nested loop join is the most likely choice, because it has low overhead cost and is usually more efficient when retrieving a few rows.
- An index that matches the ORDER BY clause is more likely, because no sort is required for the ORDER BY.
- List prefetching is less likely, because this access method requires a sort.
- Sequential prefetching is less likely, because only a small number of rows is required.
- In a join query, the table with columns in the ORDER BY clause is likely to be chosen as the outer table if an index on the outer table provides the ordering that is needed for the ORDER BY clause.

Although the OPTIMIZE FOR clause applies to all optimization levels, it works best for optimization class 3 and higher, because classes lower than 3 use the *greedy join enumeration* search strategy. This method sometimes results in access plans for multi-table joins that do not lend themselves to quick retrieval of the first few rows.

If a packaged application uses the call-level interface (CLI or ODBC), you can use the **OPTIMIZEFORNROWS** keyword in the `db2cli.ini` configuration file to have CLI automatically append an OPTIMIZE FOR clause to the end of each query statement.

When data is selected from nicknames, results can vary depending on data source support. If the data source that is referenced by a nickname supports the OPTIMIZE FOR clause, and the Db2 optimizer pushes the entire query down to the data source, then the clause is generated in the remote SQL that is sent to the data source. If the data source does not support this clause, or if the optimizer decides that the least costly plan is local execution, the OPTIMIZE FOR clause is applied locally. In this case, the Db2 optimizer prefers access plans that minimize the response time for retrieving the first few rows of a query, but the options that are available to the optimizer for generating plans are slightly limited, and performance gains from the OPTIMIZE FOR clause might be negligible.

If the OPTIMIZE FOR clause and the FETCH FIRST clause are both specified, the lower of the two *n* values affects the communications buffer size. The two values are considered independent of each other for optimization purposes.

FETCH FIRST *n* ROWS ONLY clause

The FETCH FIRST *n* ROWS ONLY clause sets the maximum number of rows that can be retrieved. Limiting the result table to the first several rows can improve performance. Only *n* rows are retrieved, regardless of the number of rows that the result set might otherwise contain.

If the FETCH FIRST clause and the OPTIMIZE FOR clause are both specified, the lower of the two *n* values affects the communications buffer size. The two values are considered independent of each other for optimization purposes.

DECLARE CURSOR WITH HOLD statement

When you declare a cursor using a DECLARE CURSOR statement that includes the WITH HOLD clause, open cursors remain open when the transaction commits, and all locks are released, except those locks that protect the current cursor position. If the transaction is rolled back, all open cursors are closed, all locks are released, and any LOB locators are freed.

In CLI applications, you can use the CLI connection attribute SQL_ATTR_CURSOR_HOLD for the same purpose. If a packaged application uses the call level interface (CLI or ODBC), use the **CURSORHOLD** keyword in the `db2cli.ini` configuration file to have CLI automatically assume the WITH HOLD clause for every declared cursor.

Specifying row blocking to reduce overhead

Row blocking, which is supported for all statements and data types (including LOB data types), reduces database manager overhead for cursors by retrieving a block of rows in a single operation.

About this task

This block of rows represents a number of pages in memory. It is not a multidimensional clustering (MDC) or insert time clustering (ITC) table block, which is physically mapped to an extent on disk.

Row blocking is specified by the following options on the **BIND** or **PREP** command:

BLOCKING ALL
Cursors that are declared with the FOR READ ONLY clause or that are not specified as FOR UPDATE will be blocked.

BLOCKING NO
Cursors will not be blocked.

BLOCKING UNAMBIG
Cursors that are declared with the FOR READ ONLY clause will be blocked. Cursors that are not declared with the FOR READ ONLY clause or the FOR UPDATE clause, that are not ambiguous, or that are read-only, will be blocked. Ambiguous cursors will not be blocked.

The following database manager configuration parameters are used during block-size calculations.

- The **aslheapsz** parameter specifies the size of the application support layer heap for local applications. It is used to determine the I/O block size when a blocking cursor is opened.
- The **rqrioblk** parameter specifies the size of the communication buffer between remote applications and their database agents on the database server. It is also used to determine the I/O block size at the data server runtime client when a blocking cursor is opened.

Before enabling the blocking of row data for LOB data types, it is important to understand the impact on system resources. More shared memory will be consumed on the server to store the references to LOB values in each block of data when LOB columns are returned. The number of such references will vary according to the value of the **rqrioblk** configuration parameter.

To increase the amount of memory allocated to the heap, modify the **database_memory** database configuration parameter by:

- Setting its value to AUTOMATIC
- Increasing its value by 256 pages if the parameter is currently set to a user-defined numeric value

To increase the performance of an existing embedded SQL application that references LOB values, rebind the application using the **BIND** command and specifying either the **BLOCKING ALL** clause or the **BLOCKING UNAMBIG** clause to request blocking. Embedded applications will retrieve the LOB values, one row at a time, after a block of rows has been retrieved from the server. User-defined functions (UDFs) returning LOB results might cause the Db2 server to revert to single-row retrieval of LOB data when large amounts of memory are being consumed on the server.

Procedure

To specify row blocking:

1. Use the values of the **aslheapsz** and **rqrioblk** configuration parameters to estimate how many rows are returned for each block. In both formulas, *orl* is the output row length, in bytes.
   - Use the following formula for local applications:

     ```
     Rows per block = aslheapsz * 4096 / orl
     ```

     The number of bytes per page is 4096.
   - Use the following formula for remote applications:

     ```
     Rows per block = rqrioblk / orl
     ```
2. To enable row blocking, specify an appropriate value for the **BLOCKING** option on the **BIND** or **PREP** command.

   If you do not specify the **BLOCKING** option, the default row blocking type is UNAMBIG. For the command line processor (CLP) and the call-level interface (CLI), the default row blocking type is ALL.

Data sampling in queries

It is often impractical and sometimes unnecessary to access all of the data that is relevant to a query. In some cases, finding overall trends or patterns in a subset of the data will suffice. One way to do this is to run a query against a random sample from the database.

The Db2 product enables you to efficiently sample data for SQL and XQuery queries, potentially improving the performance of large queries by orders of magnitude, while maintaining a high degree of accuracy.

Sampling is commonly used for aggregate queries, such as AVG, COUNT, and SUM, where reasonably accurate values for the aggregates can be obtained from a sample of the data. Sampling can also be used to obtain a random subset of the rows in a table for auditing purposes or to speed up data mining and analysis.

Two methods of sampling are available: row-level sampling and page-level sampling.

Row-level Bernoulli sampling

Row-level Bernoulli sampling obtains a sample of *P* percent of the table rows by means of a SARGable predicate that includes each row in the sample with a probability of *P*/100 and excludes it with a probability of 1-*P*/100.

Row-level Bernoulli sampling always produces a valid, random sample regardless of the degree of data clustering. However, the performance of this type of sampling is very poor if no index is available, because every row must be retrieved and the sampling predicate must be applied to it. If there is no index, there are no I/O savings over executing the query without sampling. If an index is available, performance is improved, because the sampling predicate is applied to the RIDS inside of the index leaf pages. In the usual case, this requires one I/O per selected RID, and one I/O per index leaf page.

System page-level sampling

System page-level sampling is similar to row-level sampling, except that pages (not rows) are sampled. The probability of a page being included in the sample is *P*/100. If a page is included, all of the rows on that page are included.

The performance of system page-level sampling is excellent, because only one I/O is required for each page that is included in the sample. Compared with no sampling, page-level sampling improves performance by orders of magnitude. However, the accuracy of aggregate estimates tends to be worse under page-level sampling than row-level sampling. This difference is most pronounced when there are many rows per page, or when the columns that are referenced in the query exhibit a high degree of clustering within pages.

Specifying the sampling method

Use the TABLESAMPLE clause to execute a query against a random sample of data from a table. TABLESAMPLE BERNOULLI specifies that row-level Bernoulli sampling is to be performed. TABLESAMPLE SYSTEM specifies that system page-level sampling is to be performed, unless the optimizer determines that it is more efficient to perform row-level Bernoulli sampling instead.

Parallel processing for applications

The Db2 product supports parallel environments, specifically on symmetric multiprocessor (SMP) machines.

In SMP machines, more than one processor can access the database, allowing the execution of complex SQL requests to be divided among the processors. This *intrapartition parallelism* is the subdivision of a single database operation (for example, index creation) into multiple parts, which are then executed in parallel within a single database partition.

To specify the degree of parallelism when you compile an application, use the CURRENT DEGREE special register, or the DEGREE bind option. *Degree* refers to the number of query parts that can execute concurrently. There is no strict relationship between the number of processors and the value that you select for the degree of parallelism. You can specify a value that is more or less than the number of processors on the machine. Even for uniprocessor machines, you can set the degree to be higher than one to improve performance. Note, however, that each degree of parallelism adds to the system memory and processor overhead.

You can also specify the degree of parallelism for workloads using the MAXIMUM DEGREE workload attribute. In the affected workload, values set using MAXIMUM DEGREE will override values assigned by the CURRENT DEGREE special register, or the DEGREE bind option.

Some configuration parameters must be modified to optimize performance when you use parallel execution of queries. In an environment with a high degree of parallelism, you should review and modify configuration parameters that control the amount of shared memory and prefetching.

The following configuration parameters control and manage parallel processing.

- The **intra_parallel** database manager configuration parameter enables or disables parallelism.
- The **max_querydegree** database manager configuration parameter sets an upper limit on the degree of parallelism for any query in the database. This value overrides the CURRENT DEGREE special register and the DEGREE bind option.
- The **dft_degree** database configuration parameter sets the default value for the CURRENT DEGREE special register and the DEGREE bind option.

To enable or disable intrapartition parallelism from within a database application, you can call the ADMIN_SET_INTRA_PARALLEL procedure. Setting ADMIN_SET_INTRA_PARALLEL will apply intrapartition parallelism only to your application. For your application, this value will override the **intra_parallel** database manager configuration parameter.

To enable or disable intrapartition parallelism from within a workload, you can set the MAXIMUM DEGREE workload attribute. This will apply intrapartition parallelism only to your workload. This value will override both the the **intra_parallel** database manager configuration parameter and any values assigned by the ADMIN_SET_INTRA_PARALLEL procedure.

If a query is compiled with DEGREE = ANY, the database manager chooses the degree of intrapartition parallelism on the basis of a number of factors, including the number of processors and the characteristics of the query. The actual degree used at run time might be lower than the number of processors, depending on these factors and the amount of activity on the system. The degree of parallelism might be reduced before query execution if the system is busy.

Use the Db2 explain facility to display information about the degree of parallelism chosen by the optimizer. Use the database system monitor to display information about the degree of parallelism actually being used at run time.

Parallelism in non-SMP environments

You can specify a degree of parallelism without having an SMP machine. For example, I/O-bound queries on a uniprocessor machine might benefit from declaring a degree of 2 or more. In this case, the processor might not have to wait for I/O tasks to complete before starting to process a new query. Utilities such as load can control I/O parallelism independently.

Lock management

Lock management is one of the factors that affect application performance. Review this section for details about lock management considerations that can help you to maximize the performance of database applications.

Lock escalation

Lock escalation is the process of converting many fine-grain locks to fewer coarse-grain locks, which reduces memory overhead at the cost of decreasing concurrency.

It is the act of releasing a large number of fine-grain row, MDC, LOB, or XML locks which are held by an application process on a single table, to acquire a table lock, or other coarse-grain lock such as block or LOB locks, of mode S or X instead.

Lock escalation occurs when an application exceeds the **MAXLOCKS** threshold or the database approaches the **LOCKLIST** limit. The database manager writes messages (AM5500W/ADM5501I) to the administration notification log which identifies the table for which lock escalation occurred, and some information to help you identify what plan or package was running when the escalation occurred.

The benefit of lock escalation is that operations that would otherwise fail with an SQL0912N error, can instead become successful due to lock escalation. However, the operation may still fail due to lock timeout or deadlock. As a drawback, lock escalation may negatively affect concurrency with other applications which may need to access the table.

Avoiding Lock Escalation

To avoid lock escalation, you can modify the application to acquire table locks using the **LOCK TABLE** statement. This is a good strategy for tables where concurrent access by many applications and users is not important.

You can also set the **DB2_AVOID_LOCK_ESCALATION** registry variable to ON, to fail the application which exceeded the MAXLOCKS threshold with an SQL0912N instead of performing lock escalation.

Locks and concurrency control

To provide concurrency control and prevent uncontrolled data access, the database manager places locks on buffer pools, tables, data partitions, table blocks, or table rows.

A *lock* associates a database manager resource with an application, called the *lock owner*, to control how other applications access the same resource.

The database manager uses row-level locking or table-level locking, as appropriate, based on:

- The isolation level specified at precompile time or when an application is bound to the database. The isolation level can be one of the following:
  - Uncommitted read (UR)
  - Cursor stability (CS)
  - Read stability (RS)
  - Repeatable read (RR)

  The different isolation levels are used to control access to uncommitted data, prevent lost updates, allow non-repeatable reads of data, and prevent phantom reads. To minimize performance impact, use the minimum isolation level that satisfies your application needs.
- The access plan selected by the optimizer. Table scans, index scans, and other methods of data access each require different types of access to the data.

- The LOCKSIZE attribute for the table. The LOCKSIZE clause on the ALTER TABLE statement indicates the granularity of the locks that are used when the table is accessed. The choices are: ROW for row locks, TABLE for table locks, or BLOCKINSERT for block locks on multidimensional clustering (MDC) tables only. When the BLOCKINSERT clause is used on an MDC table, row-level locking is performed, except during an insert operation, when block-level locking is done instead. Use the ALTER TABLE...LOCKSIZE BLOCKINSERT statement for MDC tables when transactions will be performing large inserts into disjointed cells. Use the ALTER TABLE...LOCKSIZE TABLE statement for read-only tables. This reduces the number of locks that are required for database activity. For partitioned tables, table locks are first acquired and then data partition locks are acquired, as dictated by the data that will be accessed.
- The amount of memory devoted to locking, which is controlled by the **locklist** database configuration parameter. If the lock list fills up, performance can degrade because of lock escalations and reduced concurrency among shared objects in the database. If lock escalations occur frequently, increase the value of **locklist**, **maxlocks**, or both. To reduce the number of locks that are held at one time, ensure that transactions commit frequently.

A buffer pool lock (exclusive) is set whenever a buffer pool is created, altered, or dropped. You might encounter this type of lock when collecting system monitoring data. The name of the lock is the identifier (ID) for the buffer pool itself.

In general, row-level locking is used unless one of the following is true:

- The isolation level is uncommitted read
- The isolation level is repeatable read and the access plan requires a scan with no index range predicates
- The table LOCKSIZE attribute is TABLE
- The lock list fills up, causing lock escalation
- An explicit table lock has been acquired through the LOCK TABLE statement, which prevents concurrent application processes from changing or using a table

In the case of an MDC table, block-level locking is used instead of row-level locking when:

- The table LOCKSIZE attribute is BLOCKINSERT
- The isolation level is repeatable read and the access plan involves predicates
- A searched update or delete operation involves predicates on dimension columns only

The duration of row locking varies with the isolation level being used:

- UR scans: No row locks are held unless row data is changing.
- CS scans: Row locks are generally held only while the cursor is positioned on the row. Note that in some cases, locks might not be held at all during a CS scan.
- RS scans: Qualifying row locks are held only for the duration of the transaction.
- RR scans: All row locks are held for the duration of the transaction.

Lock granularity

If one application holds a lock on a database object, another application might not be able to access that object. For this reason, row-level locks, which minimize the amount of data that is locked and therefore inaccessible, are better for maximum concurrency than block-level, data partition-level, or table-level locks.

However, locks require storage and processing time, so a single table lock minimizes lock overhead.

The LOCKSIZE clause of the ALTER TABLE statement specifies the granularity of locks at the row, data partition, block, or table level. Row locks are used by default. Use of this option in the table definition does not prevent normal lock escalation from occurring.

The ALTER TABLE statement specifies locks globally, affecting all applications and users that access that table. Individual applications might use the LOCK TABLE statement to specify table locks at an application level instead.

A permanent table lock defined by the ALTER TABLE statement might be preferable to a single-transaction table lock using the LOCK TABLE statement if:

- The table is read-only, and will always need only S locks. Other users can also obtain S locks on the table.
- The table is usually accessed by read-only applications, but is sometimes accessed by a single user for brief maintenance, and that user requires an X lock. While the maintenance program is running, read-only applications are locked out, but in other circumstances, read-only applications can access the table concurrently with a minimum of locking overhead.

For a multidimensional clustering (MDC) table, you can specify BLOCKINSERT with the LOCKSIZE clause in order to use block-level locking during insert operations only. When BLOCKINSERT is specified, row-level locking is performed for all other operations, but only minimally for insert operations. That is, block-level locking is used during the insertion of rows, but row-level locking is used to lock the next key if repeatable read (RR) scans are encountered in the record ID (RID) indexes as they are being updated. BLOCKINSERT locking might be beneficial when:

- There are multiple transactions doing mass insertions into separate cells
- Concurrent insertions into the same cell by multiple transactions is not occurring, or it is occurring with enough data inserted per cell by each of the transactions that the user is not concerned that each transaction will insert into separate blocks

Lock attributes

Database manager locks have several basic attributes.

These attributes include the following:

Mode The type of access allowed for the lock owner, as well as the type of access allowed for concurrent users of the locked object. It is sometimes referred to as the *state* of the lock.

Object
The resource being locked. The only type of object that you can lock explicitly is a table. The database manager also sets locks on other types of resources, such as rows and table spaces. Block locks can also be set for multidimensional clustering (MDC) or insert time clustering (ITC) tables, and data partition locks can be set for partitioned tables. The object being locked determines the *granularity* of the lock.

Lock count
The length of time during which a lock is held. The isolation level under which a query runs affects the lock count.

Table 14 lists the lock modes and describes their effects, in order of increasing control over resources.

Table 14. Lock Mode Summary

Lock Mode	Applicable Object Type	Description
IN (Intent None)	Table spaces, blocks, tables, data partitions	The lock owner can read any data in the object, including uncommitted data, but cannot update any of it. Other concurrent applications can read or update the table.
IS (Intent Share)	Table spaces, blocks, tables, data partitions	The lock owner can read data in the locked table, but cannot update this data. Other applications can read or update the table.
IX (Intent Exclusive)	Table spaces, blocks, tables, data partitions	The lock owner and concurrent applications can read and update data. Other concurrent applications can both read and update the table.
NS (Scan Share)	Rows	The lock owner and all concurrent applications can read, but not update, the locked row. This lock is acquired on rows of a table, instead of an S lock, where the isolation level of the application is either RS or CS.
NW (Next Key Weak Exclusive)	Rows	When a row is inserted into an index, an NW lock is acquired on the next row. This occurs only if the next row is currently locked by an RR scan. The lock owner can read but not update the locked row. This lock mode is similar to an X lock, except that it is also compatible with NS locks.
S (Share)	Rows, blocks, tables, data partitions	The lock owner and all concurrent applications can read, but not update, the locked data.
SIX (Share with Intent Exclusive)	Tables, blocks, data partitions	The lock owner can read and update data. Other concurrent applications can read the table.
U (Update)	Rows, blocks, tables, data partitions	The lock owner can update data. Other units of work can read the data in the locked object, but cannot update it.
X (Exclusive)	Rows, blocks, tables, buffer pools, data partitions	The lock owner can both read and update data in the locked object. Only uncommitted read (UR) applications can access the locked object.
Z (Super Exclusive)	Table spaces, tables, data partitions, blocks	This lock is acquired on a table under certain conditions, such as when the table is altered or dropped, an index on the table is created or dropped, or for some types of table reorganization. No other concurrent application can read or update the table.

Factors that affect locking

Several factors affect the mode and granularity of database manager locks.

These factors include:

- The type of processing that the application performs
- The data access method
- The values of various configuration parameters

Locks and types of application processing:

For the purpose of determining lock attributes, application processing can be classified as one of the following types: read-only, intent to change, change, and cursor controlled.

- Read-only

  This processing type includes all SELECT statements that are intrinsically read-only, have an explicit FOR READ ONLY clause, or are ambiguous, but the

query compiler assumes that they are read-only because of the BLOCKING option value that the **PREP** or **BIND** command specifies. This type requires only share locks (IS, NS, or S).

- Intent to change

  This processing type includes all SELECT statements that have a FOR UPDATE clause, a USE AND KEEP UPDATE LOCKS clause, a USE AND KEEP EXCLUSIVE LOCKS clause, or are ambiguous, but the query compiler assumes that change is intended. This type uses share and update locks (S, U, or X for rows; IX, S, U, or X for blocks; and IX, U, or X for tables).

- Change

  This processing type includes UPDATE, INSERT, and DELETE statements, but not UPDATE WHERE CURRENT OF or DELETE WHERE CURRENT OF. This type requires exclusive locks (IX or X).

- Cursor controlled

  This processing type includes UPDATE WHERE CURRENT OF and DELETE WHERE CURRENT OF. This type requires exclusive locks (IX or X).

A statement that inserts, updates, or deletes data in a target table, based on the result from a subselect statement, does two types of processing. The rules for read-only processing determine the locks for the tables that return data in the subselect statement. The rules for change processing determine the locks for the target table.

Locks and data-access methods:

An *access plan* is the method that the optimizer selects to retrieve data from a specific table. The access plan can have a significant effect on lock modes.

If an index scan is used to locate a specific row, the optimizer will usually choose row-level locking (IS) for the table. For example, if the EMPLOYEE table has an index on employee number (EMPNO), access through that index might be used to select information for a single employee:

```
select * from employee
  where empno = '000310'
```

If an index is not used, the entire table must be scanned in sequence to find the required rows, and the optimizer will likely choose a single table-level lock (S). For example, if there is no index on the column SEX, a table scan might be used to select all male employees, as follows:

```
select * from employee
  where sex = 'M'
```

Note: Cursor-controlled processing uses the lock mode of the underlying cursor until the application finds a row to update or delete. For this type of processing, no matter what the lock mode of the cursor might be, an exclusive lock is always obtained to perform the update or delete operation.

Locking in range-clustered tables works slightly differently from standard key locking. When accessing a range of rows in a range-clustered table, all rows in the range are locked, even when some of those rows are empty. In standard key locking, only rows with existing data are locked.

Deferred access to data pages implies that access to a row occurs in two steps, which results in more complex locking scenarios. The timing of lock acquisition and the persistence of locks depend on the isolation level. Because the repeatable

read (RR) isolation level retains all locks until the end of a transaction, the locks acquired in the first step are held, and there is no need to acquire further locks during the second step. For the read stability (RS) and cursor stability (CS) isolation levels, locks must be acquired during the second step. To maximize concurrency, locks are not acquired during the first step, and the reapplication of all predicates ensures that only qualifying rows are returned.

Lock type compatibility

Lock compatibility becomes an issue when one application holds a lock on an object and another application requests a lock on the same object. When the two lock modes are compatible, the request for a second lock on the object can be granted.

If the lock mode of the requested lock is not compatible with the lock that is already held, the lock request cannot be granted. Instead, the request must wait until the first application releases its lock, and all other existing incompatible locks are released.

Table 15 shows which lock types are compatible (indicated by a **yes**) and which types are not (indicated by a **no**). Note that a timeout can occur when a requestor is waiting for a lock.

Table 15. Lock Type Compatibility

	State of Held Resource										
State Being Requested	**None**	**IN**	**IS**	**NS**	**S**	**IX**	**SIX**	**U**	**X**	**Z**	**NW**
None	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
IN (Intent None)	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes
IS (Intent Share)	yes	yes	yes	yes	yes	yes	yes	yes	no	no	no
NS (Scan Share)	yes	yes	yes	yes	yes	no	no	yes	no	no	yes
S (Share)	yes	yes	yes	yes	yes	no	no	yes	no	no	no
IX (Intent Exclusive)	yes	yes	yes	no	no	yes	no	no	no	no	no
SIX (Share with Intent Exclusive)	yes	yes	yes	no	no	no	no	no	no	no	no
U (Update)	yes	yes	yes	yes	yes	no	no	no	no	no	no
X (Exclusive)	yes	yes	no	no	no	no	no	no	no	no	no
Z (Super Exclusive)	yes	no	no	no	no	no	no	no	no	no	no
NW (Next Key Weak Exclusive)	yes	yes	no	yes	no	no	no	no	no	no	no

Next-key locking

During insertion of a key into an index, the row that corresponds to the key that will follow the new key in the index is locked only if that row is currently locked by a repeatable read (RR) index scan. When this occurs, insertion of the new index key is deferred until the transaction that performed the RR scan completes.

The lock mode that is used for the next-key lock is NW (next key weak exclusive). This next-key lock is released before key insertion occurs; that is, before a row is inserted into the table.

Key insertion also occurs when updates to a row result in a change to the value of the index key for that row, because the original key value is marked deleted and

the new key value is inserted into the index. For updates that affect only the include columns of an index, the key can be updated in place, and no next-key locking occurs.

During RR scans, the row that corresponds to the key that follows the end of the scan range is locked in S mode. If no keys follow the end of the scan range, an end-of-table lock is acquired to lock the end of the index. In the case of partitioned indexes for partitioned tables, locks are acquired to lock the end of each index partition, instead of just one lock for the end of the index. If the key that follows the end of the scan range is marked deleted, one of the following actions occurs:

- The scan continues to lock the corresponding rows until it finds a key that is not marked deleted
- The scan locks the corresponding row for that key
- The scan locks the end of the index

Lock modes and access plans for standard tables

The type of lock that a standard table obtains depends on the isolation level that is in effect and on the data access plan that is being used.

The following tables show the types of locks that are obtained for standard tables under each isolation level for different access plans. Each entry has two parts: the table lock and the row lock. A hyphen indicates that a particular lock granularity is not available.

Tables 7-12 show the types of locks that are obtained when the reading of data pages is deferred to allow the list of rows to be further qualified using multiple indexes, or sorted for efficient prefetching.

- Table 1. Lock Modes for Table Scans with No Predicates
- Table 2. Lock Modes for Table Scans with Predicates
- Table 3. Lock Modes for RID Index Scans with No Predicates
- Table 4. Lock Modes for RID Index Scans with a Single Qualifying Row
- Table 5. Lock Modes for RID Index Scans with Start and Stop Predicates Only
- Table 6. Lock Modes for RID Index Scans with Index and Other Predicates (sargs, resids) Only
- Table 7. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with No Predicates
- Table 8. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with No Predicates
- Table 9. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Predicates (sargs, resids)
- Table 10. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Predicates (sargs, resids)
- Table 11. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Start and Stop Predicates Only
- Table 12. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Start and Stop Predicates Only

Note:

1. Block-level locks are also available for multidimensional clustering (MDC) and insert time clustering (ITC) tables.
2. Lock modes can be changed explicitly with the *lock-request-clause* of a SELECT statement.

Table 16. Lock Modes for Table Scans with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	S/-	U/-	SIX/X	X/-	X/-
RS	IS/NS	IX/U	IX/X	IX/X	IX/X
CS	IS/NS	IX/U	IX/X	IX/X	IX/X
UR	IN/-	IX/U	IX/X	IX/X	IX/X

Table 17. Lock Modes for Table Scans with Predicates

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	S/-	U/-	SIX/X	U/-	SIX/X
RS	IS/NS	IX/U	IX/X	IX/U	IX/X
CS	IS/NS	IX/U	IX/X	IX/U	IX/X
UR	IN/-	IX/U	IX/X	IX/U	IX/X

Note: Under the UR isolation level, if there are predicates on include columns in the index, the isolation level is upgraded to CS and the locks are upgraded to an IS table lock or NS row locks.

Table 18. Lock Modes for RID Index Scans with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	S/-	IX/S	IX/X	X/-	X/-
RS	IS/NS	IX/U	IX/X	IX/X	IX/X
CS	IS/NS	IX/U	IX/X	IX/X	IX/X
UR	IN/-	IX/U	IX/X	IX/X	IX/X

Table 19. Lock Modes for RID Index Scans with a Single Qualifying Row

Isolation level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S	IX/U	IX/X	IX/X	IX/X
RS	IS/NS	IX/U	IX/X	IX/X	IX/X
CS	IS/NS	IX/U	IX/X	IX/X	IX/X
UR	IN/-	IX/U	IX/X	IX/X	IX/X

Table 20. Lock Modes for RID Index Scans with Start and Stop Predicates Only

Isolation level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S	IX/S	IX/X	IX/X	IX/X
RS	IS/NS	IX/U	IX/X	IX/X	IX/X
CS	IS/NS	IX/U	IX/X	IX/X	IX/X
UR	IN/-	IX/U	IX/X	IX/X	IX/X

Table 21. Lock Modes for RID Index Scans with Index and Other Predicates (sargs, resids) Only

Isolation level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S	IX/S	IX/X	IX/S	IX/X
RS	IS/NS	IX/U	IX/X	IX/U	IX/X
CS	IS/NS	IX/U	IX/X	IX/U	IX/X
UR	IN/-	IX/U	IX/X	IX/U	IX/X

Table 22. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S	IX/S		X/-	
RS	IN/-	IN/-		IN/-	
CS	IN/-	IN/-		IN/-	
UR	IN/-	IN/-		IN/-	

Table 23. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IN/-	IX/S	IX/X	X/-	X/-
RS	IS/NS	IX/U	IX/X	IX/X	IX/X
CS	IS/NS	IX/U	IX/X	IX/X	IX/X
UR	IN/-	IX/U	IX/X	IX/X	IX/X

Table 24. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Predicates (sargs, resids)

Isolation level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S	IX/S		IX/S	
RS	IN/-	IN/-		IN/-	
CS	IN/-	IN/-		IN/-	
UR	IN/-	IN/-		IN/-	

Table 25. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Predicates (sargs, resids)

Isolation level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IN/-	IX/S	IX/X	IX/S	IX/X
RS	IS/NS	IX/U	IX/X	IX/U	IX/X
CS	IS/NS	IX/U	IX/X	IX/U	IX/X
UR	IN/-	IX/U	IX/X	IX/U	IX/X

Table 26. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Start and Stop Predicates Only

Isolation level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S	IX/S		IX/X	
RS	IN/-	IN/-		IN/-	
CS	IN/-	IN/-		IN/-	
UR	IN/-	IN/-		IN/-	

Table 27. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Start and Stop Predicates Only

Isolation level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IN/-	IX/S	IX/X	IX/X	IX/X
RS	IS/NS	IX/U	IX/X	IX/U	IX/X
CS	IS/NS	IX/U	IX/X	IX/U	IX/X
UR	IS/-	IX/U	IX/X	IX/U	IX/X

Lock modes for MDC and ITC tables and RID index scans

The type of lock that a multidimensional clustering (MDC) or insert time clustering (ITC) table obtains during a table or RID index scan depends on the isolation level that is in effect and on the data access plan that is being used.

The following tables show the types of locks that are obtained for MDC and ITC tables under each isolation level for different access plans. Each entry has three parts: the table lock, the block lock, and the row lock. A hyphen indicates that a particular lock granularity is not available.

Tables 9-14 show the types of locks that are obtained for RID index scans when the reading of data pages is deferred. Under the UR isolation level, if there are predicates on include columns in the index, the isolation level is upgraded to CS and the locks are upgraded to an IS table lock, an IS block lock, or NS row locks.

- Table 1. Lock Modes for Table Scans with No Predicates
- Table 2. Lock Modes for Table Scans with Predicates on Dimension Columns Only
- Table 3. Lock Modes for Table Scans with Other Predicates (sargs, resids)
- Table 4. Lock Modes for RID Index Scans with No Predicates
- Table 5. Lock Modes for RID Index Scans with a Single Qualifying Row
- Table 6. Lock Modes for RID Index Scans with Start and Stop Predicates Only
- Table 7. Lock Modes for RID Index Scans with Index Predicates Only
- Table 8. Lock Modes for RID Index Scans with Other Predicates (sargs, resids)
- Table 9. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with No Predicates
- Table 10. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with No Predicates
- Table 11. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Predicates (sargs, resids)
- Table 12. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Predicates (sargs, resids)
- Table 13. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Start and Stop Predicates Only
- Table 14. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Start and Stop Predicates Only

Note: Lock modes can be changed explicitly with the *lock-request-clause* of a SELECT statement.

Table 28. Lock Modes for Table Scans with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	S/-/-	U/-/-	SIX/IX/X	X/-/-	X/-/-
RS	IS/IS/NS	IX/IX/U	IX/IX/U	IX/X/-	IX/I/-
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/X/-	IX/X/-
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/X/-	IX/X/-

Table 29. Lock Modes for Table Scans with Predicates on Dimension Columns Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	S/-/-	U/-/-	SIX/IX/X	U/-/-	SIX/X/-
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/U/-	X/X/-
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/U/-	X/X/-
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/U/-	X/X/-

Table 30. Lock Modes for Table Scans with Other Predicates (sargs, resids)

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	S/-/-	U/-/-	SIX/IX/X	U/-/-	SIX/IX/X
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X

Table 31. Lock Modes for RID Index Scans with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	S/-/-	IX/IX/S	IX/IX/X	X/-/-	X/-/-
RS	IS/IS/NS	IX/IX/U	IX/IX/X	X/X/X	X/X/X
CS	IS/IS/NS	IX/IX/U	IX/IX/X	X/X/X	X/X/X
UR	IN/IN/-	IX/IX/U	IX/IX/X	X/X/X	X/X/X

Table 32. Lock Modes for RID Index Scans with a Single Qualifying Row

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/IS/S	IX/IX/U	IX/IX/X	X/X/X	X/X/X
RS	IS/IS/NS	IX/IX/U	IX/IX/X	X/X/X	X/X/X
CS	IS/IS/NS	IX/IX/U	IX/IX/X	X/X/X	X/X/X
UR	IN/IN/-	IX/IX/U	IX/IX/X	X/X/X	X/X/X

Table 33. Lock Modes for RID Index Scans with Start and Stop Predicates Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/IS/S	IX/IX/S	IX/IX/X	IX/IX/X	IX/IX/X
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/X	IX/IX/X

Table 33. Lock Modes for RID Index Scans with Start and Stop Predicates Only (continued)

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/X	IX/IX/X
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/X	IX/IX/X

Table 34. Lock Modes for RID Index Scans with Index Predicates Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S/S	IX/IX/S	IX/IX/X	IX/IX/S	IX/IX/X
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X

Table 35. Lock Modes for RID Index Scans with Other Predicates (sargs, resids)

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S/S	IX/IX/S	IX/IX/X	IX/IX/S	IX/IX/X
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X

Table 36. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S/S	IX/IX/S		X/-/-	
RS	IN/IN/-	IN/IN/-		IN/IN/-	
CS	IN/IN/-	IN/IN/-		IN/IN/-	
UR	IN/IN/-	IN/IN/-		IN/IN/-	

Table 37. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IN/IN/-	IX/IX/S	IX/IX/X	X/-/-	X/-/-

Table 37. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with No Predicates (continued)

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/X	IX/IX/X
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/X	IX/IX/X
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/X	IX/IX/X

Table 38. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Predicates (sargs, resids)

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S/-	IX/IX/S		IX/IX/S	
RS	IN/IN/-	IN/IN/-		IN/IN/-	
CS	IN/IN/-	IN/IN/-		IN/IN/-	
UR	IN/IN/-	IN/IN/-		IN/IN/-	

Table 39. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Predicates (sargs, resids)

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IN/IN/-	IX/IX/S	IX/IX/X	IX/IX/S	IX/IX/X
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X

Table 40. Lock Modes for Index Scans Used for Deferred Data Page Access: RID Index Scan with Start and Stop Predicates Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/IS/S	IX/IX/S		IX/IX/X	
RS	IN/IN/-	IN/IN/-		IN/IN/-	
CS	IN/IN/-	IN/IN/-		IN/IN/-	
UR	IN/IN/-	IN/IN/-		IN/IN/-	

Table 41. Lock Modes for Index Scans Used for Deferred Data Page Access: After a RID Index Scan with Start and Stop Predicates Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IN/IN/-	IX/IX/S	IX/IX/X	IX/IX/X	IX/IX/X
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
UR	IS/-/-	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X

Lock modes for MDC block index scans

The type of lock that a multidimensional clustering (MDC) table obtains during a block index scan depends on the isolation level that is in effect and on the data access plan that is being used.

The following tables show the types of locks that are obtained for MDC tables under each isolation level for different access plans. Each entry has three parts: the table lock, the block lock, and the row lock. A hyphen indicates that a particular lock granularity is not available.

Tables 5-12 show the types of locks that are obtained for block index scans when the reading of data pages is deferred.

- Table 1. Lock Modes for Index Scans with No Predicates
- Table 2. Lock Modes for Index Scans with Predicates on Dimension Columns Only
- Table 3. Lock Modes for Index Scans with Start and Stop Predicates Only
- Table 4. Lock Modes for Index Scans with Predicates
- Table 5. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with No Predicates
- Table 6. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with No Predicates
- Table 7. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Predicates on Dimension Columns Only
- Table 8. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Predicates on Dimension Columns Only
- Table 9. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Start and Stop Predicates Only
- Table 10. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Start and Stop Predicates Only
- Table 11. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Other Predicates (sargs, resids)
- Table 12. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Other Predicates (sargs, resids)

Note: Lock modes can be changed explicitly with the *lock-request-clause* of a SELECT statement.

Table 42. Lock Modes for Index Scans with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	S/--/--	IX/IX/S	IX/IX/X	X/--/--	X/--/--
RS	IS/IS/NS	IX/IX/U	IX/IX/X	X/X/--	X/X/--
CS	IS/IS/NS	IX/IX/U	IX/IX/X	X/X/--	X/X/--
UR	IN/IN/-	IX/IX/U	IX/IX/X	X/X/--	X/X/--

Table 43. Lock Modes for Index Scans with Predicates on Dimension Columns Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/-/-	IX/IX/S	IX/IX/X	X/-/-	X/-/-
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/X/-	IX/X/-
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/X/-	IX/X/-
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/X/-	IX/X/-

Table 44. Lock Modes for Index Scans with Start and Stop Predicates Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S/-	IX/IX/S	IX/IX/S	IX/IX/S	IX/IX/S
RS	IX/IX/S	IX/IX/U	IX/IX/X	IX/IX/-	IX/IX/-
CS	IX/IX/S	IX/IX/U	IX/IX/X	IX/IX/-	IX/IX/-
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/-	IX/IX/-

Table 45. Lock Modes for Index Scans with Predicates

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S/-	IX/IX/S	IX/IX/X	IX/IX/S	IX/IX/X
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X

Table 46. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S/--	IX/IX/S		X/--/--	
RS	IN/IN/--	IN/IN/--		IN/IN/--	
CS	IN/IN/--	IN/IN/--		IN/IN/--	
UR	IN/IN/--	IN/IN/--		IN/IN/--	

Table 47. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with No Predicates

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IN/IN/--	IX/IX/S	IX/IX/X	X/--/--	X/--/--
RS	IS/IS/NS	IX/IX/U	IX/IX/X	X/X/--	X/X/--
CS	IS/IS/NS	IX/IX/U	IX/IX/X	X/X/--	X/X/--
UR	IN/IN/--	IX/IX/U	IX/IX/X	X/X/--	X/X/--

Table 48. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Predicates on Dimension Columns Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S/--	IX/IX/--		IX/S/--	
RS	IS/IS/NS	IX/--/--		IX/--/--	
CS	IS/IS/NS	IX/--/--		IX/--/--	
UR	IN/IN/--	IX/--/--		IX/--/--	

Table 49. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Predicates on Dimension Columns Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IN/IN/--	IX/IX/S	IX/IX/X	IX/S/--	IX/X/--
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/U/--	IX/X/--
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/U/--	IX/X/--
UR	IN/IN/--	IX/IX/U	IX/IX/X	IX/U/--	IX/X/--

Table 50. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Start and Stop Predicates Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S/--	IX/IX/--		IX/X/--	
RS	IN/IN/--	IN/IN/--		IN/IN/--	
CS	IN/IN/--	IN/IN/--		IN/IN/--	
UR	IN/IN/--	IN/IN/--		IN/IN/--	

Table 51. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Start and Stop Predicates Only

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IN/IN/--	IX/IX/X		IX/X/--	
RS	IS/IS/NS	IN/IN/--		IN/IN/--	
CS	IS/IS/NS	IN/IN/--		IN/IN/--	
UR	IS/--/--	IN/IN/--		IN/IN/--	

Table 52. Lock Modes for Index Scans Used for Deferred Data Page Access: Block Index Scan with Other Predicates (sargs, resids)

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IS/S/--	IX/IX/--		IX/IX/--	
RS	IN/IN/--	IN/IN/--		IN/IN/--	
CS	IN/IN/--	IN/IN/--		IN/IN/--	
UR	IN/IN/--	IN/IN/--		IN/IN/--	

Table 53. Lock Modes for Index Scans Used for Deferred Data Page Access: After a Block Index Scan with Other Predicates (sargs, resids)

Isolation level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan	Update or delete
RR	IN/IN/--	IX/IX/S	IX/IX/X	IX/IX/S	IX/IX/X
RS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
CS	IS/IS/NS	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X
UR	IN/IN/--	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X

Locking behavior on partitioned tables

In addition to an overall table lock, there is a lock for each data partition of a partitioned table.

This allows for finer granularity and increased concurrency compared to a nonpartitioned table. The data partition lock is identified in output from the **db2pd** command, event monitors, administrative views, and table functions.

When a table is accessed, a table lock is obtained first, and then data partition locks are obtained as required. Access methods and isolation levels might require the locking of data partitions that are not represented in the result set. After these data partition locks are acquired, they might be held as long as the table lock. For example, a cursor stability (CS) scan over an index might keep the locks on previously accessed data partitions to reduce the cost of reacquiring data partition locks later.

Data partition locks also carry the cost of ensuring access to table spaces. For nonpartitioned tables, table space access is handled by table locks. Data partition locking occurs even if there is an exclusive or share lock at the table level.

Finer granularity allows one transaction to have exclusive access to a specific data partition and to avoid row locking while other transactions are accessing other data partitions. This can be the result of the plan that is chosen for a mass update, or because of the escalation of locks to the data partition level. The table lock for many access methods is normally an intent lock, even if the data partitions are locked in share or exclusive mode. This allows for increased concurrency. However, if non-intent locks are required at the data partition level, and the plan indicates that all data partitions might be accessed, then a non-intent lock might be chosen at the table level to prevent data partition deadlocks between concurrent transactions.

LOCK TABLE statements

For partitioned tables, the only lock acquired by the LOCK TABLE statement is a table-level lock. This prevents row locking by subsequent data manipulation language (DML) statements, and avoids deadlocks at the row, block, or data partition level. The IN EXCLUSIVE MODE option can be used to guarantee exclusive access when updating indexes, which is useful in limiting the growth of indexes during a large update.

Effect of the LOCKSIZE TABLE option on the ALTER TABLE statement

The LOCKSIZE TABLE option ensures that a table is locked in share or exclusive mode with no intent locks. For a partitioned table, this locking strategy is applied to both the table lock and to data partition locks.

Row- and block-level lock escalation

Row- and block-level locks in partitioned tables can be escalated to the data partition level. When this occurs, a table is more accessible to other transactions, even if a data partition is escalated to share, exclusive, or super exclusive mode, because other data partitions remain unaffected. The notification log entry for an escalation includes the impacted data partition and the name of the table.

Exclusive access to a nonpartitioned index cannot be ensured by lock escalation. For exclusive access, one of the following conditions must be true:

- The statement must use an exclusive table-level lock
- An explicit LOCK TABLE IN EXCLUSIVE MODE statement must be issued
- The table must have the LOCKSIZE TABLE attribute

In the case of partitioned indexes, exclusive access to an index partition is ensured by lock escalation of the data partition to an exclusive or super exclusive access mode.

Interpreting lock information

The SNAPLOCK administrative view can help you to interpret lock information that is returned for a partitioned table. The following SNAPLOCK administrative view was captured during an offline index reorganization.

```
SELECT SUBSTR(TABNAME, 1, 15) TABNAME, TAB_FILE_ID, SUBSTR(TBSP_NAME, 1, 15) TBSP_NAME,
       DATA_PARTITION_ID, LOCK_OBJECT_TYPE, LOCK_MODE, LOCK_ESCALATION L_ESCALATION
  FROM SYSIBMADM.SNAPLOCK
  WHERE TABNAME like 'TP1' and LOCK_OBJECT_TYPE like 'TABLE_%'
  ORDER BY TABNAME, DATA_PARTITION_ID, LOCK_OBJECT_TYPE, TAB_FILE_ID, LOCK_MODE


TABNAME TAB_FILE_ID TBSP_NAME  DATA_PARTITION_ID LOCK_OBJECT_TYPE LOCK_MODE L_ESCALATION
------- ----------- ---------- ----------------- --------------- --------- ------------
TP1           32768 -                         -1 TABLE_LOCK      Z                    0
TP1               4 USERSPACE1                 0 TABLE_PART_LOCK Z                    0
TP1               5 USERSPACE1                 1 TABLE_PART_LOCK Z                    0
TP1               6 USERSPACE1                 2 TABLE_PART_LOCK Z                    0
TP1               7 USERSPACE1                 3 TABLE_PART_LOCK Z                    0
TP1               8 USERSPACE1                 4 TABLE_PART_LOCK Z                    0
TP1               9 USERSPACE1                 5 TABLE_PART_LOCK Z                    0
TP1              10 USERSPACE1                 6 TABLE_PART_LOCK Z                    0
TP1              11 USERSPACE1                 7 TABLE_PART_LOCK Z                    0
TP1              12 USERSPACE1                 8 TABLE_PART_LOCK Z                    0
TP1              13 USERSPACE1                 9 TABLE_PART_LOCK Z                    0
TP1              14 USERSPACE1                10 TABLE_PART_LOCK Z                    0
TP1              15 USERSPACE1                11 TABLE_PART_LOCK Z                    0
TP1               4 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1               5 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1               6 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1               7 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1               8 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1               9 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              10 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              11 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              12 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              13 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              14 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              15 USERSPACE1                 - TABLE_LOCK      Z                    0
TP1              16 USERSPACE1                 - TABLE_LOCK      Z                    0

  26 record(s) selected.
```

In this example, a lock object of type TABLE_LOCK and a DATA_PARTITION_ID of -1 are used to control access to and concurrency on the partitioned table TP1. The lock objects of type TABLE_PART_LOCK are used to control most access to and concurrency on each data partition.

There are additional lock objects of type TABLE_LOCK captured in this output (TAB_FILE_ID 4 through 16) that do not have a value for DATA_PARTITION_ID. A lock of this type, where an object with a TAB_FILE_ID and a TBSP_NAME correspond to a data partition or index on the partitioned table, might be used to control concurrency with the online backup utility.

Lock conversion

Changing the mode of a lock that is already held is called *lock conversion*.

Lock conversion occurs when a process accesses a data object on which it already holds a lock, and the access mode requires a more restrictive lock than the one

already held. A process can hold only one lock on a data object at any given time, although it can request a lock on the same data object many times indirectly through a query.

Some lock modes apply only to tables, others only to rows, blocks, or data partitions. For rows or blocks, conversion usually occurs if an X lock is needed and an S or U lock is held.

IX and S locks are special cases with regard to lock conversion. Neither is considered to be more restrictive than the other, so if one of these locks is held and the other is required, the conversion results in a SIX (Share with Intent Exclusive) lock. All other conversions result in the requested lock mode becoming the held lock mode if the requested mode is more restrictive.

A dual conversion might also occur when a query updates a row. If the row is read through index access and locked as S, the table that contains the row has a covering intention lock. But if the lock type is IS instead of IX, and the row is subsequently changed, the table lock is converted to an IX and the row lock is converted to an X.

Lock conversion usually takes place implicitly as a query executes. The system monitor elements **lock_current_mode** and **lock_mode** can provide information about lock conversions occurring in your database.

Lock waits and timeouts

Lock timeout detection is a database manager feature that prevents applications from waiting indefinitely for a lock to be released.

For example, a transaction might be waiting for a lock that is held by another user's application, but the other user has left the workstation without allowing the application to commit the transaction, which would release the lock. To avoid stalling an application in such a case, set the **locktimeout** database configuration parameter to the maximum time that any application should have to wait to obtain a lock.

Setting this parameter helps to avoid global deadlocks, especially in distributed unit of work (DUOW) applications. If the time during which a lock request is pending is greater than the **locktimeout** value, an error is returned to the requesting application and its transaction is rolled back. For example, if APPL1 tries to acquire a lock that is already held by APPL2, APPL1 receives SQLCODE -911 (SQLSTATE 40001) with reason code 68 if the timeout period expires. The default value for **locktimeout** is -1, which means that lock timeout detection is disabled.

For table, row, data partition, and multidimensional clustering (MDC) block locks, an application can override the **locktimeout** value by changing the value of the CURRENT LOCK TIMEOUT special register.

To generate a report file about lock timeouts, set the **DB2_CAPTURE_LOCKTIMEOUT** registry variable to ON. The lock timeout report includes information about the key applications that were involved in lock contentions that resulted in lock timeouts, as well as details about the locks, such as lock name, lock type, row ID, table space ID, and table ID. Note that this variable is deprecated and might be removed in a future release because there are new methods to collect lock timeout events using the CREATE EVENT MONITOR FOR LOCKING statement.

To log more information about lock-request timeouts in the **db2diag** log files, set the value of the **diaglevel** database manager configuration parameter to 4. The logged information includes the name of the locked object, the lock mode, and the application that is holding the lock. The current dynamic SQL or XQuery statement or static package name might also be logged. A dynamic SQL or XQuery statement is logged only at **diaglevel** 4.

You can get additional information about lock waits and lock timeouts from the lock wait information system monitor elements, or from the **db.apps_waiting_locks** health indicator.

Specifying a lock wait mode strategy:

A session can specify a lock wait mode strategy, which is used when the session requires a lock that it cannot obtain immediately.

The strategy indicates whether the session will:

- Return an SQLCODE and SQLSTATE when it cannot obtain a lock
- Wait indefinitely for a lock
- Wait a specified amount of time for a lock
- Use the value of the **locktimeout** database configuration parameter when waiting for a lock

The lock wait mode strategy is specified through the SET CURRENT LOCK TIMEOUT statement, which changes the value of the CURRENT LOCK TIMEOUT special register. This special register specifies the number of seconds to wait for a lock before returning an error indicating that a lock cannot be obtained.

Traditional locking approaches can result in applications blocking each other. This happens when one application must wait for another application to release its lock. Strategies to deal with the impact of such blocking usually provide a mechanism to specify the maximum acceptable duration of the block. That is the amount of time that an application will wait prior to returning without a lock. Previously, this was only possible at the database level by changing the value of the **locktimeout** database configuration parameter.

The value of **locktimeout** applies to all locks, but the lock types that are impacted by the lock wait mode strategy include row, table, index key, and multidimensional clustering (MDC) block locks.

Deadlocks

A deadlock is created when two applications lock data that is needed by the other, resulting in a situation in which neither application can continue executing.

For example, in Figure 23 on page 220, there are two applications running concurrently: Application A and Application B. The first transaction for Application A is to update the first row in Table 1, and the second transaction is to update the second row in Table 2. Application B updates the second row in Table 2 first, and then the first row in Table 1. At time T1, Application A locks the first row in Table 1. At the same time, Application B locks the second row in Table 2. At time T2, Application A requests a lock on the second row in Table 2. However, at the same time, Application B tries to lock the first row in Table 1. Because Application A will not release its lock on the first row in Table 1 until it can complete an update to the second row in Table 2, and Application B will not release its lock on the second row in Table 2 until it can complete an update to the first row in Table 1, a deadlock occurs. The applications wait until one of them releases its lock on the

data.

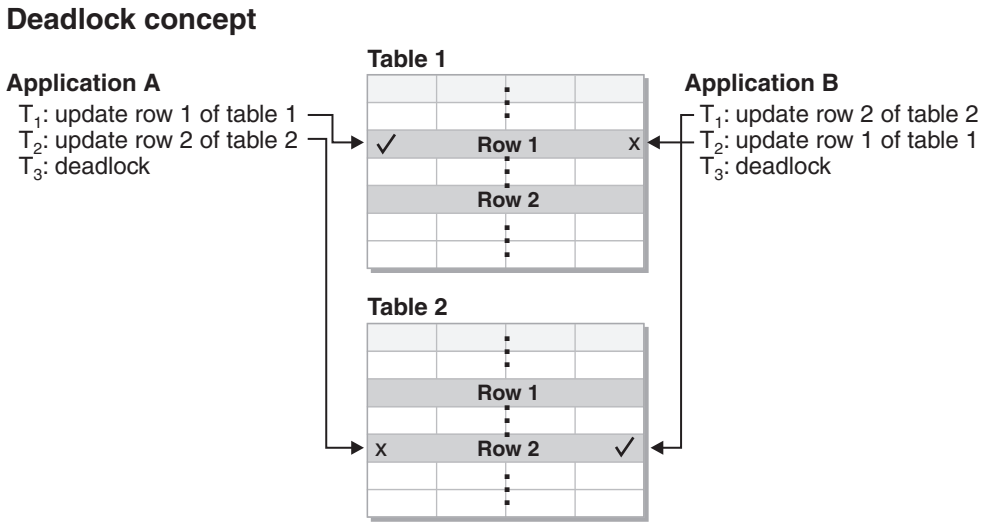


Figure 23. Deadlock between applications

Because applications do not voluntarily release locks on data that they need, a deadlock detector process is required to break deadlocks. The deadlock detector monitors information about agents that are waiting on locks, and awakens at intervals that are specified by the **dlchktime** database configuration parameter.

If it finds a deadlock, the deadlock detector arbitrarily selects one deadlocked process as the *victim process* to roll back. The victim process is awakened, and returns SQLCODE -911 (SQLSTATE 40001), with reason code 2, to the calling application. The database manager rolls back uncommitted transactions from the selected process automatically. When the rollback operation is complete, locks that belonged to the victim process are released, and the other processes involved in the deadlock can continue.

To ensure good performance, select an appropriate value for **dlchktime**. An interval that is too short causes unnecessary overhead, and an interval that is too long allows deadlocks to linger.

In a partitioned database environment, the value of **dlchktime** is applied only at the catalog database partition. If a large number of deadlocks are occurring, increase the value of **dlchktime** to account for lock waits and communication waits.

To avoid deadlocks when applications read data that they intend to subsequently update:

- Use the FOR UPDATE clause when performing a select operation. This clause ensures that a U lock is set when a process attempts to read data, and it does not allow row blocking.
- Use the WITH RR or WITH RS and USE AND KEEP UPDATE LOCKS clauses in queries. These clauses ensure that a U lock is set when a process attempts to read data, and they allow row blocking.

In a federated system, the data that is requested by an application might not be available because of a deadlock at the data source. When this happens, the Db2 server relies on the deadlock handling facilities at the data source. If deadlocks

occur across more than one data source, the Db2 server relies on data source timeout mechanisms to break the deadlocks.

To log more information about deadlocks, set the value of the **diaglevel** database manager configuration parameter to 4. The logged information includes the name of the locked object, the lock mode, and the application that is holding the lock. The current dynamic SQL and XQuery statement or static package name might also be logged.

Explicit hierarchical locking for Db2 pureScale environments

Explicit hierarchical locking (EHL) for IBM Db2 pureScale Feature takes advantage of the implicit internal locking hierarchy that exists between table locks, row locks, and page locks. EHL functionality helps avoid most communication and data sharing memory usage for tables.

Table locks supersede row locks or page locks in the locking hierarchy. When a table lock is held in super exclusive mode, EHL enhances performance for Db2 pureScale instances by not propagating row locks, page locks, or page writes to the caching facility (CF).

EHL is not enabled by default in Db2 pureScale environments. However, it can be enabled or disabled by using the **opt_direct_wrkld** database configuration parameter. When turned on, tables which are detected to be accessed primarily by a single member are optimized to avoid CF communication. The table lock is held in super exclusive mode on such a member while this state is in effect. Attempts to access the table by a remote member automatically terminates this mode.

There are two EHL states:

Table 54. EHL table states

Table state	Description
NOT_SHARED / DIRECTED_ACCESS	Refers to a table that is in explicit hierarchical locking state. Row locks, page locks, and page writes are managed only on the local member.
SHARED/FULLY SHARED	Refers to a table that is not in explicit hierarchical locking state. Row locks, page locks, and page writes are coordinated by using the CF.

Regular tables, range partitioned tables, or partitioned indexes might exist in one of the prior states, or in a transitional state between the SHARED and NOT_SHARED states:

EHL is useful for the following types of workloads, as they are able to take advantage of this optimization:

- Grid deployments, where each application has affinities to a single member and where most of its data access is only for these particular applications. In a database grid environment a Db2 pureScale cluster has multiple databases, but any single database is accessed only by a single member. In this case, all the tables in each database move into the NOT_SHARED state.
- Partitioned or partitionable workloads where work is directed such that certain tables are accessed only by a single member. These workloads include directed access workloads where applications from different members do not access the same table.

- One member configurations or batch window workloads that use only a single member. A system is set up to have nightly batch processing with almost no OLTP activity. Because the batch workload is often run by a single application, it is the only one accessing tables and they can move into the NOT_SHARED state.

An application can be partitioned so that only certain tables are accessed by the connections to a single member. Using this partitioning approach, these tables move into the NOT_SHARED state when the **opt_direct_wrkld** configuration parameter is enabled.

EHL for directed workloads must avoid workload balancing (WLB). Instead, use client affinity and the member subsetting capability for directed workloads that do not use WLB.

Use cases for Explicit Hierarchical Locking (EHL)

Explicit hierarchical locking (EHL) for the IBM Db2 pureScale Feature is designed to improve performance by avoiding CF communications for tables that are accessed from only one member. When EHL is enabled, if only one member accesses a data table, partitioned table, or partitioned index then the table transition to the NOT_SHARED state.

Transitioning to this state is an ideal scenario for data tables, partitioned tables, or partitioned indexes. Grid deployments, partitioned or partitionable workloads with directed access, or batch window workloads that use only a single member are access patterns that can benefit from using EHL. The Db2 database server automatically detects these access patterns and transition applicable tables into the NOT_SHARED state.

Example 1: When the following sample SQL statements are issued, the Db2 database server detects that tables tab1, tab2, tab3, and tab4 are accessed by only one member and would benefit from EHL. These tables transition into the NOT_SHARED state.

```
Member 1:
db2 -v "select * from tab1"
db2 -v "delete from tab3 where col1 > 100"

Member 2:
db2 -v "insert into tab2 values (20,20)"
db2 -v "select * from tab4"
```

To ensure that tables remain in the NOT_SHARED state, tune your applications or use EHL for workloads where only a single member accesses a data table, partitioned table, or partitioned index.

Example 2: In the following example, the Db2 database server detects that the EHL optimization does not apply. Multiple members are all attempting to access the same table tab1. The table does not transition to the NOT_SHARED state.

```
Member 1:
db2 -v "select * from tab1"

Member 2:
db2 -v "insert into tab1 values (20,20)"

Member 1:
db2 -v "delete from tab1 where col1 > 100"

Member 2:
db2 -v "select * from tab1"
```

Use MON_GET_TABLE to monitor whether or not tables transition the NOT_SHARED state.

Explicit hierarchical locking state changes and performance implications

In a Db2 pureScale environment, regular tables, range partitioned tables, or partitioned indexes exist in the SHARED state, the NOT_SHARED state, or in transition between these two states.

Objects entering or already in NOT_SHARED state

A table might enter the NOT_SHARED state as a result of the following activity:

- If an INSERT, UPDATE or DELETE operation, or a scan is performed on a table.
- If a table is range partitioned, a data partition that is accessed by using the preceding actions might enter NOT_SHARED state, while the logical table is unaffected.
- The **CREATE INDEX** operation in non-partitioned indexes triggers a NOT_SHARED state transition of the logical table and the index anchors. If these indexes might enter the NOT_SHARED state independent of the data partitions.

As tables in a NOT_SHARED state are running in a mode similar to Db2 Enterprise Server Edition, they also have properties similar to Db2 Enterprise Server Edition tables. When a regular table is in the NOT_SHARED state, all of its table objects, such as data, index, LONG, LOB, XDA, are also in the NOT_SHARED state. However, only the table and row locks are behaving as in the NOT_SHARED state.

A range partitioned table can have its partitions enter or leave the NOT_SHARED state independently of each other, which is beneficial because not all transactions would access all the partitions on a table. So, partition independence allows members to access different partitions without conflict on the DATA_SHARING lock. Furthermore, all table objects (such partitioned indexes) within a partition also inherit properties as in the regular table case.

Note: Nonpartitioned indexes on a partitioned table all must enter or leave the NOT_SHARED state simultaneously. This simultaneous action is because all index anchor objects are under the protection of a single logical table lock. In other words, when a nonpartitioned index is being accessed its logical table lock enters the NOT_SHARED state, which causes all nonpartitioned indexes to enter the NOT_SHARED state as well. This behavior has no effect on the partitions since they have their own partition locks for protection.

Objects exiting the NOT_SHARED state

A table exits NOT_SHARED state as a result of the following activity:

- Any access to the table from another member
- Drop table or database deactivation
- ATTACH or DETACH partition for a partitioned table

Any access to a table from another member results in an exit from NOT_SHARED state on the first member to allow the Db2 database server to grant a lock on the table to the other member. When a table is moving out of NOT_SHARED to the SHARED state, its table lock is held in Z mode (super exclusive mode) until all

page locks and row locks are registered on the global lock manager (GLM) and all dirty buffer pool pages are written to the group buffer pool (GBP). This can be a lengthy process.

The Db2 database server can detect when a transition to NOT_SHARED state is not optimal, and it avoids the state change.

For member crash recovery, the entire table is unavailable until recovery is completed. When EHL is enabled, member crash recovery takes a length of time similar to crash recovery for an Db2 Enterprise Server Edition database. Time is required because changed data pages are not cached in the caching facility (CF), only in the local buffer pool. So modified pages must be rebuilt by replaying log records. Use the **page_age_trgt_mcr** database configuration parameter to control the length of time the pages remain buffered in the local buffer pool.

Exiting EHL is not immediate and involves CF communication for locks and pages for the table. Memory and time that is required for this operation is proportional to the number of locks and pages that are held for the object in the buffer pool. A small table typically takes only a few seconds to exit EHL. However, a very large table might take several minutes to exit EHL.

In extremely rare circumstances, it is possible for the GLM to become full during EHL exit and you are unable to send all required locks to the CF. This will prevent EHL exit from completing. If this condition occurs, other members will not be able to access this table until EHL exit is able to complete. This may result in lock time out events on other members accessing the table. When this condition is detected, and if the **CF_GBP_SZ** and **CF_LOCK_SZ** database configuration parameters are both configured to AUTOMATIC, the Db2 database server will attempt to trade memory from the group buffer pool (GBP) to the GLM, to allow lock registration to complete. The size of the GBP will be slightly reduced and the size of the GLM will be increased by this amount. There is a limit on the amount of GBP memory that can be traded in this way. However, if the **CF_GBP_SZ** an **CF_LOCK_SZ** database configuration parameters are not configured to AUTOMATIC, or if these actions do not free up enough GLM memory to allow EHL exit to complete within 40 seconds of detecting this condition, then all applications holding the lock that cannot be sent to the GLM will be forced. Forcing the applications will allow the lock to be released so that it does not need to be sent to the CF and this allows EHL exit to continue. ADM1504 will be logged when this memory trading occurs and ADM1503 will be logged for each application that is forced.

Monitoring EHL

EHL can be monitored using the following monitors and APIs:

- The LOCK WAIT event monitor
- MON_GET_DATABASE() administrative API
- MON_GET_TABLE() administrative API
- MON_GET_LOCKS() administrative API
- MON_GET_APPL_LOCKWAIT()administrative API

Query optimization

Query optimization is one of the factors that affect application performance. Review this section for details about query optimization considerations that can help you to maximize the performance of database applications.

The SQL and XQuery compiler process

The SQL and XQuery compiler performs several steps to produce an access plan that can be executed.

The *query graph model* is an internal, in-memory database that represents the query as it is processed through these steps, which are shown in Figure 24. Note that some steps occur only for queries that will run against a federated database.

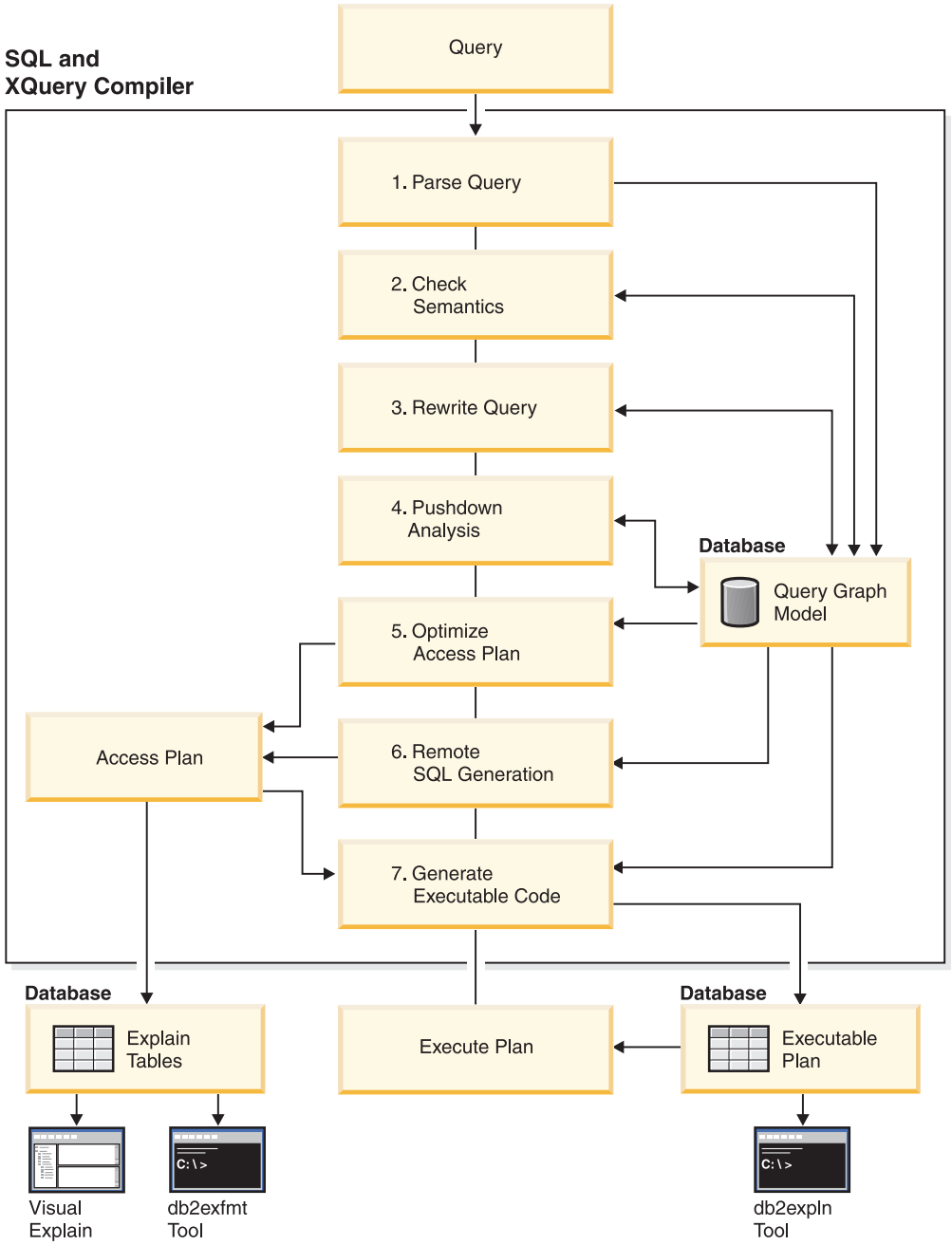


Figure 24. Steps performed by the SQL and XQuery compiler

1. Parse query

The SQL and XQuery compiler analyzes the query to validate the syntax. If any syntax errors are detected, the query compiler stops processing and returns an appropriate error to the application that submitted the query. When parsing is complete, an internal representation of the query is created and stored in the query graph model.

2. Check semantics

   The compiler ensures that there are no inconsistencies among parts of the statement. For example, the compiler verifies that a column specified for the YEAR scalar function has been defined with a datetime data type.

   The compiler also adds behavioral semantics to the query graph model, including the effects of referential constraints, table check constraints, triggers, and views. The query graph model contains all of the semantics for the query, including query blocks, subqueries, correlations, derived tables, expressions, data types, data type conversions, code page conversions, and distribution keys.

3. Rewrite query

   The compiler uses the global semantics that are stored in the query graph model to transform the query into a form that can be optimized more easily. It then stores the result in the query graph model.

   For example, the compiler might move a predicate, altering the level at which it is applied, thereby potentially improving query performance. This type of operation movement is called *general predicate pushdown*. In a partitioned database environment, the following query operations are more computationally intensive:

   - Aggregation
   - Redistribution of rows
   - Correlated subqueries, which are subqueries that contain a reference to a column in a table that is outside of the subquery

   For some queries in a partitioned database environment, decorrelation might occur as part of rewriting the query.

4. Pushdown analysis (federated databases only)

   The major task in this step is to suggest to the optimizer whether an operation can be remotely evaluated or *pushed down* at a data source. This type of pushdown activity is specific to data source queries and represents an extension to general predicate pushdown operations.

5. Optimize access plan

   Using the query graph model as input, the optimizer portion of the compiler generates many alternative execution plans for satisfying the query. To estimate the execution cost of each of these plans, the optimizer uses statistics for tables, indexes, columns and functions. It then chooses the plan with the smallest estimated execution cost. The optimizer uses the query graph model to analyze the query semantics and to obtain information about a wide variety of factors, including indexes, base tables, derived tables, subqueries, correlations, and recursion.

   The optimizer can also consider another type of pushdown operation, *aggregation and sort*, which can improve performance by pushing the evaluation of these operations to the Data Management Services (DMS) component.

   The optimizer also considers whether there are buffer pools of different sizes when determining page size selection. It considers whether the database is partitioned, or whether intraquery parallelism in a symmetric multiprocessor (SMP) environment is an option. This information is used by the optimizer to help select the best access plan for the query.

The output of this step is an access plan, and details about this access plan are captured in the explain tables. The information that is used to generate an access plan can be captured by an explain snapshot.

6. Remote SQL generation (federated databases only)

   The final plan that is selected by the optimizer might consist of a set of steps that operate on a remote data source. The remote SQL generation step creates an efficient SQL statement for operations that are performed by each data source, based on the SQL dialect at that data source.

7. Generate executable code

   In the final step, the compiler uses the access plan and the query graph model to create an executable access plan, or section, for the query. This code generation step uses information from the query graph model to avoid repetitive execution of expressions that need to be computed only once. This type of optimization is possible for code page conversions and when host variables are used.

   To enable query optimization or reoptimization of static or dynamic SQL or XQuery statements that have host variables, special registers, or parameter markers, bind the package with the REOPT bind option. The access path for a statement belonging to such a package, and containing host variables, special registers, or parameter markers, will be optimized using the values of these variables rather than default estimates that are chosen by the compiler. This optimization takes place at query execution time when the values are available.

   Information about access plans for static SQL and XQuery statements is stored in the system catalog tables. When a package is executed, the database manager uses the information that is stored in the system catalog to determine how to access the data and provide results for the query. This information is used by the **db2expln** tool.

Note: Execute the **RUNSTATS** command at appropriate intervals against tables that change often. The optimizer needs up-to-date statistical information about tables and their data to create the most efficient access plans. Rebind your application to take advantage of updated statistics. If **RUNSTATS** is not executed, or the optimizer assumes that this command was executed against empty or nearly empty tables, it might use default values or attempt to derive certain statistics based on the number of file pages that are used to store the table on disk. See also "Automatic statistics collection".

Query rewriting methods and examples:

During the query rewrite stage, the query compiler transforms SQL and XQuery statements into forms that can be optimized more easily; this can improve the possible access plans. Rewriting queries is particularly important for very complex queries, including those queries that have many subqueries or many joins. Query generator tools often create these types of very complex queries.

To influence the number of query rewrite rules that are applied to an SQL or XQuery statement, change the optimization class. To see some of the results of the query rewrite process, use the explain facility.

Queries might be rewritten in any one of the following ways:

- Operation merging

To construct a query so that it has the fewest number of operations, especially SELECT operations, the SQL and XQuery compiler rewrites queries to merge query operations. The following examples illustrate some of the operations that can be merged:

- Example - View merges

  A SELECT statement that uses views can restrict the join order of the table and can also introduce redundant joining of tables. If the views are merged during query rewrite, these restrictions can be lifted.

- Example - Subquery to join transforms

  If a SELECT statement contains a subquery, selection of order processing of the tables might be restricted.

- Example - Redundant join elimination

  During query rewrite, redundant joins can be removed to simplify the SELECT statement.

- Example - Shared aggregation

  When a query uses different functions, rewriting can reduce the number of calculations that need to be done.

- Operation movement

  To construct a query with the minimum number of operations and predicates, the compiler rewrites the query to move query operations. The following examples illustrate some of the operations that can be moved:

  - Example - DISTINCT elimination

    During query rewrite, the optimizer can move the point at which the DISTINCT operation is performed, to reduce the cost of this operation. In some cases, the DISTINCT operation can be removed completely.

  - Example - General predicate pushdown

    During query rewrite, the optimizer can change the order in which predicates are applied, so that more selective predicates are applied to the query as early as possible.

  - Example - Decorrelation

    In a partitioned database environment, the movement of result sets among database partitions is costly. Reducing the size of what must be broadcast to other database partitions, or reducing the number of broadcasts, or both, is an objective of the query rewriting process.

- Predicate translation

  The SQL and XQuery compiler rewrites queries to translate existing predicates into more optimal forms. The following examples illustrate some of the predicates that might be translated:

  - Example - Addition of implied predicates

    During query rewrite, predicates can be added to a query to enable the optimizer to consider additional table joins when selecting the best access plan for the query.

  - Example - OR to IN transformations

    During query rewrite, an OR predicate can be translated into an IN predicate for a more efficient access plan. The SQL and XQuery compiler can also translate an IN predicate into an OR predicate if this transformation would create a more efficient access plan.

Compiler rewrite example: Operation merging:

A SELECT statement that uses views can restrict the join order of the table and can also introduce redundant joining of tables. If the views are merged during query rewrite, these restrictions can be lifted.

Suppose you have access to the following two views that are based on the EMPLOYEE table: one showing employees that have a high level of education and the other showing employees that earn more than $35,000 per year:

```
create view emp_education (empno, firstnme, lastname, edlevel) as
  select empno, firstnme, lastname, edlevel
    from employee
    where edlevel > 17

create view emp_salaries (empno, firstname, lastname, salary) as
  select empno, firstnme, lastname, salary
    from employee
    where salary > 35000
```

The following user-written query lists those employees who have a high level of education and who earn more than $35,000 per year:

```
select e1.empno, e1.firstnme, e1.lastname, e1.edlevel, e2.salary
  from emp_education e1, emp_salaries  e2
  where e1.empno = e2.empno
```

During query rewrite, these two views could be merged to create the following query:

```
select e1.empno, e1.firstnme, e1.lastname, e1.edlevel, e2.salary
  from employee e1, employee e2
  where
    e1.empno = e2.empno and
    e1.edlevel > 17 and
    e2.salary  > 35000
```

By merging the SELECT statements from the two views with the user-written SELECT statement, the optimizer can consider more choices when selecting an access plan. In addition, if the two views that have been merged use the same base table, additional rewriting might be performed.

Example - Subquery to join transformations

The SQL and XQuery compiler will take a query containing a subquery, such as:

```
select empno, firstnme, lastname, phoneno
  from employee
  where workdept in
    (select deptno
      from department
      where deptname = 'OPERATIONS')
```

and convert it to a join query of the form:

```
select distinct empno, firstnme, lastname, phoneno
  from employee emp, department dept
  where
    emp.workdept = dept.deptno and
    dept.deptname = 'OPERATIONS'
```

A join is generally much more efficient to execute than a subquery.

Example - Redundant join elimination

Queries sometimes have unnecessary joins.

```
select e1.empno, e1.firstnme, e1.lastname, e1.edlevel, e2.salary
  from employee e1,
       employee e2
 where e1.empno = e2.empno
   and e1.edlevel > 17
   and e2.salary  > 35000
```

The SQL and XQuery compiler can eliminate the join and simplify the query to:

```
select empno, firstnme, lastname, edlevel, salary
  from employee
  where
    edlevel > 17 and
    salary  > 35000
```

The following example assumes that a referential constraint exists between the EMPLOYEE and DEPARTMENT tables on the department number. First, a view is created.

```
create view peplview as
  select firstnme, lastname, salary, deptno, deptname, mgrno
    from employee e department d
    where e.workdept = d.deptno
```

Then a query such as the following:

```
select lastname, salary
  from peplview
```

becomes:

```
select lastname, salary
  from employee
  where workdept not null
```

Note that in this situation, even if you know that the query can be rewritten, you might not be able to do so because you do not have access to the underlying tables. You might only have access to the view (shown previously). Therefore, this type of optimization has to be performed by the database manager.

Redundancy in referential integrity joins likely occurs when:

- Views are defined with joins
- Queries are automatically generated

Example - Shared aggregation

Using multiple functions within a query can generate several calculations that take time. Reducing the number of required calculations improves the plan. The compiler takes a query that uses multiple functions, such as the following:

```
select sum(salary+bonus+comm) as osum,
    avg(salary+bonus+comm) as oavg,
    count(*) as ocount
  from employee
```

and transforms it:

```
select osum, osum/ocount ocount
  from (
    select sum(salary+bonus+comm) as osum,
        count(*) as ocount
      from employee
  ) as shared_agg
```

This rewrite halves the required number of sums and counts.

Compiler rewrite example: Operation movement:

During query rewrite, the optimizer can move the point at which the DISTINCT operation is performed, to reduce the cost of this operation. In some cases, the DISTINCT operation can be removed completely.

For example, if the EMPNO column of the EMPLOYEE table were defined as the primary key, the following query:

```
select distinct empno, firstnme, lastname
  from employee
```

could be rewritten by removing the DISTINCT clause:

```
select empno, firstnme, lastname
  from employee
```

In this example, because the primary key is being selected, the compiler knows that each returned row is unique. In this case, the DISTINCT keyword is redundant. If the query is not rewritten, the optimizer must build a plan with necessary processing (such as a sort) to ensure that the column values are unique.

Example - General predicate pushdown

Altering the level at which a predicate is normally applied can result in improved performance. For example, the following view provides a list of all employees in department D11:

```
create view d11_employee
  (empno, firstnme, lastname, phoneno, salary, bonus, comm) as
    select empno, firstnme, lastname, phoneno, salary, bonus, comm
      from employee
      where workdept = 'D11'
```

The following query against this view is not as efficient as it could be:

```
select firstnme, phoneno
  from d11_employee
  where lastname = 'BROWN'
```

During query rewrite, the compiler pushes the `lastname = 'BROWN'` predicate down into the D11_EMPLOYEE view. This allows the predicate to be applied sooner and potentially more efficiently. The actual query that could be executed in this example is as follows:

```
select firstnme, phoneno
  from employee
  where
    lastname = 'BROWN' and
    workdept = 'D11'
```

Predicate pushdown is not limited to views. Other situations in which predicates can be pushed down include UNION, GROUP BY, and derived tables (nested table expressions or common table expressions).

Example - Decorrelation

In a partitioned database environment, the compiler can rewrite the following query, which is designed to find all of the employees who are working on programming projects and who are underpaid.

```
select p.projno, e.empno, e.lastname, e.firstname,
    e.salary+e.bonus+e.comm as compensation
  from employee e, project p
  where
    p.empno = e.empno and
    p.projname like '%PROGRAMMING%' and
    e.salary+e.bonus+e.comm <
      (select avg(e1.salary+e1.bonus+e1.comm)
        from employee e1, project p1
        where
          p1.projname like '%PROGRAMMING%' and
          p1.projno = p.projno and
          e1.empno = p1.empno)
```

Because this query is correlated, and because both PROJECT and EMPLOYEE are unlikely to be partitioned on PROJNO, the broadcasting of each project to each database partition is possible. In addition, the subquery would have to be evaluated many times.

The compiler can rewrite the query as follows:

- Determine the distinct list of employees working on programming projects and call it DIST_PROJS. It must be distinct to ensure that aggregation is done only once for each project:

  ```
  with dist_projs(projno, empno) as
    (select distinct projno, empno
      from project p1
      where p1.projname like '%PROGRAMMING%')
  ```

- Join DIST_PROJS with the EMPLOYEE table to get the average compensation per project, AVG_PER_PROJ:

  ```
  avg_per_proj(projno, avg_comp) as
    (select p2.projno, avg(e1.salary+e1.bonus+e1.comm)
      from employee e1, dist_projs p2
      where e1.empno = p2.empno
      group by p2.projno)
  ```

- The rewritten query is:

  ```
  select p.projno, e.empno, e.lastname, e.firstname,
      e.salary+e.bonus+e.comm as compensation
    from project p, employee e, avg_per_prog a
    where
      p.empno = e.empno and
      p.projname like '%PROGRAMMING%' and
      p.projno = a.projno and
      e.salary+e.bonus+e.comm < a.avg_comp
  ```

This query computes the `avg_comp` per project (`avg_per_proj`). The result can then be broadcast to all database partitions that contain the EMPLOYEE table.

Compiler rewrite example: Operation movement - Predicate pushdown for combined SQL/XQuery statements:

One fundamental technique for the optimization of relational SQL queries is to move predicates in the WHERE clause of an enclosing query block into an enclosed lower query block (for example, a view), thereby enabling early data filtering and potentially better index usage.

This is even more important in partitioned database environments, because early filtering potentially reduces the amount of data that must be shipped between database partitions.

Similar techniques can be used to move predicates or XPath filters inside of an XQuery. The basic strategy is always to move filtering expressions as close to the data source as possible. This optimization technique is called *predicate pushdown* in SQL and *extraction pushdown* (for filters and XPath extractions) in XQuery.

Because the data models employed by SQL and XQuery are different, you must move predicates, filters, or extractions across the boundary between the two languages. Data mapping and casting rules have to be considered when transforming an SQL predicate into a semantically equivalent filter and pushing it down into the XPath extraction. The following examples address the pushdown of relation predicates into XQuery query blocks.

Consider the following two XML documents containing customer information:

```
Document 1                    Document 2

<customer>                    <customer>
  <name>John</name>             <name>Michael</name>
  <lastname>Doe</lastname>      <lastname>Miller </lastname>
  <date_of_birth>               <date_of_birth>
   1976-10-10                   1975-01-01
  </date_of_birth>              </date_of_birth>
  <address>                     <address>
    <zip>95141.0</zip>            <zip>95142.0</zip>
  </address>                    </address>
<volume>80000.0</volume>      <volume>100000.00</volume>
</customer>                   </customer>
<customer>                    <customer>
  <name>Jane</name>             <name>Michaela</name>
  <lastname>Doe</lastname>      <lastname>Miller</lastname>
  <date_of_birth>               <date_of_birth>
    1975-01-01                   1980-12-23
  </date_of_birth>              </date_of_birth>
  <address>                     <address>
    <zip>95141.4</zip>            <zip>95140.5</zip>
  </address>                    </address>
<volume>50000.00</volume>     <volume>100000</volume>
</customer>                   </customer>
```

Example - Pushing INTEGER predicates

Consider the following query:

```
select temp.name, temp.zip
  from xmltable('db2-fn:xmlcolumn("T.XMLDOC")'
    columns name varchar(20) path 'customer/name',
    zip integer path 'customer/zip'
    ) as temp
  where zip = 95141
```

To use possible indexes on T.XMLDOC and to filter out records that are not needed early on, the `zip = 95141` predicate will be internally converted into the following equivalent XPATH filtering expression:

```
T.XMLCOL/customer/zip[. >= 95141.0 and . < 95142.0]
```

Because schema information for XML fragments is not used by the compiler, it cannot be assumed that ZIP contains integers only. It is possible that there are other numeric values with a fractional part and a corresponding double XML index

on this specific XPath extraction. The XML2SQL cast would handle this transformation by truncating the fractional part before casting the value to INTEGER. This behavior must be reflected in the pushdown procedure, and the predicate must be changed to remain semantically correct.

Example - Pushing DECIMAL(*x*,*y*) predicates

Consider the following query:

```
select temp.name, temp.volume
  from xmltable('db2-fn:xmlcolumn("T.XMLDOC")'
    columns name varchar(20) path 'customer/name',
    volume decimal(10,2) path 'customer/volume'
    ) as temp
  where volume = 100000.00
```

To use possible DECIMAL or DOUBLE indexes on T.XMLDOC and to filter out records that are not needed early on, similar to the handling for the INTEGER type, the `volume = 100000.00` predicate will be internally converted into the following XPATH range filtering expression:

```
T.XMLCOL/customer/volume[.>=100000.00 and .<100000.01]
```

Example - Pushing VARCHAR(*n*) predicates

Consider the following query:

```
select temp.name, temp.lastname
  from xmltable('db2-fn:xmlcolumn("T.XMLDOC")'
    columns name varchar(20) path 'customer/name',
    lastname varchar(20) path 'customer/lastname'
    ) as temp
  where lastname = 'Miller'
```

To use possible indexes on T.XMLDOC and to filter out records that are not needed early on, the `lastname = 'Miller'` predicate will be internally converted into an equivalent XPATH filtering expression. A high-level representation of this expression is: :

```
T.XMLCOL/customer/lastname[. >= rtrim("Miller") and . <
   RangeUpperBound("Miller", 20)]
```

Trailing blanks are treated differently in SQL than in XPath or XQuery. The original SQL predicate will not distinguish between the two customers whose last name is “Miller”, even if one of them (Michael) has a trailing blank. Consequently, both customers are returned, which would not be the case if an unchanged predicate were pushed down.

The solution is to transform the predicate into a range filter.

- The first boundary is created by truncating all trailing blanks from the comparison value, using the RTRIM() function.
- The second boundary is created by looking up all possible strings that are greater than or equal to “Miller”, so that all strings that begin with “Miller” are located. Therefore, the original string is replaced with an upperbound string that represents this second boundary.

Compiler rewrite example: Predicate translation:

During query rewrite, predicates can be added to a query to enable the optimizer to consider additional table joins when selecting the best access plan for the query.

The following query returns a list of the managers whose departments report to department E01, and the projects for which those managers are responsible:

```
select dept.deptname dept.mgrno, emp.lastname, proj.projname
  from department dept, employee emp, project proj
  where
    dept.admrdept = 'E01' and
    dept.mgrno = emp.empno and
    emp.empno = proj.respemp
```

This query can be rewritten with the following implied predicate, known as a *predicate for transitive closure*:

```
dept.mgrno = proj.respemp
```

The optimizer can now consider additional joins when it tries to select the best access plan for the query.

During the query rewrite stage, additional local predicates are derived on the basis of the transitivity that is implied by equality predicates. For example, the following query returns the names of the departments whose department number is greater than E00, and the employees who work in those departments.

```
select empno, lastname, firstname, deptno, deptname
  from employee emp, department dept
  where
    emp.workdept = dept.deptno and
    dept.deptno > 'E00'
```

This query can be rewritten with the following implied predicate:

```
emp.workdept > 'E00'
```

This rewrite reduces the number of rows that need to be joined.

Example - OR to IN transformations

Suppose that an OR clause connects two or more simple equality predicates on the same column, as in the following example:

```
select *
  from employee
  where
    deptno = 'D11' or
    deptno = 'D21' or
    deptno = 'E21'
```

If there is no index on the DEPTNO column, using an IN predicate in place of OR causes the query to be processed more efficiently:

```
select *
  from employee
  where deptno in ('D11', 'D21', 'E21')
```

In some cases, the database manager might convert an IN predicate into a set of OR clauses so that index ORing can be performed.

Predicate typology and access plans:

A *predicate* is an element of a search condition that expresses or implies a comparison operation. Predicates, which usually appear in the WHERE clause of a query, are used to reduce the scope of the result set that is returned by the query.

Predicates can be grouped into four categories, depending on how and when they are used in the evaluation process. The following list ranks these categories in order of best to worst performance:

1. Range-delimiting predicates
2. Index SARGable predicates
3. Data SARGable predicates
4. Residual predicates

A *SARGable* term is a term that can be used as a *s*earch *arg*ument.

Table 55 summarizes the characteristics of these predicate categories.

Table 55. Summary of Predicate Type Characteristics

Characteristic	Predicate Type			
	Range-delimiting	Index-SARGable	Data-SARGable	Residual
Reduce index I/O	Yes	No	No	No
Reduce data page I/O	Yes	Yes	No	No
Reduce the number of rows that are passed internally	Yes	Yes	Yes	No
Reduce the number of qualifying rows	Yes	Yes	Yes	Yes

Range-delimiting and index-SARGable predicates

Range-delimiting predicates limit the scope of an index scan. They provide start and stop key values for the index search. Index-SARGable predicates cannot limit the scope of a search, but can be evaluated from the index, because the columns that are referenced in the predicate are part of the index key. For example, consider the following index:

```
INDEX IX1:  NAME    ASC,
            DEPT    ASC,
            MGR     DESC,
            SALARY  DESC,
            YEARS   ASC
```

Consider also a query that contains the following WHERE clause:

```
where
  name = :hv1 and
  dept = :hv2 and
  years > :hv5
```

The first two predicates (`name = :hv1 and dept = :hv2`) are range-delimiting predicates, and `years > :hv5` is an index-SARGable predicate.

Attention: When a jump scan or skip scan is applied, for performance purposes, `years > :hv5` is treated as both a range-delimited predicate and an index-SARGable predicate.

The optimizer uses index data instead of reading the base table when it evaluates these predicates. Index-SARGable predicates reduce the number of rows that need to be read from the table, but they do not affect the number of index pages that are accessed.

Data SARGable predicates

Predicates that cannot be evaluated by the index manager, but that can be evaluated by Data Management Services (DMS), are called data-SARGable predicates. These predicates usually require access to individual rows in a table. If required, DMS retrieves the columns that are needed to evaluate a predicate, as well as any other columns that are needed for the SELECT list, but that could not be obtained from the index.

For example, consider the following index that is defined on the PROJECT table:

```
INDEX IX0:  PROJNO ASC
```

In the following query, `deptno = 'D11'` is considered to be a data-SARGable predicate.

```
select projno, projname, respemp
  from project
  where deptno = 'D11'
  order by projno
```

Residual predicates

Residual predicates are more expensive, in terms of I/O cost, than accessing a table. Such predicates might:

- Use correlated subqueries
- Use quantified subqueries, which contain ANY, ALL, SOME, or IN clauses
- Read LONG VARCHAR or LOB data, which is stored in a file that is separate from the table

Such predicates are evaluated by Relational Data Services (RDS).

Some predicates that are applied only to an index must be reapplied when the data page is accessed. For example, access plans that use index ORing or index ANDing always reapply the predicates as residual predicates when the data page is accessed.

Federated database query-compiler phases:

In a federated database, there are additional steps that are taken by the query compiler. These additional steps are needed to determine which remote data sources to use to improve query performance.

Global analysis on the federated database might result in information that can be used to optimize the federated environment overall.

Federated database pushdown analysis:

For queries that are to run against federated databases, the optimizer performs pushdown analysis to determine whether a particular operation can be performed at a remote data source.

An operation might be a function, such as a relational operator, or a system or user function; or it might be an SQL operator, such as, for example, ORDER BY or GROUP BY.

Be sure to update local catalog information regularly, so that the Db2 query compiler has access to accurate information about SQL support at remote data sources. Use Db2 data definition language (DDL) statements (such as CREATE FUNCTION MAPPING or ALTER SERVER, for example) to update the catalog.

If functions cannot be pushed down to the remote data source, they can significantly impact query performance. Consider the effect of forcing a selective predicate to be evaluated locally instead of at the data source. Such evaluation could require the Db2 server to retrieve the entire table from the remote data source and then filter it locally against the predicate. Network constraints and a large table could cause performance to suffer.

Operators that are not pushed down can also significantly affect query performance. For example, having a GROUP BY operator aggregate remote data locally could also require the Db2 server to retrieve an entire table from the remote data source.

For example, consider nickname N1, which references the data source table EMPLOYEE in a Db2 for z/OS® data source. The table has 10 000 rows, one of the columns contains the last names of employees, and one of the columns contains salaries. The optimizer has several options when processing the following statement, depending on whether the local and remote collating sequences are the same:

```
select lastname, count(*)  from n1
  where
    lastname > 'B' and
    salary > 50000
  group by lastname
```

- If the collating sequences are the same, the query predicates can probably be pushed down to Db2 for z/OS. Filtering and grouping results at the data source is usually more efficient than copying the entire table and performing the operations locally. For this query, the predicates and the GROUP BY operation can take place at the data source.
- If the collating sequences are not the same, both predicates cannot be evaluated at the data source. However, the optimizer might decide to push down the `salary > 50000` predicate. The range comparison must still be done locally.
- If the collating sequences are the same, and the optimizer knows that the local Db2 server is very fast, the optimizer might decide that performing the GROUP BY operation locally is the least expensive approach. The predicate is evaluated at the data source. This is an example of pushdown analysis combined with global optimization.

In general, the goal is to ensure that the optimizer evaluates functions and operators at remote data sources. Many factors affect whether a function or an SQL operator can be evaluated at a remote data source, including the following:

- Server characteristics
- Nickname characteristics
- Query characteristics

Server characteristics that affect pushdown opportunities

Certain data source-specific factors can affect pushdown opportunities. In general, these factors exist because of the rich SQL dialect that is supported by the Db2 product. The Db2 data server can compensate for the lack of function that is available at another data server, but doing so might require that the operation take place at the Db2 server.

- SQL capabilities

  Each data source supports a variation of the SQL dialect and different levels of functionality. For example, most data sources support the GROUP BY operator, but some limit the number of items on the GROUP BY list, or have restrictions on whether an expression is allowed on the GROUP BY list. If there is a restriction at the remote data source, the Db2 server might have to perform a GROUP BY operation locally.

- SQL restrictions

  Each data source might have different SQL restrictions. For example, some data sources require parameter markers to bind values to remote SQL statements. Therefore, parameter marker restrictions must be checked to ensure that each data source can support such a bind mechanism. If the Db2 server cannot determine a good method to bind a value for a function, this function must be evaluated locally.

- SQL limits

  Although the Db2 server might allow the use of larger integers than those that are permitted on remote data sources, values that exceed remote limits cannot be embedded in statements that are sent to data sources, and any impacted functions or operators must be evaluated locally.

- Server specifics

  Several factors fall into this category. For example, if null values at a data source are sorted differently from how the Db2 server would sort them, ORDER BY operations on a nullable expression cannot be remotely evaluated.

- Collating sequence

  Retrieving data for local sorts and comparisons usually decreases performance. If you configure a federated database to use the same collating sequence that a data source uses and then set the COLLATING_SEQUENCE server option to Y, the optimizer can consider pushing down many query operations. The following operations might be pushed down if collating sequences are the same:

  – Comparisons of character or numeric data
  – Character range comparison predicates
  – Sorts

  You might get unusual results, however, if the weighting of null characters is different between the federated database and the data source. Comparisons might return unexpected results if you submit statements to a case-insensitive data source. The weights that are assigned to the characters “I” and “i” in a case-insensitive data source are the same. The Db2 server, by default, is case sensitive and assigns different weights to these characters.

  To improve performance, the federated server allows sorts and comparisons to take place at data sources. For example, in Db2 for z/OS, sorts that are defined by ORDER BY clauses are implemented by a collating sequence that is based on an EBCDIC code page. To use the federated server to retrieve Db2 for z/OS data that is sorted in accordance with ORDER BY clauses, configure the federated database so that it uses a predefined collating sequence based on the EBCDIC code page.

If the collating sequences of the federated database and the data source differ, the Db2 server retrieves the data to the federated database. Because users expect to see query results ordered by the collating sequence that is defined for the federated server, ordering the data locally ensures that this expectation is fulfilled. Submit your query in passthrough mode, or define the query in a data source view if you need to see the data ordered in the collating sequence of the data source.

- Server options

  Several server options can affect pushdown opportunities, including COLLATING_SEQUENCE, VARCHAR_NO_TRAILING_BLANKS, and PUSHDOWN.

- Db2 type mapping and function mapping factors

  The default local data type mappings on the Db2 server are designed to provide sufficient buffer space for each data source data type, which avoids loss of data. You can customize the type mapping for a specific data source to suit specific applications. For example, if you are accessing an Oracle data source column with a DATE data type, which by default is mapped to the Db2 TIMESTAMP data type, you might change the local data type to the Db2 DATE data type.

In the following three cases, the Db2 server can compensate for functions that a data source does not support:

- The function does not exist at the remote data source.
- The function exists, but the characteristics of the operand violate function restrictions. The IS NULL relational operator is an example of this situation. Most data sources support it, but some might have restrictions, such as allowing a column name to appear only on the left hand side of the IS NULL operator.
- The function might return a different result if it is evaluated remotely. An example of this situation is the greater than ('>') operator. For data sources with different collating sequences, the greater than operator might return different results if it is evaluated locally by the Db2 server.

Nickname characteristics that affect pushdown opportunities

The following nickname-specific factors can affect pushdown opportunities.

- Local data type of a nickname column

  Ensure that the local data type of a column does not prevent a predicate from being evaluated at the data source. Use the default data type mappings to avoid possible overflow. However, a joining predicate between two columns of different lengths might not be considered at a data source whose joining column is shorter, depending on how Db2 binds the longer column. This situation can affect the number of possibilities that the Db2 optimizer can evaluate in a joining sequence. For example, Oracle data source columns that were created using the INTEGER or INT data type are given the type NUMBER(38). A nickname column for this Oracle data type is given the local data type FLOAT, because the range of a Db2 integer is from 2**31 to (-2**31)-1, which is roughly equivalent to NUMBER(9). In this case, joins between a Db2 integer column and an Oracle integer column cannot take place at the Db2 data source (because of the shorter joining column); however, if the domain of this Oracle integer column can be accommodated by the Db2 INTEGER data type, change its local data type with the ALTER NICKNAME statement so that the join can take place at the Db2 data source.

- Column options

Use the ALTER NICKNAME statement to add or change column options for nicknames.

Use the VARCHAR_NO_TRAILING_BLANKS option to identify a column that contains no trailing blanks. The compiler pushdown analysis step will then take this information into account when checking all operations that are performed on such columns. The Db2 server might generate a different but equivalent form of a predicate to be used in the SQL statement that is sent to a data source. You might see a different predicate being evaluated against the data source, but the net result should be equivalent.

Use the NUMERIC_STRING option to indicate whether the values in that column are always numbers without trailing blanks.

Table 56 describes these options.

Table 56. Column Options and Their Settings

Option	Valid Settings	Default Setting
NUMERIC_STRING	Y: Specifies that this column contains only strings of numeric data. It does not contain blank characters that could interfere with sorting of the column data. This option is useful when the collating sequence of a data source is different from that of the Db2 server. Columns that are marked with this option are not excluded from local (data source) evaluation because of a different collating sequence. If the column contains only numeric strings that are followed by trailing blank characters, do not specify Y. N: Specifies that this column is not limited to strings of numeric data.	N
VARCHAR_NO_ TRAILING_BLANKS	Y: Specifies that this data source uses non-blank-padded VARCHAR comparison semantics, similar to the Db2 data server. For variable-length character strings that contain no trailing blank characters, non-blank-padded comparison semantics of some data servers return the same results as Db2 comparison semantics. Specify this value if you are certain that all VARCHAR table or view columns at a data source contain no trailing blank characters N: Specifies that this data source does not use non-blank-padded VARCHAR comparison semantics, similar to the Db2 data server.	N

Query characteristics that affect pushdown opportunities

A query can reference an SQL operator that might involve nicknames from multiple data sources. The operation must take place on the Db2 server to combine the results from two referenced data sources that use one operator, such as a set operator (for example, UNION). The operator cannot be evaluated at a remote data source directly.

Guidelines for determining where a federated query is evaluated:

The Db2 explain utility, which you can start by invoking the **db2expln** command, shows where queries are evaluated. The execution location for each operator is included in the command output.

- If a query is pushed down, you should see a RETURN operator, which is a standard Db2 operator. If a SELECT statement retrieves data from a nickname, you also see a SHIP operator, which is unique to federated database operations: it changes the server property of the data flow and separates local operators from remote operators. The SELECT statement is generated using the SQL dialect that is supported by the data source.
- If an INSERT, UPDATE, or DELETE statement can be entirely pushed down to the remote data source, you might not see a SHIP operator in the access plan. All remotely executed INSERT, UPDATE, or DELETE statements are shown for the RETURN operator. However, if a query cannot be pushed down in its entirety, the SHIP operator shows which operations were performed remotely.

Understanding why a query is evaluated at a data source instead of by the Db2 server

Consider the following key questions when you investigate ways to increase pushdown opportunities:

- Why isn't this predicate being evaluated remotely?

  This question arises when a very selective predicate could be used to filter rows and reduce network traffic. Remote predicate evaluation also affects whether a join between two tables of the same data source can be evaluated remotely.

  Areas to examine include:

  – Subquery predicates. Does this predicate contain a subquery that pertains to another data source? Does this predicate contain a subquery that involves an SQL operator that is not supported by this data source? Not all data sources support set operators in a subquery predicate.
  – Predicate functions. Does this predicate contain a function that cannot be evaluated by this remote data source? Relational operators are classified as functions.
  – Predicate bind requirements. If it is remotely evaluated, does this predicate require bind-in of some value? Would that violate SQL restrictions at this data source?
  – Global optimization. The optimizer might have decided that local processing is more cost effective.
- Why isn't the GROUP BY operator evaluated remotely?

  Areas to examine include:

  – Is the input to the GROUP BY operator evaluated remotely? If the answer is no, examine the input.
  – Does the data source have any restrictions on this operator? Examples include:
    - A limited number of GROUP BY items
    - Limited byte counts for combined GROUP BY items
    - Column specifications only on the GROUP BY list
  – Does the data source support this SQL operator?
  – Global optimization. The optimizer might have decided that local processing is more cost effective.
  – Does the GROUP BY clause contain a character expression? If it does, verify that the remote data source and the Db2 server have the same case sensitivity.
- Why isn't the set operator evaluated remotely?

  Areas to examine include:

  - Are both of its operands evaluated in their entirety at the same remote data source? If the answer is no, and it should be yes, examine each operand.
  - Does the data source have any restrictions on this set operator? For example, are large objects (LOBs) or LONG field data valid input for this specific set operator?
- Why isn't the ORDER BY operation evaluated remotely?

  Areas to examine include:
  - Is the input to the ORDER BY operation evaluated remotely? If the answer is no, examine the input.
  - Does the ORDER BY clause contain a character expression? If yes, do the remote data source and the Db2 server have different collating sequences or case sensitivities?
  - Does the remote data source have any restrictions on this operator? For example, is there a limit to the number of ORDER BY items? Does the remote data source restrict column specification to the ORDER BY list?

Remote SQL generation and global optimization in federated databases:

For a federated database query that uses relational nicknames, the access strategy might involve breaking down the original query into a set of remote query units and then combining the results. Such remote SQL generation helps to produce a globally optimized access strategy for a query.

The optimizer uses the output of pushdown analysis to decide whether each operation is to be evaluated locally at the Db2 server or remotely at a data source. It bases its decision on the output of its cost model, which includes not only the cost of evaluating the operation, but also the cost of shipping the data and messages between the Db2 server and the remote data source.

Although the goal is to produce an optimized query, the following factors significantly affect global optimization, and thereby query performance.

- Server characteristics
- Nickname characteristics

Server options that affect global optimization

The following data source server options can affect global optimization:

- Relative ratio of processing speed

  Use the CPU_RATIO server option to specify how fast or slow the processing speed at the data source should be relative to the processing speed at the Db2 server. A low ratio indicates that the processing speed at the data source is faster than the processing speed at the Db2 server; in this case, the Db2 optimizer is more likely to consider pushing processor-intensive operations down to the data source.
- Relative ratio of I/O speed

  Use the IO_RATIO server option to specify how fast or slow the system I/O speed at the data source should be relative to the system I/O speed at the Db2 server. A low ratio indicates that the I/O speed at the data source is faster than the I/O speed at the Db2 server; in this case, the Db2 optimizer is more likely to consider pushing I/O-intensive operations down to the data source.
- Communication rate between the Db2 server and the data source

  Use the COMM_RATE server option to specify network capacity. Low rates, which indicate slow network communication between the Db2 server and a data

source, encourage the Db2 optimizer to reduce the number of messages that are sent to or from this data source. If the rate is set to 0, the optimizer creates an access plan that requires minimal network traffic.

- Data source collating sequence

  Use the COLLATING_SEQUENCE server option to specify whether a data source collating sequence matches the local Db2 database collating sequence. If this option is not set to Y, the Db2 optimizer considers any data that is retrieved from this data source as being unordered.

- Remote plan hints

  Use the PLAN_HINTS server option to specify that plan hints should be generated or used at a data source. By default, the Db2 server does not send any plan hints to the data source.

  Plan hints are statement fragments that provide extra information to the optimizer at a data source. For some queries, this information can improve performance. The plan hints can help the optimizer at a data source to decide whether to use an index, which index to use, or which table join sequence to use.

  If plan hints are enabled, the query that is sent to the data source contains additional information. For example, a statement with plan hints that is sent to an Oracle optimizer might look like this:

  ```
  select /*+ INDEX (table1, t1index)*/
      col1
    from table1
  ```

  The plan hint is the string: `/*+ INDEX (table1, t1index)*/`

- Information in the Db2 optimizer knowledge base

  The Db2 server has an optimizer knowledge base that contains data about native data sources. The Db2 optimizer does not generate remote access plans that cannot be generated by specific database management systems (DBMSs). In other words, the Db2 server avoids generating plans that optimizers at remote data sources cannot understand or accept.

Nickname characteristics that affect global optimization

The following nickname-specific factors can affect global optimization.

- Index considerations

  To optimize queries, the Db2 server can use information about indexes at data sources. For this reason, it is important that the available index information be current. Index information for a nickname is initially acquired when the nickname is created. Index information is not collected for view nicknames.

- Creating index specifications on nicknames

  You can create an index specification for a nickname. Index specifications build an index definition (not an actual index) in the catalog for the Db2 optimizer to use. Use the CREATE INDEX SPECIFICATION ONLY statement to create index specifications. The syntax for creating an index specification on a nickname is similar to the syntax for creating an index on a local table. Consider creating index specifications in the following circumstances:

  – When the Db2 server cannot retrieve any index information from a data source during nickname creation
  – When you want an index for a view nickname

– When you want to encourage the Db2 optimizer to use a specific nickname as the inner table of a nested-loop join. You can create an index on the joining column, if none exists.

Before you issue CREATE INDEX statements against a nickname for a view, consider whether you need one. If the view is a simple SELECT on a table with an index, creating local indexes on the nickname to match the indexes on the table at the data source can significantly improve query performance. However, if indexes are created locally over a view that is not a simple SELECT statement, such as a view that is created by joining two tables, query performance might suffer. For example, if you create an index over a view that is a join between two tables, the optimizer might choose that view as the inner element in a nested-loop join. The query will perform poorly, because the join is evaluated several times. An alternate approach is to create nicknames for each of the tables that are referenced in the data source view, and then to create a local view at the Db2 server that references both nicknames.

- Catalog statistics considerations

  System catalog statistics describe the overall size of nicknames and the range of values in associated columns. The optimizer uses these statistics when it calculates the least-cost path for processing queries that contain nicknames. Nickname statistics are stored in the same catalog views as table statistics.

  Although the Db2 server can retrieve the statistical data that is stored at a data source, it cannot automatically detect updates to that data. Furthermore, the Db2 server cannot automatically detect changes to the definition of objects at a data source. If the statistical data for-or the definition of-an object has changed, you can:

  – Run the equivalent of a **RUNSTATS** command at the data source, drop the current nickname, and then recreate it. Use this approach if an object's definition has changed.
  – Manually update the statistics in the SYSSTAT.TABLES catalog view. This approach requires fewer steps, but it does not work if an object's definition has changed.

Global analysis of federated database queries:

The Db2 explain utility, which you can start by invoking the **db2expln** command, shows the access plan that is generated by the remote optimizer for those data sources that are supported by the remote explain function. The execution location for each operator is included in the command output.

You can also find the remote SQL statement that was generated for each data source in the SHIP or RETURN operator, depending on the type of query. By examining the details for each operator, you can see the number of rows that were estimated by the Db2 optimizer as input to and output from each operator.

Understanding Db2 optimization decisions

Consider the following key questions when you investigate ways to increase performance:

- Why isn't a join between two nicknames of the same data source being evaluated remotely?

  Areas to examine include:

  – Join operations. Can the remote data source support them?
  – Join predicates. Can the join predicate be evaluated at the remote data source? If the answer is no, examine the join predicate.

- Why isn't the GROUP BY operator being evaluated remotely?

  Examine the operator syntax, and verify that the operator can be evaluated at the remote data source.
- Why is the statement not being completely evaluated by the remote data source?

  The Db2 optimizer performs cost-based optimization. Even if pushdown analysis indicates that every operator can be evaluated at the remote data source, the optimizer relies on its cost estimate to generate a global optimization plan. There are a great many factors that can contribute to that plan. For example, even though the remote data source can process every operation in the original query, its processing speed might be much slower than the processing speed of the Db2 server, and it might turn out to be more beneficial to perform the operations at the Db2 server instead. If results are not satisfactory, verify your server statistics in the SYSCAT.SERVEROPTIONS catalog view.
- Why does a plan that is generated by the optimizer, and that is completely evaluated at a remote data source, perform much more poorly than the original query executed directly at the remote data source?

  Areas to examine include:

  – The remote SQL statement that is generated by the Db2 optimizer. Ensure that this statement is identical to the original query. Check for changes in predicate order. A good query optimizer should not be sensitive to the order of predicates in a query. The optimizer at the remote data source might generate a different plan, based on the order of input predicates. Consider either modifying the order of predicates in the input to the Db2 server, or contacting the service organization of the remote data source for assistance.

    You can also check for predicate replacements. A good query optimizer should not be sensitive to equivalent predicate replacements. The optimizer at the remote data source might generate a different plan, based on the input predicates. For example, some optimizers cannot generate transitive closure statements for predicates.

  – Additional functions. Does the remote SQL statement contain functions that are not present in the original query? Some of these functions might be used to convert data types; be sure to verify that they are necessary.

Data-access methods

When it compiles an SQL or XQuery statement, the query optimizer estimates the execution cost of different ways of satisfying the query.

Based on these estimates, the optimizer selects an optimal access plan. An *access plan* specifies the order of operations that are required to resolve an SQL or XQuery statement. When an application program is bound, a package is created. This *package* contains access plans for all of the static SQL and XQuery statements in that application program. Access plans for dynamic SQL and XQuery statements are created at run time.

There are three ways to access data in a table:

- By scanning the entire table sequentially
- By accessing an index on the table to locate specific rows
- By scan sharing

Rows might be filtered according to conditions that are defined in predicates, which are usually stated in a WHERE clause. The selected rows in accessed tables are joined to produce the result set, and this data might be further processed by grouping or sorting of the output.

Starting with Db2 9.7, *scan sharing*, which is the ability of a scan to use the buffer pool pages of another scan, is default behavior. Scan sharing increases workload concurrency and performance. With scan sharing, the system can support a larger number of concurrent applications, queries can perform better, and system throughput can increase, benefiting even queries that do not participate in scan sharing. Scan sharing is particularly effective in environments with applications that perform scans such as table scans or multidimensional clustering (MDC) block index scans of large tables. The compiler determines whether a scan is eligible to participate in scan sharing based on the type of scan, its purpose, the isolation level, how much work is done per record, and so on.

Data access through index scans:

An *index scan* occurs when the database manager accesses an index to narrow the set of qualifying rows (by scanning the rows in a specified range of the index) before accessing the base table; to order the output; or to retrieve the requested column data directly (*index-only access*).

When scanning the rows in a specified range of the index, the *index scan range* (the start and stop points of the scan) is determined by the values in the query against which index columns are being compared. In the case of index-only access, because all of the requested data is in the index, the indexed table does not need to be accessed.

If indexes are created with the ALLOW REVERSE SCANS option, scans can also be performed in a direction that is opposite to that with which they were defined.

The optimizer chooses a table scan if no appropriate index is created, or if an index scan would be more costly. An index scan might be more costly:

- When the table is small.
- When the index-clustering ratio is low.
- When the query requires most of the table rows.
- Where extra sorts are required when a partitioned index is used. A partitioned index cannot preserve the order in certain cases.
- Where extra sorts are required when an index with random ordering is used.

To determine whether the access plan uses a table scan or an index scan, use the Db2 explain facility.

Index scans to limit a range

To determine whether an index can be used for a particular query, the optimizer evaluates each column of the index, starting with the first column, to see if it can be used to satisfy equality and other predicates in the WHERE clause. A *predicate* is an element of a search condition in a WHERE clause that expresses or implies a comparison operation. Predicates can be used to limit the scope of an index scan in the following cases:

- Tests for IS NULL or IS NOT NULL
- Tests for strict and inclusive inequality
- Tests for equality against a constant, a host variable, an expression that evaluates to a constant, or a keyword

- Tests for equality against a basic subquery, which is a subquery that does not contain ANY, ALL, or SOME; this subquery must not have a correlated column reference to its immediate parent query block (that is, the select for which this subquery is a subselect).

Note: A range predicate or IS NOT NULL predicate on the random column of an index cannot be used as a start-stop predicate for that index. Only equality predicates on the random column of an index can be used in a start-stop predicate. However, the random index can still be chosen to satisfy the query by doing a full index scan. The random index can also be chosen to satisfy the query by using jumpscan with the random column treated as a gap.

The following examples show how an index could be used to limit the range of a scan.

- Consider an index with the following definition:

```
INDEX IX1:  NAME    ASC,
            DEPT    ASC,
            MGR     DESC,
            SALARY  DESC,
            YEARS   ASC
```

  The following predicates could be used to limit the range of a scan that uses index IX1:

```
where
  name = :hv1 and
  dept = :hv2
```

  or

```
where
  mgr = :hv1 and
  name = :hv2 and
  dept = :hv3
```

  The second WHERE clause demonstrates that the predicates do not have to be specified in the order in which the key columns appear in the index. Although the examples use host variables, other variables, such as parameter markers, expressions, or constants, could be used instead.

  In the following WHERE clause, only the predicates that reference NAME and DEPT would be used for limiting the range of the index scan:

```
where
  name = :hv1 and
  dept = :hv2 and
  salary = :hv4 and
  years = :hv5
```

  Because there is a key column (MGR) separating these columns from the last two index key columns, the ordering would be off. However, after the range is determined by the `name = :hv1` and `dept = :hv2` predicates, the other predicates can be evaluated against the remaining index key columns.

- Consider an index that was created using the ALLOW REVERSE SCANS option:

```
create index iname on tname (cname desc) allow reverse scans
```

  In this case, the index (INAME) is based on descending values in the CNAME column. Although the index is defined for scans running in descending order, a scan can be done in ascending order. Use of the index is controlled by the optimizer when creating and considering access plans.

Index scans to test inequality

Certain inequality predicates can limit the range of an index scan. There are two types of inequality predicates:

- Strict inequality predicates

  The strict inequality operators that are used for range-limiting predicates are greater than (>) and less than (<).

  Only one column with strict inequality predicates is considered for limiting the range of an index scan. In the following example, predicates on the NAME and DEPT columns can be used to limit the range, but the predicate on the MGR column cannot be used for that purpose.

  ```
  where
    name = :hv1 and
    dept > :hv2 and
    dept < :hv3 and
    mgr < :hv4
  ```

  However, in access plans that use jump scans, multiple columns with strict inequality predicates can be considered for limiting the range of an index scan. In this example (assuming the optimizer chooses an access plan with a jump scan), the strict inequality predicates on the DEPT, and MGR columns can be used to limit the range. While this example focuses on strict inequality predicates, note that the equality predicate on the NAME column is also used to limit the range.

- Inclusive inequality predicates

  The inclusive inequality operators that are used for range-limiting predicates are:

  – >= and <=
  – BETWEEN
  – LIKE

  Multiple columns with inclusive inequality predicates can be considered for limiting the range of an index scan or a jump scan. In the following example, all of the predicates can be used to limit the range.

  ```
  where
    name = :hv1 and
    dept >= :hv2 and
    dept <= :hv3 and
    mgr <= :hv4
  ```

  Suppose that `:hv2 = 404`, `:hv3 = 406`, and `:hv4 = 12345`. The database manager will scan the index for departments 404 and 405, but it will stop scanning department 406 when it reaches the first manager whose employee number (MGR column) is greater than 12345.

Index scans to order data

If a query requires sorted output, an index can be used to order the data if the ordering columns appear consecutively in the index, starting from the first index key column. Ordering or sorting can result from operations such as ORDER BY, DISTINCT, GROUP BY, an "= ANY" subquery, a "> ALL" subquery, a "< ALL" subquery, INTERSECT or EXCEPT, and UNION. Exceptions to this are as follows:

- If the index is partitioned, it can be used to order the data only if the index key columns are prefixed by the table-partitioning key columns, or if partition elimination eliminations all but one partition.

- Ordering columns can be different from the first index key columns when index key columns are tested for equality against “constant values” or any expression that evaluates to a constant.
- Indexes with random ordering cannot be used to order data.

Consider the following query:

```
where
  name = 'JONES' and
  dept = 'D93'
order by mgr
```

For this query, the index might be used to order the rows, because NAME and DEPT will always be the same values and will therefore be ordered. That is, the preceding WHERE and ORDER BY clauses are equivalent to:

```
where
  name = 'JONES' and
  dept = 'D93'
order by name, dept, mgr
```

A unique index can also be used to truncate a sort-order requirement. Consider the following index definition and ORDER BY clause:

```
UNIQUE INDEX IX0: PROJNO  ASC

select projno, projname, deptno
  from project
  order by projno, projname
```

Additional ordering on the PROJNAME column is not required, because the IX0 index ensures that PROJNO is unique: There is only one PROJNAME value for each PROJNO value.

Jump scans

Queries against tables with composite (multi-column) indexes present a particular challenge when designing indexes for tables. Ideally, a query's predicates are consistent with a table's composite index. This would mean that each predicate could be used as a start-stop key, which would, in turn, reduce the scope of the index needing to be searched. When a query contains predicates that are inconsistent with a composite index, this is known as an *index gap*. As such, index gaps are a characteristic of how a query measures up to a table's indexes.

Unconstrained index gap

Consider the following query against a table with the `IX1` composite index defined earlier in this topic:

```
where
  name = :hv1 and
  dept = :hv2 and
  mgr = :hv3 and
  years = IS NOT NULL
```

This query contains an index gap: on the `SALARY` key part of the composite index (this is assuming that the access plan contains an index scan on the composite index). The `SALARY` column cannot be included as a start-stop predicate. The `SALARY` column is an example of an *unconstrained index gap*.

Note: For some queries it can be difficult to assess whether or not there are index gaps. Use the EXPLAIN output to determine if index gaps are present.

Constrained index gap

Consider the following query against a table with the IX1 composite index defined earlier in this topic:

```
where
  name = :hv1 and
  dept = :hv2 and
  mgr = :hv3 and
  salary < :hv4 and
  years = :hv5
```

This query contains an index gap on the SALARY key part of the composite index (this is assuming that the access plan contains an index scan on the composite index). Since the SALARY column in the query is not an equality predicate, start-stop values cannot be generated for this column. The SALARY key part represents a *constrained index gap*.

To avoid poor performance in queries with index gaps, the optimizer can perform a *jump scan* operation. In a jump scan operation, the index manager identifies qualifying keys for small sections of a composite index where there are gaps, and fills these gaps with these qualifying keys. The end result is that the index manager skips over parts of the index that will not yield any results.

Jump scan restrictions

For queries being issued where you expect a jump scan, verify that the target table has an appropriate composite index and that the query predicates introduce an index gap. The Db2 optimizer will not create plans with jump scans if there are no index gaps.

Jump scans do not scan the following types of indexes:

- range-clustered table indexes
- extended indexes (for example, spatial indexes)
- XML indexes
- text indexes (for Text Search)

With jump scans, a gap column with an IN-List predicate might be treated as an unconstrained gap column. Also, in databases with Unicode Collation Algorithm (UCA) collation, jump scans might treat LIKE predicates with host variables or parameter markers as unconstrained gaps.

Note: When evaluating queries, there can be cases where the optimizer chooses an access plan that does not include a jump scan operation, even if index gaps are present. This would occur if the optimizer deems an alternative to using a jump scan to be more efficient.

Types of index access:

In some cases, the optimizer might find that all of the data that a query requires can be retrieved from an index on the table. In other cases, the optimizer might use more than one index to access tables. In the case of range-clustered tables, data can be accessed through a “virtual” index, which computes the location of data records.

Index-only access

In some cases, all of the required data can be retrieved from an index without accessing the table. This is known as *index-only access*. For example, consider the following index definition:

```
INDEX IX1:  NAME    ASC,
            DEPT    ASC,
            MGR     DESC,
            SALARY  DESC,
            YEARS   ASC
```

The following query can be satisfied by accessing only the index, without reading the base table:

```
select name, dept, mgr, salary
  from employee
  where name = 'SMITH'
```

Often, however, required columns do not appear in an index. To retrieve the data from these columns, table rows must be read. To enable the optimizer to choose index-only access, create a unique index with include columns. For example, consider the following index definition:

```
create unique index ix1 on employee
  (name asc)
  include (dept, mgr, salary, years)
```

This index enforces the uniqueness of the NAME column and also stores and maintains data for the DEPT, MGR, SALARY, and YEARS columns. In this way, the following query can be satisfied by accessing only the index:

```
select name, dept, mgr, salary
  from employee
  where name = 'SMITH'
```

Be sure to consider whether the additional storage space and maintenance costs of include columns are justified. If queries that exploit include columns are rarely executed, the costs might not be justified.

Multiple-index access

The optimizer can choose to scan multiple indexes on the same table to satisfy the predicates of a WHERE clause. For example, consider the following two index definitions:

```
INDEX IX2:  DEPT    ASC
INDEX IX3:  JOB     ASC,
            YEARS   ASC
```

The following predicates can be satisfied by using these two indexes:

```
where
  dept = :hv1 or
  (job = :hv2 and
  years >= :hv3)
```

Scanning index IX2 produces a list of record IDs (RIDs) that satisfy the `dept = :hv1` predicate. Scanning index IX3 produces a list of RIDs that satisfy the `job = :hv2 and years >= :hv3` predicate. These two lists of RIDs are combined, and duplicates are removed before the table is accessed. This is known as *index ORing*.

Index ORing can also be used for predicates that are specified by an IN clause, as in the following example:

```
where
  dept in (:hv1, :hv2, :hv3)
```

Although the purpose of index ORing is to eliminate duplicate RIDs, the objective of *index ANDing* is to find common RIDs. Index ANDing might occur when applications that create multiple indexes on corresponding columns in the same table run a query with multiple AND predicates against that table. Multiple index scans against each indexed column produce values that are hashed to create bitmaps. The second bitmap is used to probe the first bitmap to generate the qualifying rows for the final result set. For example, the following indexes:

```
INDEX IX4: SALARY   ASC
INDEX IX5: COMM     ASC
```

can be used to resolve the following predicates:

```
where
  salary between 20000 and 30000 and
  comm between 1000 and 3000
```

In this example, scanning index IX4 produces a bitmap that satisfies the `salary between 20000 and 30000` predicate. Scanning IX5 and probing the bitmap for IX4 produces a list of qualifying RIDs that satisfy both predicates. This is known as *dynamic bitmap ANDing*. It occurs only if the table has sufficient cardinality, its columns have sufficient values within the qualifying range, or there is sufficient duplication if equality predicates are used.

To realize the performance benefits of dynamic bitmaps when scanning multiple indexes, it might be necessary to change the value of the **sortheap** database configuration parameter and the **sheapthres** database manager configuration parameter. Additional sort heap space is required when dynamic bitmaps are used in access plans. When **sheapthres** is set to be relatively close to **sortheap** (that is, less than a factor of two or three times per concurrent query), dynamic bitmaps with multiple index access must work with much less memory than the optimizer anticipated. The solution is to increase the value of **sheapthres** relative to **sortheap**.

The optimizer does not combine index ANDing and index ORing when accessing a single table.

Index access in range-clustered tables

Unlike standard tables, a range-clustered table does not require a physical index (like a traditional B-tree index) that maps a key value to a row. Instead, it leverages the sequential nature of the column domain and uses a functional mapping to generate the location of a specific row in a table. In the simplest example of this type of mapping, the first key value in the range is the first row in the table, the second value in the range is the second row in the table, and so on.

The optimizer uses the range-clustered property of the table to generate access plans that are based on a perfectly clustered index whose only cost is computing the range clustering function. The clustering of rows within the table is guaranteed, because range-clustered tables retain their original key value order.

Index access in column-organized tables

The performance of a select, update, or delete operation that affects only one row in a column-organized table can be improved if the table has unique indexes, because the query optimizer can use an index scan instead of a full table scan.

A column-organized table can be accessed by using an index if that access returns no more than one row. Index access does not return more than one row if either of the following conditions is true:

- The index is defined as unique, and a predicate that equates to a constant value is applied to each key column in the index.
- The FETCH FIRST 1 ROW ONLY clause was specified in an SQL statement, and this option can be applied during the index scan.

Although you cannot explicitly create an index on a column-organized table, primary key or unique key constraints are enforced by unique indexes. You can create primary key or unique key constraints on column-organized tables to improve performance if the following conditions are true:

- The table data is truly unique.
- Workloads against the table have frequent select, update, or delete operations on unique columns that affect only one row.
- No primary key constraints or unique constraints exist on the columns that are being referenced by select, update, or delete operations that affect only one row.

Db2 Cancun Release 10.5.0.4 and later fix packs include index access support for SELECT statements that are run with isolation level CS (cursor stability). Moreover, the performance of an update or delete operation that references exactly one row in one table, where the row is identified by equality predicates that fully qualify a unique constraint, can be improved by using a unique index that supports a primary key or unique key constraint. The time that is needed to update or delete the single row can be improved if the statement meets the following criteria:

- No more than one row is produced by the index scan.
- No FETCH operation is necessary, meaning that access must be index-only access.
- The predicates that qualify the single row can all be applied as start or stop key values for the index search.
- The path from the index access to the update or delete operation must be unbroken; that is, the path cannot contain blocking operations such as TEMP, SORT, or HSJN.

For example, assume that unique index PK exists on column C1 and that unique index UK2 exists on columns (C1,C2). The absence of a CTQ operator immediately below DELETE is an indication that this optimization is occurring.

```
delete from t1
  where c1 = 99

             Rows
            RETURN
            (   1)
             Cost
              I/O
               |
               1
            DELETE
            (   2)
            16.3893
```

```
              2
         /----+-----\
        1            1000
     IXSCAN   CO-TABLE: BLUUSER
     (   3)           T1
     9.10425          Q1
        |
        1
       1000
INDEX: BLUUSER
       UK2
       Q2
```

Index access and cluster ratios:

When it chooses an access plan, the optimizer estimates the number of I/Os that are required to fetch pages from disk to the buffer pool. This estimate includes a prediction of buffer pool usage, because additional I/Os are not required to read rows from a page that is already in the buffer pool.

For index scans, information from the system catalog helps the optimizer to estimate the I/O cost of reading data pages into a buffer pool. It uses information from the following columns in the SYSCAT.INDEXES view:

- CLUSTERRATIO information indicates the degree to which the table data is clustered in relation to this index. The higher the number, the better the rows are ordered in index key sequence. If table rows are in close to index-key sequence, rows can be read from a data page while the page is in the buffer. If the value of this column is -1, the optimizer uses PAGE_FETCH_PAIRS and CLUSTERFACTOR information, if it is available.
- The PAGE_FETCH_PAIRS column contains pairs of numbers that model the number of I/Os required to read the data pages into buffer pools of various sizes, together with CLUSTERFACTOR information. Data is collected for these columns only if you invoke the **RUNSTATS** command against the index, specifying the DETAILED clause.

If index clustering statistics are not available, the optimizer uses default values, which assume poor clustering of the data with respect to the index. The degree to which the data is clustered can have a significant impact on performance, and you should try to keep one of the indexes that are defined on the table close to 100 percent clustered. In general, only one index can be one hundred percent clustered, except when the keys for an index represent a superset of the keys for the clustering index, or when there is an actual correlation between the key columns of the two indexes.

When you reorganize a table, you can specify an index that will be used to cluster the rows and keep them clustered during insert processing. Because update and insert operations can make a table less clustered in relation to the index, you might need to periodically reorganize the table. To reduce the number of reorganizations for a table that experiences frequent insert, update, or delete operations, specify the PCTFREE clause on the ALTER TABLE statement.

Scan sharing:

Scan sharing refers to the ability of one scan to exploit the work done by another scan. Examples of shared work include disk page reads, disk seeks, buffer pool content reuse, decompression, and so on.

Heavy scans, such as table scans or multidimensional clustering (MDC) block index scans of large tables, are sometimes eligible for sharing page reads with other scans. Such shared scans can start at an arbitrary point in the table, to take advantage of pages that are already in the buffer pool. When a sharing scan reaches the end of the table, it continues at the beginning and finishes when it reaches the point at which it started. This is called a *wrapping scan*. Figure 25 shows the difference between regular scans and wrapping scans for both tables and indexes.

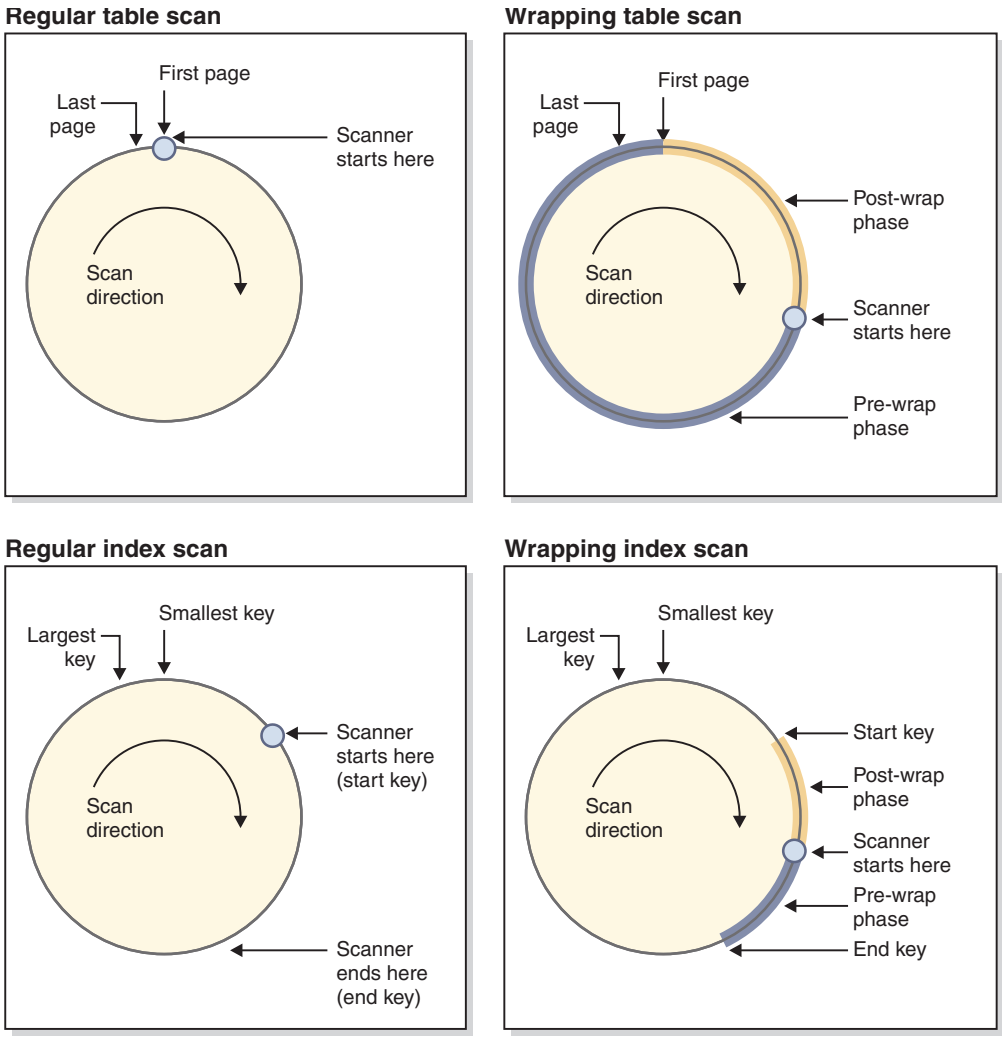


Figure 25. Conceptual view of regular and wrapping scans

The scan sharing feature is enabled by default, and eligibility for scan sharing and for wrapping are determined automatically by the SQL compiler. At run time, an eligible scan might or might not participate in sharing or wrapping, based on factors that were not known at compile time.

Shared scanners are managed in *share groups*. These groups keep their members together as long as possible, so that the benefits of sharing are maximized. If one scan is faster than another scan, the benefits of page sharing can be lost. In this situation, buffer pool pages that are accessed by the first scan might be cleared from the buffer pool before another scan in the share group can access them. The data server measures the distance between two scans in the same share group by the number of buffer pool pages that lies between them. The data server also

monitors the speed of the scans. If the distance between two scans in the same share group grows too large, they might not be able to share buffer pool pages. To reduce this effect, faster scans can be throttled to allow slower scans to access the data pages before they are cleared. Figure 26 shows two sharing sets, one for a table and one for a block index. A *sharing set* is a collection of share groups that are accessing the same object (for example, a table) through the same access mechanism (for example, a table scan or a block index scan). For table scans, the page read order increases by page ID; for block index scans, the page read order increases by key value.

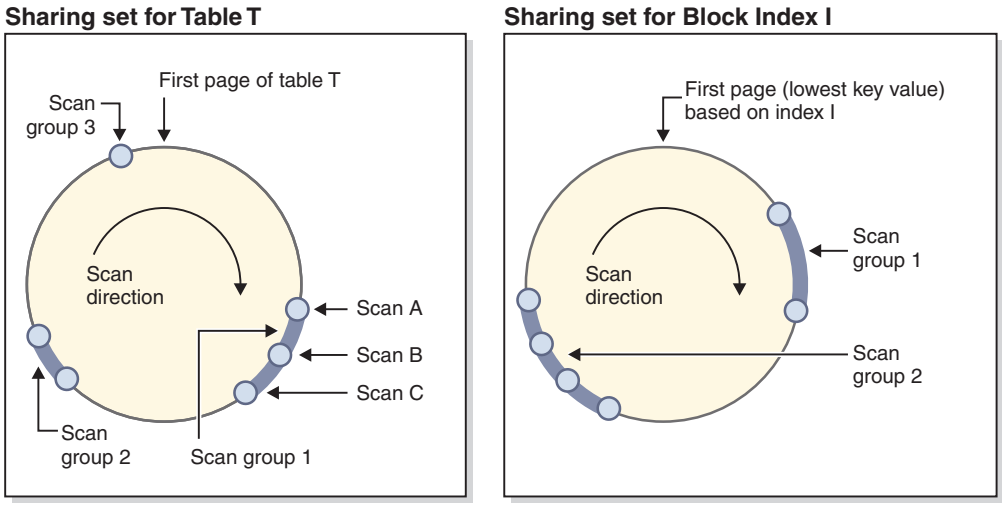


Figure 26. Sharing sets for table and block index scan sharing

The figure also shows how buffer pool content is reused within groups. Consider scan C, which is the leading scan of group 1. The following scans (A and B) are grouped with C, because they are close and can likely reuse the pages that C has brought into the buffer pool.

A high-priority scanner is never throttled by a lower priority one, and might move to another share group instead. A high priority scanner might be placed in a group where it can benefit from the work being done by the lower priority scanners in the group. It will stay in that group for as long as that benefit is available. By either throttling the fast scanner, or moving it to a faster share group (if the scanner encounters one), the data server adjusts the share groups to ensure that sharing remains optimized.

You can use the **db2pd** command to view information about scan sharing. For example, for an individual shared scan, the **db2pd** output will show data such as the scan speed and the amount of time that the scan was throttled. For a sharing group, the command output shows the number of scans in the group and the number of pages shared by the group.

The EXPLAIN_ARGUMENT table has new rows to contain scan-sharing information about table scans and index scans (you can use the **db2exfmt** command to format and view the contents of this table).

You can use optimizer profiles to override decisions that the compiler makes about scan sharing (see "Access types"). Such overrides are for use only when a special need arises; for example, the wrapping hint can be useful when a repeatable order of records in a result set is needed, but an ORDER BY clause (which might trigger

a sort) is to be avoided. Otherwise, it is recommended that you not use these optimization profiles unless requested to do so by Db2 Service.

Predicate processing for queries

A predicate is an element of a search condition that expresses or implies a comparison operation. Predicates can be grouped into four categories that are determined by how and when the predicate is used in the evaluation process.

The categories are provided in the following list, ordered in terms of performance starting with the most favorable:

- Range delimiting predicates are those used to bracket an index scan; they provide start or stop key values for the index search. These predicates are evaluated by the index manager.
- Index sargable predicates are not used to bracket a search, but are evaluated from the index if one is chosen, because the columns involved in the predicate are part of the index key. These predicates are also evaluated by the index manager.
- Data sargable predicates are predicates that cannot be evaluated by the index manager, but can be evaluated by Data Management Services (DMS). Typically, these predicates require the access of individual rows from a base table. If necessary, DMS will retrieve the columns needed to evaluate the predicate, as well as any others to satisfy the columns in the SELECT list that could not be obtained from the index.
- Residual predicates are those that require I/O beyond the simple accessing of a base table. Examples of residual predicates include those using quantified subqueries (subqueries with ANY, ALL, SOME, or IN), or reading large object (LOB) data that is stored separately from the table. These predicates are evaluated by Relational Data Services (RDS) and are the most expensive of the four categories of predicates.

The following table provides examples of various predicates and identifies their type based on the context in which they are used.

Note: In these examples, assume that a multi-column ascending index exists on (c1, c2, c3) and is used in evaluating the predicates where appropriate. If any column in the index is in descending order, the start and stop keys might be switched for range delimiting predicates.

Table 57. Predicate processing for different queries

Predicates	Column c1	Column c2	Column c3	Comments	During a jump scan or skip scan
c1 = 1 and c2 = 2 and c3 = 3	Range delimiting (start-stop)	Range delimiting (start-stop)	Range delimiting (start-stop)	The equality predicates on all the columns of the index can be applied as start-stop keys.	Not applicable
c1 = 1 and c2 = 2 and c3 ≥ 3	Range delimiting (start-stop)	Range delimiting (start-stop)	Range delimiting (start)	Columns c1 and c2 are bound by equality predicates, and the predicate on c3 is only applied as a start key.	Not applicable

Table 57. Predicate processing for different queries (continued)

Predicates	Column c1	Column c2	Column c3	Comments	During a jump scan or skip scan
c1 ≥1 and c2 = 2	Range delimiting (start)	Range delimiting (start-stop)	Not applicable	The leading column c1 has a ≥ predicate and can be used as a start key. The following column c2 has an equality predicate, and therefore can also be applied as a start-stop key.	Not applicable
c1 = 1 and c3 = 3	Range delimiting (start-stop)	Not applicable	Index sargable	The predicate on c3 cannot be used as a start-stop key, because there is no predicate on c2. It can, however, be applied as an index sargable predicate.	If skip scan is used, then there is a predicate on c2. c3 = 3 is applied as both a range delimiting predicate and an index sargable predicate. c3 = 3 is applied as a start-stop key when it can eliminate keys less than (1, x, 3) or greater than (1, y, 3) where x is the minimum value for c2 and y is the maximum value for c2. Otherwise, it is reapplied as an index sargable to eliminate keys such as (1, 2, 4).
c1 = 1 and c2 > 2 and c3 = 3	Range delimiting (start-stop)	Range delimiting (start)	Index sargable	The predicate on c3 cannot be applied as a start-stop predicate because the previous column has a > predicate. Had it been a ≥ instead, you would be able to use it as a start-stop key.	If skip scan is used, the predicate on c3 can be applied as a start-stop predicate despite the previous column with a > predicate. As in the example predicates case of "c1 = 1 and c3 = 3", c3 = 3 can be applied as a start-stop key and also be applied as an index sargable predicate.

Table 57. Predicate processing for different queries (continued)

Predicates	Column c1	Column c2	Column c3	Comments	During a jump scan or skip scan
c1 = 1 and c2 ≤ 2 and c4 = 4	Range delimiting (start-stop)	Range delimiting (stop)	Data sargable	Here the predicate on c2 is a ≤ predicate. It can be used as a stop key. The predicate on c4 cannot be applied on the index and is applied as a data sargable predicate during the FETCH.	Not applicable
c2 = 2 and UDF_with_external_action(c4)	Not applicable	Index sargable	Residual	The leading column c1 does not have a predicate, and therefore the predicate on c2 can be applied as an index sargable predicate where the whole index is scanned. The predicate involving the user-defined function with external action is applied as a residual predicate.	As in the example predicates case of “c1 =1 and c3 = 3”, c2 = 2 can be applied as a start-stop key and also be applied as an index sargable predicate.
c1 = 1 or c2 = 2	Index sargable	Index sargable	Not applicable	The presence of an OR does not allow us this multi-column index to be used as start-stop keys. This might have been possible had there been two indexes, one with a leading column on c1, and the other with a leading column on c2, and the Db2 optimizer chose an "index-ORing" plan. However, in this case the two predicates are treated as index sargable predicates.	Not applicable
c1 < 5 and (c2 = 2 or c3 = 3)	Range delimiting (stop)	Index sargable	Index sargable	Here the leading column c1 is exploited to stop the index scan from using the predicate with a stop key. The OR predicate on c2 and c3 are applied as index sargable predicates.	Not applicable

The Db2 optimizer employs the query rewrite mechanism to transform many complex user-written predicates into better performing queries, as shown in the following table:

Table 58. Query rewrite predicates

Original predicate or query	Optimized predicates	Comments
c1 between 5 and 10	c1 ≥ 5 and c1 ≤ 10	The BETWEEN predicates are rewritten into the equivalent range delimiting predicates so that they can be used internally as though the user specified the range delimiting predicates.
c1 not between 5 and 10	c1 < 5 or c1 > 10	The presence of the OR predicate does not allow the use of a start-stop key unless the Db2 optimizer chooses an index-ORing plan.
SELECT * **FROM** t1 **WHERE EXISTS** (**SELECT** c1 **FROM** t2 **WHERE** t1.c1 = t2.c1)	**SELECT** t1.* **FROM** t1 EOJOIN t2 **WHERE** t1.c1= t2.c1	The subquery might be transformed into a join.
SELECT * **FROM** t1 **WHERE** t1.c1 **IN** (**SELECT** c1 **FROM** t2)	**SELECT** t1* **FROM** t1 EOJOIN t2 **WHERE** t1.c1= t2.c1	This is similar to the transformation for the EXISTS predicate example in the previous row.
c1 like 'abc%'	c1 ≥ 'abc X X X ' and c1 ≤ 'abc Y Y Y'	If you have c1 as the leading column of an index, Db2 generates these predicates so that they can be applied as range-delimiting start-stop predicates. Here the characters X and Y are symbolic of the lowest and highest collating character.
c1 like 'abc%def'	c1 ≥ 'abc X X X ' and c1 ≤ 'abc Y Y Y' and c1 like 'abc%def'	This is similar to the previous case, except that you also have to apply the original predicate as an index sargable predicate. This ensures that the characters def match correctly.

Restrictions

The following restrictions are applicable to scenarios that involve indexes with random ordering:

- The key part of an index with random ordering cannot use range delimiting predicates to satisfy LIKE, <, <= ,> ,>=, or IS NOT NULL predicates.
- Predicates that compare BIGINT or DECIMAL random ordered index column types to REAL or DOUBLE values cannot be applied as start-stop keys on the random ordered index.
- Predicates that compare REAL or DOUBLE random ordered index column types to DECFLOAT values cannot be applied as start-stop keys on the random ordered index.

Query performance for common SQL statements

A number of performance improvements have been made to improve the speed of many queries. These improvements are automatic. There are no configuration settings or changes to the SQL statements required.

Partial early distinct (PED)

An efficient hashing function will now be used to partially remove duplicates early in the processing of the query. This may not remove all duplicates, but will reduce the amount of data that must be processed later in the query evaluation. Removing some of the initial duplicate rows will speed up the query, and reduce the chance that it will run out of sort heap memory, thereby eliminating the need to use relatively slow disk space for temporary storage in these cases. This improvement is termed partial early distinct (PED).

To determine if this improvement is being used for a particular query, activate the Explain facility and run the query. A new value in the EXPLAIN_ARGUMENT table indicates when this new functionality has been applied to a query:

- ARGUMENT_TYPE column = UNIQUE
- ARGUMENT_VALUE column can now also have the value: `HASHED PARTIAL` which indicates that the new feature has been used

The **db2exfmt** tool will also show `HASHED PARTIAL` in its output, as shown in the following example:

```
6) UNIQUE: (Unique)
      Cumulative Total Cost:   132.519
      Cumulative CPU Cost:   1.98997e+06
      ...
      ...
      Arguments:
      ---------
      JN INPUT: (Join input leg)
            INNER
      UNIQKEY : (Unique Key columns)
            1: Q1.C22
      UNIQKEY : (Unique Key columns)
            2: Q1.C21
      pUNIQUE  : (Uniqueness required flag)
            HASHED PARTIAL
```

Partial early aggregation (PEA)

Similar to partial early distinct (PED), partial early aggregation (PEA) is an attempt to do a partial aggregation of data early in the processing of the query. While it is unlikely that all aggregation can take place at this point, it will at least reduce the amount of data that must be processed later in the query evaluation.

To determine if partial early aggregation is being used for a particular query activate the Explain facility and run the query. A new value in the EXPLAIN_ARGUMENT table indicates when this new functionality has been applied to a query:

- ARGUMENT_TYPE column = AGGMODE
- ARGUMENT_VALUE column can now also have the value: `HASHED PARTIAL` which indicates that this new feature has been used

The **db2exfmt** tool will also show `HASHED PARTIAL` in its output for `GRPBY` sections, along with a `pGRPBY` in the tree view, if this new functionality has been applied within that part of the query.

Hash join now selected by the query optimizer for a wider range of SQL queries

The query optimizer chooses between three basic join strategies when determining how to run an SQL query that includes a join. In many cases a hash join is the most efficient method, and with this release it can be used in more situations.

Data type mismatches
A hash join will now be considered even if the two columns in the join are not the same data type. This is the case in all but the most extreme situations.

Expressions used in join predicate
Join predicates that contain an expression no longer restrict the join method to a nested loop join. In this release a hash join is considered in cases where the WHERE clause contains an expression, like: `WHERE T1.C1 = UPPER(T1.C3)`

In these cases the hash join is considered automatically. There is no need to change any existing SQL queries to take advantage of this improved functionality. Note that hash joins make use of sort heap memory.

Improved cost estimates of network communication traffic generated by a query

The query optimizer relies on a range of information to choose an access plan that is efficient as possible. The estimated communication costs of queries has now been improved, enabling the optimizer to more accurately consider and compare all of the CPU, IO, and communication costs. In many cases this will result in faster query performance.

The estimated per-node communication costs of a query, as returned by the explain elements **COMM_COST** and **FIRST_COMM_COST**, have been improved. They are now more consistent with the existing CPU and IO costs per-node calculations. This enables the query optimizer to effectively balance all three of these cost estimations when evaluating different access plans. It also helps increase parallelism when possible by enabling the network traffic to be spread more evenly across multiple network adapters. In particular:

- If there is more than one network adapter involved, the cumulative communication cost for the adapter with the highest value is returned. In previous releases the total number of frames transmitted throughout the entire network was returned.
- The values only include the costs of network traffic between physical machines. They do not include the virtual communication costs between node partitions on the same physical machine in a partitioned database environment.

Joins

A *join* is the process of combining data from two or more tables based on some common domain of information. Rows from one table are paired with rows from another table when information in the corresponding rows match on the basis of the joining criterion (the *join predicate*).

For example, consider the following two tables:

TABLE1	
PROJ	PROJ_ID
A	1
B	2
C	3
D	4

TABLE2	
PROJ_ID	NAME
1	Sam
3	Joe
4	Mary
1	Sue
2	Mike

To join TABLE1 and TABLE2, such that the PROJ_ID columns have the same values, use the following SQL statement:

```
select proj, x.proj_id, name
  from table1 x, table2 y
  where x.proj_id = y.proj_id
```

In this case, the appropriate join predicate is: `where x.proj_id = y.proj_id`.

The query yields the following result set:

PROJ	PROJ_ID	NAME
A	1	Sam
A	1	Sue
B	2	Mike
C	3	Joe
D	4	Mary

Depending on the nature of any join predicates, as well as any costs determined on the basis of table and index statistics, the optimizer chooses one of the following join methods:

- Nested-loop join
- Merge join
- Hash join

When two tables are joined, one table is selected as the outer table and the other table is regarded as the inner table of the join. The outer table is accessed first and is scanned only once. Whether the inner table is scanned multiple times depends on the type of join and the indexes that are available. Even if a query joins more than two tables, the optimizer joins only two tables at a time. If necessary, temporary tables are created to hold intermediate results.

You can provide explicit join operators, such as INNER or LEFT OUTER JOIN, to determine how tables are used in the join. Before you alter a query in this way, however, you should allow the optimizer to determine how to join the tables, and then analyze query performance to decide whether to add join operators.

Join methods:

The optimizer can choose one of three basic join strategies when queries require tables to be joined: nested-loop join, merge join, or hash join.

Nested-loop join

A nested-loop join is performed in one of the following two ways:

- Scanning the inner table for each accessed row of the outer table

For example, column A in table T1 and column A in table T2 have the following values:

Outer table T1: Column A	Inner table T2: Column A
2	3
3	2
3	2
	3
	1

To complete a nested-loop join between tables T1 and T2, the database manager performs the following steps:

1. Read the first row in T1. The value for A is 2.
2. Scan T2 until a match (2) is found, and then join the two rows.
3. Repeat Step 2 until the end of the table is reached.
4. Go back to T1 and read the next row (3).
5. Scan T2 (starting at the first row) until a match (3) is found, and then join the two rows.
6. Repeat Step 5 until the end of the table is reached.
7. Go back to T1 and read the next row (3).
8. Scan T2 as before, joining all rows that match (3).

- Performing an index lookup on the inner table for each accessed row of the outer table

  This method can be used if there is a predicate of the form:

  ```
  expr(outer_table.column) relop inner_table.column
  ```

  where `relop` is a relative operator (for example =, >, >=, <, or <=) and `expr` is a valid expression on the outer table. For example:

  ```
  outer.c1 + outer.c2 <= inner.c1
  outer.c4 < inner.c3
  ```

  This method might significantly reduce the number of rows that are accessed in the inner table for each access of the outer table; the degree of benefit depends on a number of factors, including the selectivity of the join predicate.

When it evaluates a nested-loop join, the optimizer also decides whether to sort the outer table before performing the join. If it sorts the outer table, based on the join columns, the number of read operations against the inner table to access pages on disk might be reduced, because they are more likely to be in the buffer pool already. If the join uses a highly clustered index to access the inner table, and the outer table has been sorted, the number of accessed index pages might be minimized.

If the optimizer expects that the join will make a later sort more expensive, it might also choose to perform the sort before the join. A later sort might be required to support a GROUP BY, DISTINCT, ORDER BY, or merge join operation.

Merge join

A merge join, sometimes known as a *merge scan join* or a *sort merge join*, requires a predicate of the form `table1.column = table2.column`. This is called an *equality join predicate*. A merge join requires ordered input on the joining columns, either through index access or by sorting. A merge join cannot be used if the join column is a LONG field column or a large object (LOB) column.

In a merge join, the joined tables are scanned at the same time. The outer table of the merge join is scanned only once. The inner table is also scanned once, unless repeated values occur in the outer table. If repeated values occur, a group of rows in the inner table might be scanned again.

For example, column A in table T1 and column A in table T2 have the following values:

Outer table T1: Column A	Inner table T2: Column A
2	1
3	2
3	2
	3
	3

To complete a merge join between tables T1 and T2, the database manager performs the following steps:

1. Read the first row in T1. The value for A is 2.
2. Scan T2 until a match (2) is found, and then join the two rows.
3. Keep scanning T2 while the columns match, joining rows.
4. When the 3 in T2 is read, go back to T1 and read the next row.
5. The next value in T1 is 3, which matches T2, so join the rows.
6. Keep scanning T2 while the columns match, joining rows.
7. When the end of T2 is reached, go back to T1 to get the next row. Note that the next value in T1 is the same as the previous value from T1, so T2 is scanned again, starting at the first 3 in T2. The database manager remembers this position.

Hash join

A hash join requires one or more predicates of the form `table1.columnX = table2.columnY` None of the columns can be either a LONG field column or a LOB column.

A hash join is performed as follows: First, the designated inner table is scanned and rows are copied into memory buffers that are drawn from the sort heap specified by the **sortheap** database configuration parameter. The memory buffers are divided into sections, based on a hash value that is computed on the columns of the join predicates. If the size of the inner table exceeds the available sort heap space, buffers from selected sections are written to temporary tables.

When the inner table has been processed, the second (or outer) table is scanned and its rows are matched with rows from the inner table by first comparing the hash value that was computed for the columns of the join predicates. If the hash value for the outer row column matches the hash value for the inner row column, the actual join predicate column values are compared.

Outer table rows that correspond to portions of the table that are not written to a temporary table are matched immediately with inner table rows in memory. If the corresponding portion of the inner table was written to a temporary table, the outer row is also written to a temporary table. Finally, matching pairs of table portions from temporary tables are read, the hash values of their rows are matched, and the join predicates are checked.

Hash join processing can also use filters to improve performance. If the optimizer chooses to use filters, and sufficient memory is available, filters can often reduce the number of outer table rows which need to be processed by the join.

For the full performance benefit of hash joins, you might need to change the value of the **sortheap** database configuration parameter and the **sheapthres** database manager configuration parameter.

Hash join performance is best if you can avoid hash loops and overflow to disk. To tune hash join performance, estimate the maximum amount of memory that is available for **sheapthres**, and then tune the **sortheap** parameter. Increase its setting until you avoid as many hash loops and disk overflows as possible, but do not reach the limit that is specified by the **sheapthres** parameter.

Increasing the **sortheap** value should also improve the performance of queries that have multiple sorts.

Strategies for selecting optimal joins:

The optimizer uses various methods to select an optimal join strategy for a query. Among these methods, which are determined by the optimization class of the query, are several search strategies, star-schema joins, early out joins, and composite tables.

The join-enumeration algorithm is an important determinant of the number of plan combinations that the optimizer explores.

- Greedy join enumeration
  - Is efficient with respect to space and time requirements
  - Uses single direction enumeration; that is, once a join method is selected for two tables, it is not changed during further optimization
  - Might miss the best access plan when joining many tables. If your query joins only two or three tables, the access plan that is chosen by greedy join enumeration is the same as the access plan that is chosen by dynamic programming join enumeration. This is particularly true if the query has many join predicates on the same column that are either explicitly specified, or implicitly generated through predicate transitive closure.
- Dynamic programming join enumeration
  - Is not efficient with respect to space and time requirements, which increase exponentially as the number of joined tables increases
  - Is efficient and exhaustive when searching for the best access plan
  - Is similar to the strategy that is used by Db2 for z/OS

Star-schema joins

The tables that are referenced in a query are almost always related by join predicates. If two tables are joined without a join predicate, the Cartesian product of the two tables is formed. In a Cartesian product, every qualifying row of the first table is joined with every qualifying row of the second table. This creates a result table that is usually very large, because its size is the cross product of the size of the two source tables. Because such a plan is unlikely to perform well, the optimizer avoids even determining the cost of this type of access plan.

The only exceptions occur when the optimization class is set to 9, or in the special case of star schemas. A *star schema* contains a central table called the *fact table*, and

other tables called *dimension tables*. The dimension tables have only a single join that attaches them to the fact table, regardless of the query. Each dimension table contains additional values that expand information about a particular column in the fact table. A typical query consists of multiple local predicates that reference values in the dimension tables and contains join predicates connecting the dimension tables to the fact table. For these queries, it might be beneficial to compute the Cartesian product of multiple small dimension tables before accessing the large fact table. This technique is useful when multiple join predicates match a multicolumn index.

The Db2 data server can recognize queries against databases that were designed with star schemas having at least two dimension tables, and can increase the search space to include possible plans that compute the Cartesian product of dimension tables. A zigzag join is considered before the Cartesian product is even materialized. It probes the fact table using a multicolumn index, so that the fact table is filtered along two or more dimension tables simultaneously. When a zigzag join is used, it returns the next combination of values that is available from the fact table index. This next combination of values, known as feedback, is used to skip over probe values provided by the Cartesian product of dimension tables that will not find a match in the fact table. Filtering the fact table on two or more dimension tables simultaneously, and skipping probes that are known to be unproductive, together makes the zigzag join an efficient method for querying large fact tables.

This star schema join strategy assumes that primary key indexes are used in the join. Another scenario involves foreign key indexes. If the foreign key columns in the fact table are single-column indexes, and there is relatively high selectivity across all dimension tables, the following star-schema join technique can be used:

1. Process each dimension table by:
   - Performing a semi-join between the dimension table and the foreign key index on the fact table
   - Hashing the record ID (RID) values to dynamically create a bitmap
2. For each bitmap, use AND predicates against the previous bitmap.
3. Determine the surviving RIDs after processing the last bitmap.
4. Optionally sort these RIDs.
5. Fetch a base table row.
6. Rejoin the fact table with each of its dimension tables, accessing the columns in dimension tables that are needed for the SELECT clause.
7. Reapply the residual predicates.

This technique does not require multicolumn indexes. Explicit referential integrity constraints between the fact table and the dimension tables are not required, but are recommended.

The dynamic bitmaps that are created and used by star-schema join techniques require sort heap memory, the size of which is specified by the **sortheap** database configuration parameter.

Early out joins

The optimizer might select an early out join when it detects that each row from one of the tables only needs to be joined with at most one row from the other table.

An early out join is possible when there is a join predicate on the key column or columns of one of the tables. For example, consider the following query that returns the names of employees and their immediate managers.

```
select employee.name as employee_name,
    manager.name as manager_name
  from employee as employee, employee as manager
  where employee.manager_id = manager.id
```

Assuming that the ID column is a key in the EMPLOYEE table and that every employee has at most one manager, this join avoids having to search for a subsequent matching row in the MANAGER table.

An early out join is also possible when there is a DISTINCT clause in the query. For example, consider the following query that returns the names of car makers with models that sell for more than $30000.

```
select distinct make.name
  from make, model
  where
    make.make_id = model.make_id and
    model.price > 30000
```

For each car maker, you only need to determine whether any one of its manufactured models sells for more than $30000. Joining a car maker with all of its manufactured models selling for more than $30000 is unnecessary, because it does not contribute towards the accuracy of the query result.

An early out join is also possible when the join feeds a GROUP BY clause with a MIN or MAX aggregate function. For example, consider the following query that returns stock symbols with the most recent date before the year 2000, for which a particular stock's closing price is at least 10% higher than its opening price:

```
select dailystockdata.symbol, max(dailystockdata.date) as date
  from sp500, dailystockdata
  where
    sp500.symbol = dailystockdata.symbol and
    dailystockdata.date < '01/01/2000' and
    dailystockdata.close / dailystockdata.open >= 1.1
  group by dailystockdata.symbol
```

The *qualifying set* is the set of rows from the DAILYSTOCKDATA table that satisfies the date and price requirements and joins with a particular stock symbol from the SP500 table. If the qualifying set from the DAILYSTOCKDATA table (for each stock symbol row from the SP500 table) is ordered as descending on DATE, it is only necessary to return the first row from the qualifying set for each symbol, because that first row represents the most recent date for a particular symbol. The other rows in the qualifying set are not required.

Composite tables

When the result of joining a pair of tables is a new table (known as a *composite table*), this table usually becomes the outer table of a join with another inner table. This is known as a *composite outer join*. In some cases, particularly when using the greedy join enumeration technique, it is useful to make the result of joining two tables the inner table of a later join. When the inner table of a join consists of the result of joining two or more tables, this plan is known as a *composite inner join*. For example, consider the following query:

```
select count(*)
  from t1, t2, t3, t4
  where
    t1.a = t2.a and
    t3.a = t4.a and
    t2.z = t3.z
```

It might be beneficial to join table T1 and T2 (T1xT2), then join T3 and T4 (T3xT4), and finally, to select the first join result as the outer table and the second join result as the inner table. In the final plan ((T1xT2) x (T3xT4)), the join result (T3xT4) is known as a *composite inner join*. Depending on the query optimization class, the optimizer places different constraints on the maximum number of tables that can be the inner table of a join. Composite inner joins are allowed with optimization classes 5, 7, and 9.

Replicated materialized query tables in partitioned database environments:

Replicated materialized query tables (MQTs) improve the performance of frequently executed joins in a partitioned database environment by allowing the database to manage precomputed values of the table data.

Note that a replicated MQT in this context pertains to intra-database replication. Inter-database replication is concerned with subscriptions, control tables, and data that is located in different databases and on different operating systems.

In the following example:

- The SALES table is in a multi-partition table space named REGIONTABLESPACE, and is split on the REGION column.
- The EMPLOYEE and DEPARTMENT tables are in a single-partition database partition group.

Create a replicated MQT based on information in the EMPLOYEE table.

```
create table r_employee as (
  select empno, firstnme, midinit, lastname, workdept
    from employee
  )
  data initially deferred refresh immediate
  in regiontablespace
  replicated
```

Update the content of the replicated MQT:

```
refresh table r_employee
```

After using the REFRESH statement, you should invoke the runstats utility against the replicated table, as you would against any other table.

The following query calculates sales by employee, the total for the department, and the grand total:

```
select d.mgrno, e.empno, sum(s.sales)
  from department as d, employee as e, sales as s
  where
    s.sales_person = e.lastname and
    e.workdept = d.deptno
  group by rollup(d.mgrno, e.empno)
  order by d.mgrno, e.empno
```

Instead of using the EMPLOYEE table, which resides on only one database partition, the database manager uses R_EMPLOYEE, the MQT that is replicated on

each of the database partitions on which the SALES table is stored. The performance enhancement occurs because the employee information does not have to be moved across the network to each database partition when performing the join.

Replicated materialized query tables in collocated joins

Replicated MQTs can also assist in the collocation of joins. For example, if a star schema contains a large fact table that is spread across twenty database partitions, the joins between the fact table and the dimension tables are most efficient if these tables are collocated. If all of the tables are in the same database partition group, at most one dimension table is partitioned correctly for a collocated join. The other dimension tables cannot be used in a collocated join, because the join columns in the fact table do not correspond to the distribution key for the fact table.

Consider a table named FACT (C1, C2, C3, ...), split on C1; a table named DIM1 (C1, dim1a, dim1b, ...), split on C1; a table named DIM2 (C2, dim2a, dim2b, ...), split on C2; and so on. In this case, the join between FACT and DIM1 is perfect, because the predicate `dim1.c1 = fact.c1` is collocated. Both of these tables are split on column C1.

However, the join involving DIM2 and the predicate `dim2.c2 = fact.c2` cannot be collocated, because FACT is split on column C1, not on column C2. In this case, you could replicate DIM2 in the database partition group of the fact table so that the join occurs locally on each database partition.

When you create a replicated MQT, the source table can be a single-partition table or a multi-partition table in a database partition group. In most cases, the replicated table is small and can be placed in a single-partition database partition group. You can limit the data that is to be replicated by specifying only a subset of the columns from the table, or by restricting the number of qualifying rows through predicates.

A replicated MQT can also be created in a multi-partition database partition group, so that copies of the source table are created on all of the database partitions. Joins between a large fact table and the dimension tables are more likely to occur locally in this environment, than if you broadcast the source table to all database partitions.

Indexes on replicated tables are not created automatically. You can create indexes that are different from those on the source table. However, to prevent constraints violations that are not present in the source table, you cannot create unique indexes or define constraints on replicated tables, even if the same constraints occur on the source table.

Replicated tables can be referenced directly in a query, but you cannot use the DBPARTITIONNUM scalar function with a replicated table to see the table data on a particular database partition.

Use the Db2 explain facility to determine whether a replicated MQT was used by the access plan for a query. Whether or not the access plan that is chosen by the optimizer uses a replicated MQT depends on the data that is to be joined. A replicated MQT might not be used if the optimizer determines that it would be cheaper to broadcast the original source table to the other database partitions in the database partition group.

Join strategies for partitioned databases:

Join strategies for a partitioned database environment can be different than strategies for a nonpartitioned database environment. Additional techniques can be applied to standard join methods to improve performance.

Table collocation should be considered for tables that are frequently joined. In a partitioned database environment, *table collocation* refers to a state that occurs when two tables that have the same number of compatible partitioning keys are stored in the same database partition group. When this happens, join processing can be performed at the database partition where the data is stored, and only the result set needs to be moved to the coordinator database partition.

Table queues

Descriptions of join techniques in a partitioned database environment use the following terminology:

- *Table queue* (sometimes referred to as TQ) is a mechanism for transferring rows between database partitions, or between processors in a single-partition database.
- *Directed table queue* (sometimes referred to as DTQ) is a table queue in which rows are hashed to one of the receiving database partitions.
- *Broadcast table queue* (sometimes referred to as BTQ) is a table queue in which rows are sent to all of the receiving database partitions, but are not hashed.

A table queue is used to pass table data:

- From one database partition to another when using interpartition parallelism
- Within a database partition when using intrapartition parallelism
- Within a database partition when using a single-partition database

Each table queue passes the data in a single direction. The compiler decides where table queues are required, and includes them in the plan. When the plan is executed, connections between the database partitions initiate the table queues. The table queues close as processes end.

There are several types of table queues:

- Asynchronous table queues

  These table queues are known as asynchronous, because they read rows in advance of any fetch requests from an application. When a FETCH statement is issued, the row is retrieved from the table queue.

  Asynchronous table queues are used when you specify the FOR FETCH ONLY clause on the SELECT statement. If you are only fetching rows, the asynchronous table queue is faster.
- Synchronous table queues

  These table queues are known as synchronous, because they read one row for each FETCH statement that is issued by an application. At each database partition, the cursor is positioned on the next row to be read from that database partition.

  Synchronous table queues are used when you do not specify the FOR FETCH ONLY clause on the SELECT statement. In a partitioned database environment, if you are updating rows, the database manager will use synchronous table queues.
- Merging table queues

These table queues preserve order.

- Non-merging table queues

  These table queues, also known as *regular table queues*, do not preserve order.
- Listener table queues (sometimes referred to as LTQ)

  These table queues are used with correlated subqueries. Correlation values are passed down to the subquery, and the results are passed back up to the parent query block using this type of table queue.

Join methods for partitioned databases:

Several join methods are available for partitioned database environments, including: collocated joins, broadcast outer-table joins, directed outer-table joins, directed inner-table and outer-table joins, broadcast inner-table joins, and directed inner-table joins.

In the following diagrams, q1, q2, and q3 refer to table queues. The referenced tables are divided across two database partitions, and the arrows indicate the direction in which the table queues are sent. The coordinator database partition is database partition 0.

If the join method chosen by the compiler is hash join, the filters created at each remote database partition may be used to eliminate tuples before they are sent to the database partition where the hash join is processed, thus improving performance.

Collocated joins

A collocated join occurs locally on the database partition on which the data resides. The database partition sends the data to the other database partitions after the join is complete. For the optimizer to consider a collocated join, the joined tables must be collocated, and all pairs of the corresponding distribution keys must participate in the equality join predicates. Figure 27 on page 274 provides an example.

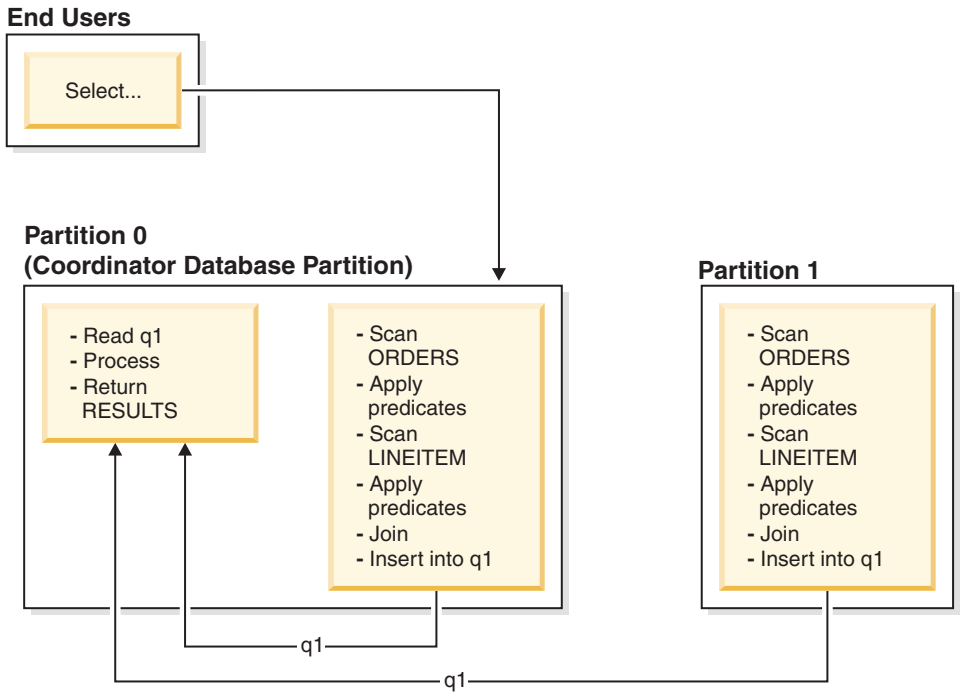


Figure 27. Collocated Join Example

The LINEITEM and ORDERS tables are both partitioned on the ORDERKEY column. The join is performed locally at each database partition. In this example, the join predicate is assumed to be: `orders.orderkey = lineitem.orderkey`.

Replicated materialized query tables (MQTs) enhance the likelihood of collocated joins.

Broadcast outer-table joins

Broadcast outer-table joins represent a parallel join strategy that can be used if there are no equality join predicates between the joined tables. It can also be used in other situations in which it proves to be the most cost-effective join method. For example, a broadcast outer-table join might occur when there is one very large table and one very small table, neither of which is split on the join predicate columns. Instead of splitting both tables, it might be cheaper to broadcast the smaller table to the larger table. Figure 28 on page 275 provides an example.

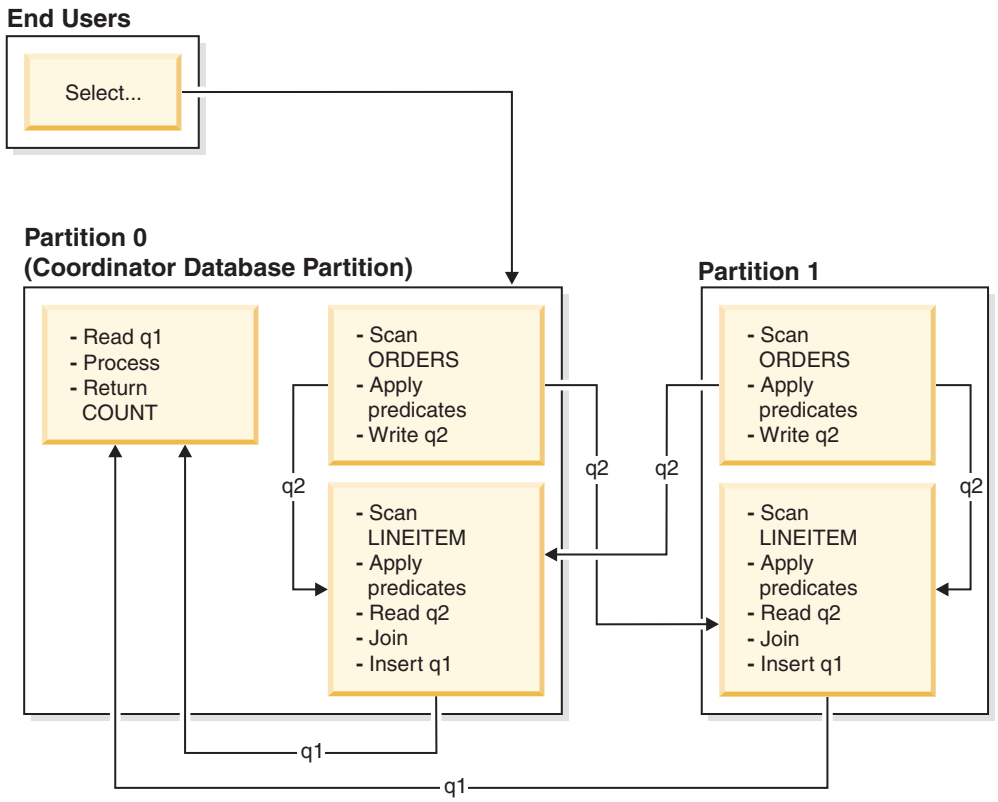
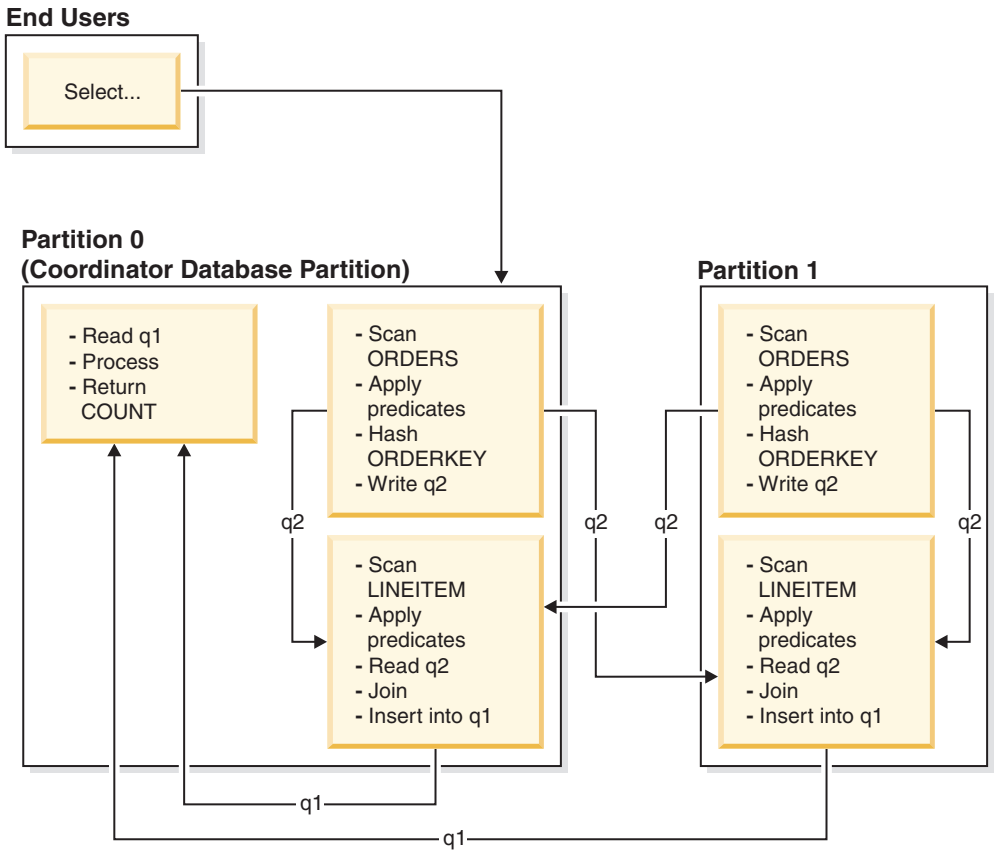


Figure 28. Broadcast Outer-Table Join Example

The ORDERS table is sent to all database partitions that have the LINEITEM table. Table queue q2 is broadcast to all database partitions of the inner table.

Directed outer-table joins

In the directed outer-table join strategy, each row of the outer table is sent to one portion of the inner table, based on the splitting attributes of the inner table. The join occurs on this database partition. Figure 29 on page 276 provides an example.



The LINEITEM table is partitioned on the ORDERKEY column. The ORDERS table is partitioned on a different column. The ORDERS table is hashed and sent to the correct database partition of the LINEITEM table. In this example, the join predicate is assumed to be: `orders.orderkey = lineitem.orderkey`.

Figure 29. Directed Outer-Table Join Example

Directed inner-table and outer-table joins

In the directed inner-table and outer-table join strategy, rows of both the outer and inner tables are directed to a set of database partitions, based on the values of the joining columns. The join occurs on these database partitions. Figure 30 on page 277 provides an example.

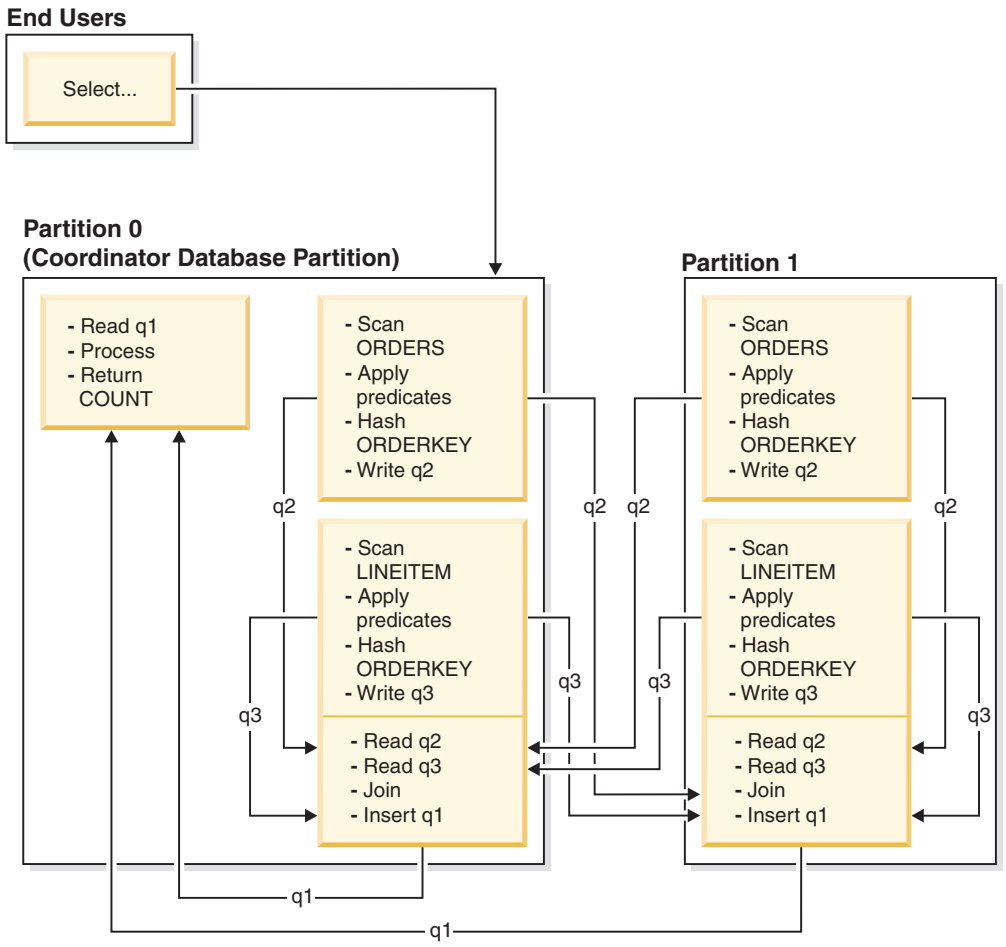
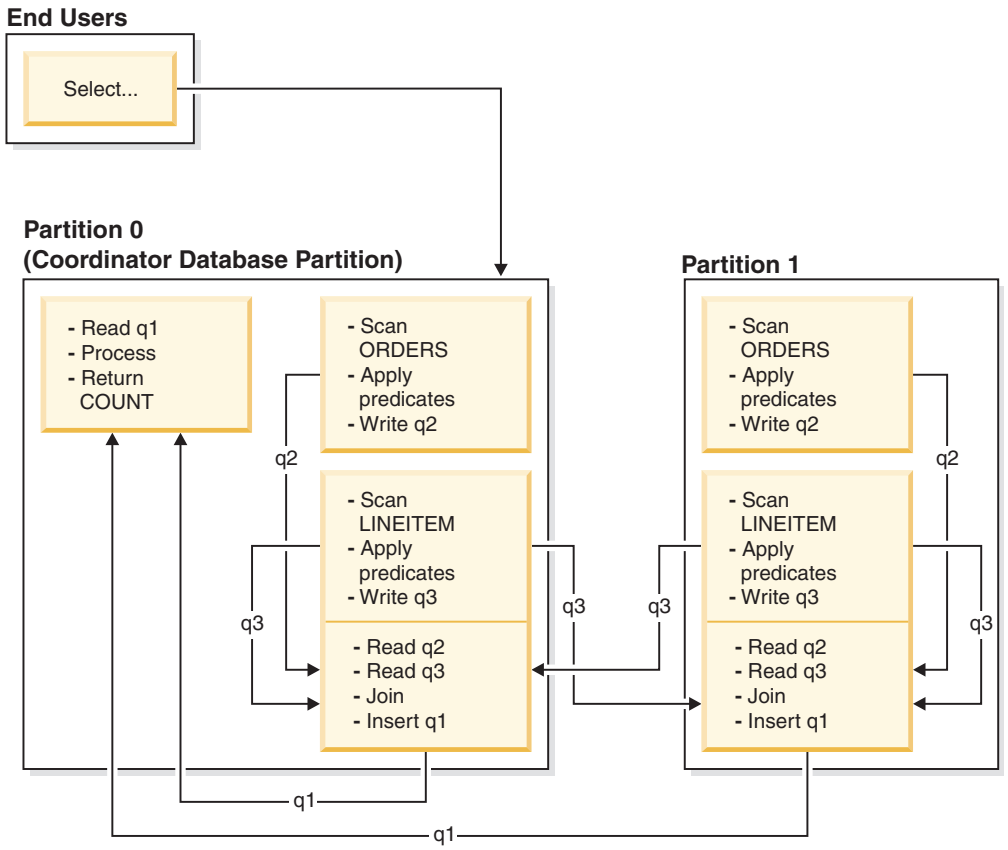


Figure 30. Directed Inner-Table and Outer-Table Join Example

Neither table is partitioned on the ORDERKEY column. Both tables are hashed and sent to new database partitions, where they are joined. Both table queue q2 and q3 are directed. In this example, the join predicate is assumed to be: `orders.orderkey = lineitem.orderkey`.

Broadcast inner-table joins

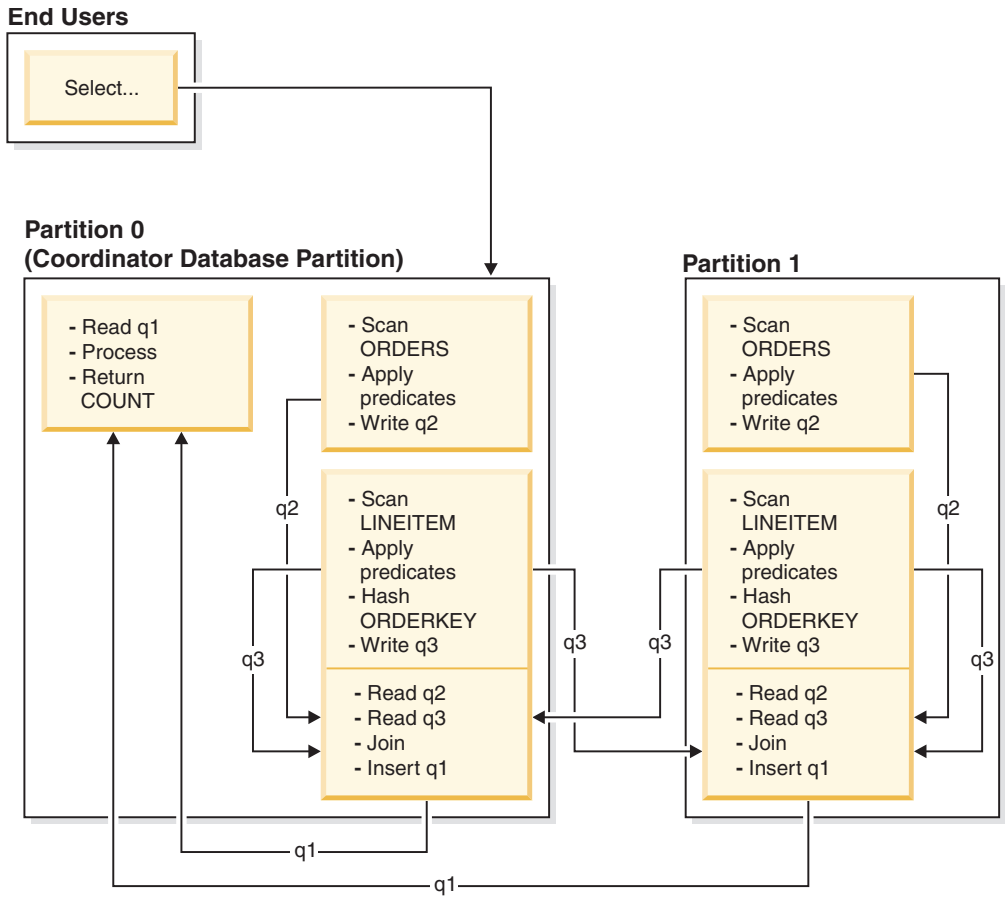
In the broadcast inner-table join strategy, the inner table is broadcast to all the database partitions of the outer table. Figure 31 on page 278 provides an example.



The LINEITEM table is sent to all database partitions that have the ORDERS table. Table queue q3 is broadcast to all database partitions of the outer table.
Figure 31. Broadcast Inner-Table Join Example

Directed inner-table joins

In the directed inner-table join strategy, each row of the inner table is sent to one database partition of the outer table, based on the splitting attributes of the outer table. The join occurs on this database partition. Figure 32 on page 279 provides an example.



The ORDERS table is partitioned on the ORDERKEY column. The LINEITEM table is partitioned on a different column. The LINEITEM table is hashed and sent to the correct database partition of the ORDERS table. In this example, the join predicate is assumed to be: `orders.orderkey = lineitem.orderkey`.

Figure 32. Directed Inner-Table Join Example

Effects of sorting and grouping on query optimization

When the optimizer chooses an access plan, it considers the performance impact of sorting data. Sorting occurs when no index satisfies the requested ordering of fetched rows. Sorting might also occur when the optimizer determines that a sort is less expensive than an index scan.

The optimizer handles sorted data in one of the following ways:

- It pipes the results of the sort when the query executes
- It lets the database manager handle the sort internally

Piped and non-piped sorts

If the final sorted list of data can be read in a single sequential pass, the results can be *piped*. Piping is quicker than non-piped ways of communicating the results of a sort. The optimizer chooses to pipe the results of a sort whenever possible.

Whether or not a sort is piped, the sort time depends on a number of factors, including the number of rows to be sorted, the key size and the row width. If the rows to be sorted occupy more than the space that is available in the sort heap,

several sort passes are performed. A subset of the entire set of rows is sorted during each pass. Each pass is stored in a temporary table in the buffer pool. If there is not enough space in the buffer pool, pages from this temporary table might be written to disk. When all of the sort passes are complete, these sorted subsets are merged into a single sorted set of rows. If the sort is piped, the rows are passed directly to Relational Data Services (RDS) as they are merged. (RDS is the Db2 component that processes requests to access or manipulate the contents of a database.)

Group and sort pushdown operators

In some cases, the optimizer can choose to push down a sort or aggregation operation to Data Management Services (DMS) from RDS. (DMS is the Db2 component that controls creating, removing, maintaining, and accessing the tables and table data in a database.) Pushing down these operations improves performance by allowing DMS to pass data directly to a sort or aggregation routine. Without this pushdown, DMS first passes this data to RDS, which then interfaces with the sort or aggregation routines. For example, the following query benefits from this type of optimization:

```
select workdept, avg(salary) as avg_dept_salary
  from employee
  group by workdept
```

Group operations in sorts

When sorting produces the order that is required for a GROUP BY operation, the optimizer can perform some or all of the GROUP BY aggregations while doing the sort. This is advantageous if the number of rows in each group is large. It is even more advantageous if doing some of the grouping during the sort reduces or eliminates the need for the sort to spill to disk.

Aggregation during sorting requires one or more of the following three stages of aggregation to ensure that proper results are returned.

- The first stage of aggregation, *partial aggregation*, calculates the aggregate values until the sort heap is filled. During partial aggregation, non-aggregated data is taken in and partial aggregates are produced. If the sort heap is filled, the rest of the data spills to disk, including all of the partial aggregations that have been calculated in the current sort heap. After the sort heap is reset, new aggregations are started.
- The second stage of aggregation, *intermediate aggregation*, takes all of the spilled sort runs and aggregates further on the grouping keys. The aggregation cannot be completed, because the grouping key columns are a subset of the distribution key columns. Intermediate aggregation uses existing partial aggregates to produce new partial aggregates. This stage does not always occur. It is used for both intrapartition and interpartition parallelism. In intrapartition parallelism, the grouping is finished when a global grouping key is available. In interpartition parallelism, this occurs when the grouping key is a subset of the distribution key dividing groups across database partitions, and thus requires redistribution to complete the aggregation. A similar case exists in intrapartition parallelism, when each agent finishes merging its spilled sort runs before reducing to a single agent to complete the aggregation.
- The last stage of aggregation, *final aggregation*, uses all of the partial aggregates and produces final aggregates. This step always takes place in a GROUP BY operator. Sorting cannot perform complete aggregation, because it cannot be guaranteed that the sort will not split. Complete aggregation takes in

non-aggregated data and produces final aggregates. This method of aggregation is usually used to group data that is already in the correct order.

Optimization strategies

The optimization strategies are dependent on the configuration of the Db2 environment. You need be aware of this configuration while you are designing performance improvements.

Optimization strategies for intrapartition parallelism:

The optimizer can choose an access plan to execute a query in parallel within a single database partition if a degree of parallelism is specified when the SQL statement is compiled.

At run time, multiple database agents called subagents are created to execute the query. The number of subagents is less than or equal to the degree of parallelism that was specified when the SQL statement was compiled.

To parallelize an access plan, the optimizer divides it into a portion that is run by each subagent and a portion that is run by the coordinating agent. The subagents pass data through table queues to the coordinating agent or to other subagents. In a partitioned database environment, subagents can send or receive data through table queues from subagents in other database partitions.

intrapartition parallel scan strategies

Relational scans and index scans can be performed in parallel on the same table or index. For parallel relational scans, the table is divided into ranges of pages or rows, which are assigned to subagents. A subagent scans its assigned range and is assigned another range when it has completed work on the current range.

For parallel index scans, the index is divided into ranges of records based on index key values and the number of index entries for a key value. The parallel index scan proceeds like a parallel table scan, with subagents being assigned a range of records. A subagent is assigned a new range when it has completed work on the current range.

Parallel table scans can be run against range partitioned tables, and similarly, parallel index scans can be run against partitioned indexes. For a parallel scan, partitioned indexes are divided into ranges of records, based on index key values and the number of key entries for a key value. When a parallel scan begins, subagents are assigned a range of records, and once the subagent completes a range, it is assigned a new range. The index partitions are scanned sequentially with subagents potentially scanning unreserved index partitions at any point in time without waiting for each other. Only the subset of index partitions that is relevant to the query based on data partition elimination analysis is scanned.

The optimizer determines the scan unit (either a page or a row) and the scan granularity.

Parallel scans provide an even distribution of work among the subagents. The goal of a parallel scan is to balance the load among the subagents and to keep them equally busy. If the number of busy subagents equals the number of available processors, and the disks are not overworked with I/O requests, the machine resources are being used effectively.

Other access plan strategies might cause data imbalance as the query executes. The optimizer chooses parallel strategies that maintain data balance among subagents.

intrapartition parallel sort strategies

The optimizer can choose one of the following parallel sort strategies:

- Round-robin sort

  This is also known as a *redistribution sort*. This method uses shared memory to efficiently redistribute the data as evenly as possible to all subagents. It uses a round-robin algorithm to provide the even distribution. It first creates an individual sort for each subagent. During the insert phase, subagents insert into each of the individual sorts in a round-robin fashion to achieve a more even distribution of data.
- Partitioned sort

  This is similar to the round-robin sort in that a sort is created for each subagent. The subagents apply a hash function to the sort columns to determine into which sort a row should be inserted. For example, if the inner and outer tables of a merge join are a partitioned sort, a subagent can use merge join to join the corresponding table portions and execute in parallel.
- Replicated sort

  This sort is used if each subagent requires all of the sort output. One sort is created and subagents are synchronized as rows are inserted into the sort. When the sort is complete, each subagent reads the entire sort. If the number of rows is small, this sort can be used to rebalance the data stream.
- Shared sort

  This sort is the same as a replicated sort, except that subagents open a parallel scan on the sorted result to distribute the data among the subagents in a way that is similar to a round-robin sort.

intrapartition parallel temporary tables

Subagents can cooperate to produce a temporary table by inserting rows into the same table. This is called a *shared temporary table*. The subagents can open private scans or parallel scans on the shared temporary table, depending on whether the data stream is to be replicated or split.

intrapartition parallel aggregation strategies

Aggregation operations can be performed by subagents in parallel. An aggregation operation requires the data to be ordered on the grouping columns. If a subagent can be guaranteed to receive all the rows for a set of grouping column values, it can perform a complete aggregation. This can happen if the stream is already split on the grouping columns because of a previous partitioned sort.

Otherwise, the subagent can perform a partial aggregation and use another strategy to complete the aggregation. Some of these strategies are:

- Send the partially aggregated data to the coordinator agent through a merging table queue. The coordinator agent completes the aggregation.
- Insert the partially aggregated data into a partitioned sort. The sort is split on the grouping columns and guarantees that all rows for a set of grouping columns are contained in one sort partition.

- If the stream needs to be replicated to balance processing, the partially aggregated data can be inserted into a replicated sort. Each subagent completes the aggregation using the replicated sort, and receives an identical copy of the aggregation result.

intrapartition parallel join strategies

Join operations can be performed by subagents in parallel. Parallel join strategies are determined by the characteristics of the data stream.

A join can be parallelized by partitioning or by replicating the data stream on the inner and outer tables of the join, or both. For example, a nested-loop join can be parallelized if its outer stream is partitioned for a parallel scan and the inner stream is again evaluated independently by each subagent. A merged join can be parallelized if its inner and outer streams are value-partitioned for partitioned sorts.

Data filtering and data skew can cause workloads between subagents to become imbalanced while a query executes. The inefficiency of imbalanced workloads is magnified by joins and other computationally expensive operations. The optimizer looks for sources of imbalance in the query's access plan and applies a balancing strategy, ensuring that work is evenly divided between the subagents. For an unordered outer data stream, the optimizer balances the join using the REBAL operator on the outer data stream. For an ordered data stream (where ordered data is produced by an index access or a sort), the optimizer balances the data using a shared sort. A shared sort will be not be used if the sort overflows into the temporary tables, due to the high cost of a sort overflow.

Optimization strategies for MDC tables:

If you create multidimensional clustering (MDC) tables, the performance of many queries might improve, because the optimizer can apply additional optimization strategies. These strategies are primarily based on the improved efficiency of block indexes, but the advantage of clustering on more than one dimension also permits faster data retrieval.

MDC table optimization strategies can also exploit the performance advantages of intrapartition parallelism and interpartition parallelism. Consider the following specific advantages of MDC tables:

- Dimension block index lookups can identify the required portions of the table and quickly scan only the required blocks.
- Because block indexes are smaller than record identifier (RID) indexes, lookups are faster.
- Index ANDing and ORing can be performed at the block level and combined with RIDs.
- Data is guaranteed to be clustered on extents, which makes retrieval faster.
- Rows can be deleted faster when rollout can be used.

Consider the following simple example for an MDC table named SALES with dimensions defined on the REGION and MONTH columns:

```
select * from sales
  where month = 'March' and region = 'SE'
```

For this query, the optimizer can perform a dimension block index lookup to find blocks in which the month of March and the SE region occur. Then it can scan only those blocks to quickly fetch the result set.

Rollout deletion

When conditions permit delete using rollout, this more efficient way to delete rows from MDC tables is used. The required conditions are:

- The DELETE statement is a searched DELETE, not a positioned DELETE (the statement does not use the WHERE CURRENT OF clause).
- There is no WHERE clause (all rows are to be deleted), or the only conditions in the WHERE clause apply to dimensions.
- The table is not defined with the DATA CAPTURE CHANGES clause.
- The table is not the parent in a referential integrity relationship.
- The table does not have ON DELETE triggers defined.
- The table is not used in any MQTs that are refreshed immediately.
- A cascaded delete operation might qualify for rollout if its foreign key is a subset of the table's dimension columns.
- The DELETE statement cannot appear in a SELECT statement executing against the temporary table that identifies the set of affected rows prior to a triggering SQL operation (specified by the OLD TABLE AS clause on the CREATE TRIGGER statement).

During a rollout deletion, the deleted records are not logged. Instead, the pages that contain the records are made to look empty by reformatting parts of the pages. The changes to the reformatted parts are logged, but the records themselves are not logged.

The default behavior, *immediate cleanup rollout*, is to clean up RID indexes at delete time. This mode can also be specified by setting the **DB2_MDC_ROLLOUT** registry variable to IMMEDIATE, or by specifying IMMEDIATE on the SET CURRENT MDC ROLLOUT MODE statement. There is no change in the logging of index updates, compared to a standard delete operation, so the performance improvement depends on how many RID indexes there are. The fewer RID indexes, the better the improvement, as a percentage of the total time and log space.

An estimate of the amount of log space that is saved can be made with the following formula:

```
S + 38*N - 50*P
```

where *N* is the number of records deleted, *S* is total size of the records deleted, including overhead such as null indicators and VARCHAR lengths, and *P* is the number of pages in the blocks that contain the deleted records. This figure is the reduction in actual log data. The savings on active log space required is double that value, due to the saving of space that was reserved for rollback.

Alternatively, you can have the RID indexes updated after the transaction commits, using *deferred cleanup rollout*. This mode can also be specified by setting the **DB2_MDC_ROLLOUT** registry variable to DEFER, or by specifying DEFERRED on the SET CURRENT MDC ROLLOUT MODE statement. In a deferred rollout, RID indexes are cleaned up asynchronously in the background after the delete commits. This method of rollout can result in significantly faster deletion times for very large deletes, or when a number of RID indexes exist on the table. The speed of the

overall cleanup operation is increased, because during a deferred index cleanup, the indexes are cleaned up in parallel, whereas in an immediate index cleanup, each row in the index is cleaned up one by one. Moreover, the transactional log space requirement for the DELETE statement is significantly reduced, because the asynchronous index cleanup logs the index updates by index page instead of by index key.

Note: Deferred cleanup rollout requires additional memory resources, which are taken from the database heap. If the database manager is unable to allocate the memory structures it requires, the deferred cleanup rollout fails, and a message is written to the administration notification log.

When to use a deferred cleanup rollout

If delete performance is the most important factor, and there are RID indexes defined on the table, use deferred cleanup rollout. Note that prior to index cleanup, index-based scans of the rolled-out blocks suffer a small performance penalty, depending on the amount of rolled-out data. The following issues should also be considered when deciding between immediate index cleanup and deferred index cleanup:

- Size of the delete operation

  Choose deferred cleanup rollout for very large deletions. In cases where dimensional DELETE statements are frequently issued on many small MDC tables, the overhead to asynchronously clean index objects might outweigh the benefit of time saved during the delete operation.
- Number and type of indexes

  If the table contains a number of RID indexes, which require row-level processing, use deferred cleanup rollout.
- Block availability

  If you want the block space freed by the delete operation to be available immediately after the DELETE statement commits, use immediate cleanup rollout.
- Log space

  If log space is limited, use deferred cleanup rollout for large deletions.
- Memory constraints

  Deferred cleanup rollout consumes additional database heap space on all tables that have deferred cleanup pending.

To disable rollout behavior during deletions, set the **DB2_MDC_ROLLOUT** registry variable to OFF or specify NONE on the SET CURRENT MDC ROLLOUT MODE statement.

Note: In Db2 Version 9.7 and later releases, deferred cleanup rollout is not supported on a data partitioned MDC table with partitioned RID indexes. Only the NONE and IMMEDIATE modes are supported. The cleanup rollout type will be IMMEDIATE if the **DB2_MDC_ROLLOUT** registry variable is set to DEFER, or if the CURRENT MDC ROLLOUT MODE special register is set to DEFERRED to override the **DB2_MDC_ROLLOUT** setting.

If only nonpartitioned RID indexes exist on the MDC table, deferred index cleanup rollout is supported.

Optimization strategies for partitioned tables:

Data partition elimination refers to the database server's ability to determine, based on query predicates, that only a subset of the data partitions in a table need to be accessed to answer a query. Data partition elimination is particularly useful when running decision support queries against a partitioned table.

A partitioned table uses a data organization scheme in which table data is divided across multiple storage objects, called data partitions or ranges, according to values in one or more table partitioning key columns of the table. Data from a table is partitioned into multiple storage objects based on specifications provided in the PARTITION BY clause of the CREATE TABLE statement. These storage objects can be in different table spaces, in the same table space, or a combination of both.

The following example demonstrates the performance benefits of data partition elimination.

```
create table custlist(
  subsdate date, province char(2), accountid int)
  partition by range(subsdate) (
    starting from '1/1/1990' in ts1,
    starting from '1/1/1991' in ts1,
    starting from '1/1/1992' in ts1,
    starting from '1/1/1993' in ts2,
    starting from '1/1/1994' in ts2,
    starting from '1/1/1995' in ts2,
    starting from '1/1/1996' in ts3,
    starting from '1/1/1997' in ts3,
    starting from '1/1/1998' in ts3,
    starting from '1/1/1999' in ts4,
    starting from '1/1/2000' in ts4,
    starting from '1/1/2001'
    ending '12/31/2001' in ts4)
```

Assume that you are only interested in customer information for the year 2000.

```
select * from custlist
  where subsdate between '1/1/2000' and '12/31/2000'
```

As Figure 33 on page 287 shows, the database server determines that only one data partition in table space TS4 must be accessed to resolve this query.

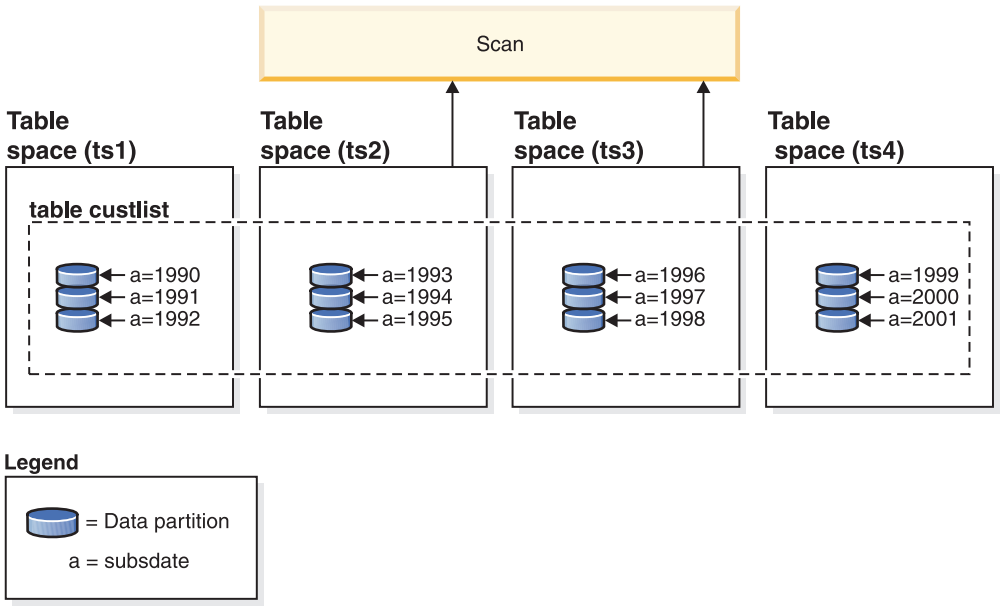


Figure 33. The performance benefits of data partition elimination

Another example of data partition elimination is based on the following scheme:

```
create table multi (
  sale_date date, region char(2))
  partition by (sale_date) (
    starting '01/01/2005'
    ending '12/31/2005'
    every 1 month)

create index sx on multi(sale_date)

create index rx on multi(region)
```

Assume that you issue the following query:

```
select * from multi
  where sale_date between '6/1/2005'
    and '7/31/2005' and region = 'NW'
```

Without table partitioning, one likely plan is index ANDing. Index ANDing performs the following tasks:

- Reads all relevant index entries from each index
- Saves both sets of row identifiers (RIDs)
- Matches RIDs to determine which occur in both indexes
- Uses the RIDs to fetch the rows

As Figure 34 on page 288 demonstrates, with table partitioning, the index is read to find matches for both REGION and SALE_DATE, resulting in the fast retrieval of matching rows.

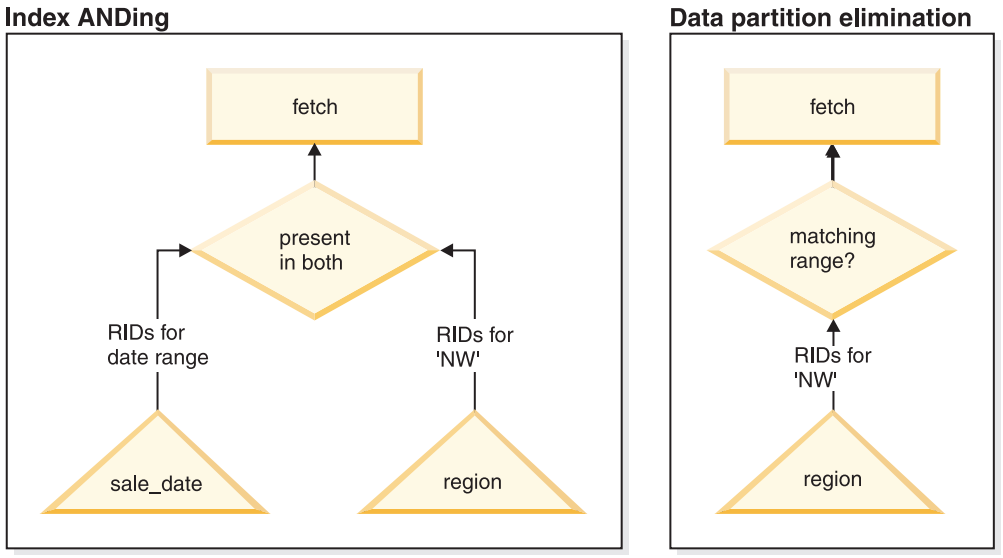


Figure 34. Optimizer decision path for both table partitioning and index ANDing

Db2 Explain

You can also use the explain facility to determine the data partition elimination plan that was chosen by the query optimizer. The "DP Elim Predicates" information shows which data partitions are scanned to resolve the following query:

```
   select * from custlist
     where subsdate between '12/31/1999' and '1/1/2001'
Arguments:
---------
DPESTFLG: (Number of data partitions accessed are Estimated)
    FALSE
DPLSTPRT: (List of data partitions accessed)
    9-11
DPNUMPRT: (Number of data partitions accessed)
   3

DP Elim Predicates:
------------------
Range 1)
  Stop  Predicate: (Q1.A <= '01/01/2001')
 Start Predicate: ('12/31/1999' <= Q1.A)



Objects Used in Access Plan:
 --------------------------

  Schema: MRSRINI
  Name:    CUSTLIST
  Type:    Data Partitioned Table
   Time of creation:      2005-11-30-14.21.33.857039
   Last statistics update:    2005-11-30-14.21.34.339392
    Number of columns:      3
    Number of rows:      100000
   Width of rows:      19
   Number of buffer pool pages:    1200
    Number of data partitions:    12
    Distinct row values:      No
    Tablespace name:      <VARIOUS>
```

Multi-column support

Data partition elimination works in cases where multiple columns are used as the table partitioning key. For example:

```
create table sales (
  year int, month int)
 partition by range(year, month) (
    starting from (2001,1)
    ending at (2001,3) in ts1,
   ending at (2001,6) in ts2,
   ending at (2001,9) in ts3,
   ending at (2001,12) in ts4,
   ending at (2002,3) in ts5,
   ending at (2002,6) in ts6,
   ending at (2002,9) in ts7,
   ending at (2002,12) in ts8)

select * from sales where year = 2001 and month < 8
```

The query optimizer deduces that only data partitions in TS1, TS2, and TS3 must be accessed to resolve this query.

Note: In the case where multiple columns make up the table partitioning key, data partition elimination is only possible when you have predicates on the leading columns of the composite key, because the non-leading columns that are used for the table partitioning key are not independent.

Multi-range support

It is possible to obtain data partition elimination with data partitions that have multiple ranges (that is, those that are ORed together). Using the SALES table that was created in the previous example, execute the following query:

```
select * from sales
  where (year = 2001 and month <= 3)
    or (year = 2002 and month >= 10)
```

The database server only accesses data for the first quarter of 2001 and the last quarter of 2002.

Generated columns

You can use generated columns as table partitioning keys. For example:

```
create table sales (
  a int, b int generated always as (a / 5))
  in ts1,ts2,ts3,ts4,ts5,ts6,ts7,ts8,ts9,ts10
  partition by range(b) (
    starting from (0)
    ending at (1000) every (50))
```

In this case, predicates on the generated column are used for data partition elimination. In addition, when the expression that is used to generate the columns is monotonic, the database server translates predicates on the source columns into predicates on the generated columns, which enables data partition elimination on the generated columns. For example:

```
select * from sales where a > 35
```

The database server generates an extra predicate on b (b > 7) from a (a > 35), thus allowing data partition elimination.

Join predicates

Join predicates can also be used in data partition elimination, if the join predicate is pushed down to the table access level. The join predicate is only pushed down to the table access level on the inner join of a nested loop join (NLJN).

Consider the following tables:

```
create table t1 (a int, b int)
  partition by range(a,b) (
    starting from (1,1)
    ending (1,10) in ts1,
    ending (1,20) in ts2,
    ending (2,10) in ts3,
    ending (2,20) in ts4,
    ending (3,10) in ts5,
    ending (3,20) in ts6,
    ending (4,10) in ts7,
    ending (4,20) in ts8)

create table t2 (a int, b int)
```

The following two predicates will be used:

```
P1: T1.A = T2.A
P2: T1.B > 15
```

In this example, the exact data partitions that will be accessed at compile time cannot be determined, due to unknown outer values of the join. In this case, as well as cases where host variables or parameter markers are used, data partition elimination occurs at run time when the necessary values are bound.

During run time, when T1 is the inner of an NLJN, data partition elimination occurs dynamically, based on the predicates, for every outer value of T2.A. During run time, the predicates T1.A = 3 and T1.B > 15 are applied for the outer value T2.A = 3, which qualifies the data partitions in table space TS6 to be accessed.

Suppose that column A in tables T1 and T2 have the following values:

Outer table T2: column A	Inner table T1: column A	Inner table T1: column B	Inner table T1: data partition location
2	3	20	TS6
3	2	10	TS3
3	2	18	TS4
	3	15	TS6
	1	40	TS3

To perform a nested loop join (assuming a table scan for the inner table), the database manager performs the following steps:

1. Reads the first row from T2. The value for A is 2.
2. Binds the T2.A value (which is 2) to the column T2.A in the join predicate T1.A = T2.A. The predicate becomes T1.A = 2.
3. Applies data partition elimination using the predicates T1.A = 2 and T1.B > 15. This qualifies data partitions in table space TS4.
4. After applying T1.A = 2 and T1.B > 15, scans the data partitions in table space TS4 of table T1 until a row is found. The first qualifying row found is row 3 of T1.
5. Joins the matching row.

6. Scans the data partitions in table space TS4 of table T1 until the next match (T1.A = 2 and T1.B > 15) is found. No more rows are found.
7. Repeats steps 1 through 6 for the next row of T2 (replacing the value of A with 3) until all the rows of T2 have been processed.

Indexes over XML data

Starting in Db2 Version 9.7 Fix Pack 1, you can create an index over XML data on a partitioned table as either partitioned or nonpartitioned. The default is a partitioned index.

Partitioned and nonpartitioned XML indexes are maintained by the database manager during table insert, update, and delete operations in the same way as any other relational indexes on a partitioned table are maintained. Nonpartitioned indexes over XML data on a partitioned table are used in the same way as indexes over XML data on a nonpartitioned table to speed up query processing. Using the query predicate, it is possible to determine that only a subset of the data partitions in the partitioned table need to be accessed to answer the query.

Data partition elimination and indexes over XML columns can work together to enhance query performance. Consider the following partitioned table:

```
create table employee (a int, b xml, c xml)
  index in tbspx
  partition by (a) (
    starting 0 ending 10,
    ending 20,
    ending 30,
    ending 40)
```

Now consider the following query:

```
select * from employee
  where a > 21
  and xmlexist('$doc/Person/Name/First[.="Eric"]'
    passing "EMPLOYEE"."B" as "doc")
```

The optimizer can immediately eliminate the first two partitions based on the predicate `a > 21`. If the nonpartitioned index over XML data on column B is chosen by the optimizer in the query plan, an index scan using the index over XML data will be able to take advantage of the data partition elimination result from the optimizer and only return results belonging to partitions that were not eliminated by the relational data partition elimination predicates.

Query optimization with materialized query tables

Materialized query tables (MQTs) are a powerful way to improve response time for complex analytical queries because their data consists of precomputed results from the tables that you specify in the materialized query table definitions.

MQTs can help improve response time particularly for queries that use one or more of the following types of data:

- Aggregate data over one or more dimensions
- Joins and aggregate data over a group of tables
- Data from a commonly accessed subset of data
- Repartitioned data from a table, or part of a table, in a partitioned database environment

The larger the base tables, the more significant are the potential improvements in response time when you use MQTs.

Knowledge of MQTs is integrated into the SQL and XQuery compiler. During the query rewrite phase, the optimizer determines whether to use an available MQT in place of accessing the referenced base tables directly. If an MQT is used, you need access privileges on the base tables, not the MQT, and the explain facility can provide information about which MQT was selected.

MQTs can effectively eliminate overlapping work among queries. Computations are performed only once when MQTs are built and once each time that they are refreshed, and their content can be reused during the execution of many queries.

Because MQTs behave like regular tables in many ways, use the same guidelines for optimizing data access, such as using table space definitions and indexes and running the **RUNSTATS** utility.

Db2 Cancun Release 10.5.0.4 and later includes support for column-organized user-maintained MQTs.

If you are upgrading your Db2 server from version 10.1 or earlier releases and you plan to convert row-organized tables with existing MQTs to column-organized tables, using column-organized user-maintained MQTs can simplify portability and improve query performance.

Materialized query table restrictions:

Materialized query tables (MQTs), including shadow tables, are subject to certain restrictions.

Fullselect statements that form part of the definition of MQTs are subject to the following restrictions:

- Every select element must have a name.
- A fullselect must not reference any of the following object types:
  - Created global temporary tables
  - Declared global temporary tables
  - Materialized query tables
  - Nicknames that you created by using the DISALLOW CACHING clause of the CREATE NICKNAME or ALTER NICKNAME statement
  - Protected tables
  - Staging tables
  - System catalog tables
  - Typed tables
  - Views that violate any MQT restrictions
  - Views that directly or indirectly depend on protected tables
- A fullselect must not contain any column references or expressions that involve the following data types:
  - LOB
  - LONG
  - XML
  - Reference
  - User-defined structured types

  - Any distinct type that is based on these data types
- A fullselect must not contain any column references, expressions, or functions that meet the following criteria:
  - Depend on the physical characteristics of the data, such as DBPARTITIONNUM, HASHEDVALUE, RID_BIT, and RID
  - Depend on changes to the data, such as a row change expression or a row change timestamp column
  - Are defined as EXTERNAL ACTION
  - Are defined as LANGUAGE SQL, CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA
- A fullselect must not include a CONNECT BY clause.
- If you specify the MAINTAINED BY FEDERATED_TOOL option for the CREATE TABLE statement, the SELECT clause must not contain a reference to a base table.
- If you specify the REFRESH IMMEDIATE option for the CREATE TABLE statement, the following restrictions apply:
  - The CREATE TABLE statement for the MQT must not contain duplicate grouping sets.
  - The table must not be a system-maintained columnar MQT.
  - At least one unique key from each table that is referenced must be in the select list.
  - The fullselect must be a subselect. The exception is that UNION ALL is supported in the input table expression of a GROUP BY clause.
  - The input table expressions of a UNION ALL or a JOIN must not contain aggregate functions.
  - The fullselect must not contain any of the following items:
    - Aggregate functions without the fullselect also containing a GROUP BY clause
    - Any expressions that use the result of aggregate functions
    - Built-in functions that depend on the value of a special register
    - Functions that are not deterministic
    - OLAP functions
    - Recursive common table expressions
    - References to global variables
    - References to nicknames
    - References to special registers
    - Sampling functions
    - SELECT DISTINCT statements
    - Subqueries
    - Text functions
- If you specify the REFRESH IMMEDIATE option for the CREATE TABLE statement and the fullselect contains a GROUP BY clause, the following conditions and restrictions apply:
  - The select list must contain the COUNT() or COUNT_BIG() function.
  - For each nullable column C, if the select list contains the SUM(C) function, the COUNT(C) function is also required.
  - You must include the SUM() or GROUPING() aggregate function. You cannot include any other aggregate function.

  - You cannot specify the HAVING clause.
  - For partitioned database environments, the GROUP BY columns must contain the partitioning key of the materialized query table.
  - Nesting of aggregate functions is not allowed.
- If you specify the REFRESH IMMEDIATE option for the CREATE TABLE statement and the FROM clause references more than one table, an inner join that does not use the explicit INNER JOIN syntax is the only type of join that is supported.
- If you specify the DISTRIBUTED BY REPLICATION option for the CREATE TABLE statement, the following restrictions apply:
  - Aggregate functions and the GROUP BY clause are not allowed.
  - The MQT must reference only a single table. It cannot include a join, union, or subquery.
  - You cannot specify the PARTITIONING KEY clause.
  - Unique indexes are not allowed for system maintained MQTs.
- To create a user-maintained MQT that has a column-organized table as the base table, you must specify the MAINTAINED BY USER clause for the CREATE TABLE statement.

Column-organized MQT and shadow tables are subject to the following additional restrictions:

- Column-organized MQTs are not supported in Db2 pureScale environments.
- An MQT other than a shadow table must reference a table with the same organization as the MQT.
- You must specify the ORGANIZE BY COLUMN clause when creating a column-organized MQT.
- The MAINTAINED BY SYSTEM clause is supported for column-organized MQTs only when defined with REFRESH DEFERRED and the DISTRIBUTE BY REPLICATION options.

Shadow tables are subject to the following additional restrictions:

- MAINTAINED BY REPLICATION must be specified.
- REFRESH DEFERRED must be specified.
- REFRESH IMMEDIATE is unsupported.
- A primary key must be created on the shadow table and must correspond to a primary key or unique constraint on the base table.
- The CREATE TABLE statement or the fullselect must not reference any of the following object types:
  - Partitioned tables.
  - Multidimensional clustering (MDC) tables.
  - Range-clustered tables (RCTs).
  - Tables containing LONG VARCHAR or LONG VARGRAPHIC columns. These data types have been deprecated since Version 9.5 Fix Pack 1.
  - Tables that are protected with row and column access control (RCAC) or label-based access control (LBAC).
  - Temporal tables.
- A permission or mask cannot reference a shadow table.
- A base table can be referenced by only one shadow table.
- The following restrictions apply to the fullselect in a shadow table definition:

  - The fullselect can reference only one base table; joins are not supported.
  - The base table must be a row-organized table.
  - The subselect can contain only a select-clause and a from-clause.
  - The projection list of the shadow table can reference only base table columns that are valid in a column-organized table. Expressions are not supported in the projection list. You cannot rename the columns that are referenced in the projection list by using the column list or the AS clause.
  - The projection list of the shadow table must include at least one set of enforced unique constraint or primary key columns from the base table.
  - The fullselect cannot include references to a nickname, a typed table, or a view or contain the SELECT DISTINCT clause.
- A Materialized query table may not be created as a random distribution table.

Optimize query execution with user-maintained materialized query tables:

Review the following example to understand how to optimize query execution with user-maintained materialized query tables (MQTs).

MQTs can reduce the need for expensive joins, sorts, and aggregation of base data. Although the precise answer for a query is not included in the MQT, the cost of computing the answer by using an MQT could be significantly less than the cost of using a large base table, because a portion of the answer is already computed.

Example for optimization of multidimensional-analysis queries

Scenario description

Consider a database warehouse that contains a set of customers and a set of credit card accounts. The warehouse records the set of transactions that are made with the credit cards. Each transaction contains a set of items that are purchased together.

The schema is classified as a multi-star schema, because it has two large tables, one containing transaction items, and the other identifying the purchase transactions. The following hierarchical dimensions describe a transaction:

- The product hierarchy is stored in two normalized tables representing the product group and the product line.
- The location hierarchy contains city, state, and country, region, or territory information, and is stored in a single denormalized table.
- The time hierarchy contains day, month, and year information, and is encoded in a single date field. The date dimensions are extracted from the date field of the transaction using built-in functions.

Other tables in this schema represent account information for customers, and customer information.

A user-maintained MQT is created for sales at each level of the following hierarchies:

- Product
- Location
- Time composed of year, month, and day

Example of a user-maintained MQT

You can create user-maintained MQTs to satisfy many queries by storing aggregate data. The following CREATE TABLE statement shows how to

create an MQT that computes the sum and count of sales data along the product group and line dimensions; along the city, state, and country, region, or territory dimensions; and along the time dimension. It also includes several other columns in its GROUP BY clause.

```
create table dba.pg_salessum
  as (
    select l.id as prodline, pg.id as pgroup,
      loc.country, loc.state, loc.city,
      l.name as linename, pg.name as pgname,
      year(pdate) as year, month(pdate) as month,
      t.status,
      sum(ti.amount) as amount,
      count(*) as count
    from cube.transitem as ti, cube.trans as t,
      cube.loc as loc, cube.pgroup as pg, cube.prodline as l
    where
      ti.transid = t.id and
      ti.pgid = pg.id and
      pg.lineid = l.id and
      t.locid = loc.id and
      year(pdate) > 1990
    group by l.id, pg.id, loc.country, loc.state, loc.city,
      year(pdate), month(pdate), t.status, l.name, pg.name
  )
data initially deferred refresh deferred;

refresh table dba.pg_salessum;
```

The following queries can take advantage of the precomputed values in the *dba.pg_salessum* MQT:

- Sales by month and product group
- Total sales for the years after 1990
- Sales for 1995 or 1996
- The sum of sales for a specific product group or product line
- The sum of sales for a specific product group or product line in 1995 and 1996
- The sum of sales for a specific country, region, or territory

Example of a query that returns the total sales for 1995 and 1996

The following query obtains significant performance improvements because it uses the aggregated data in the *dba.pg_salessum* MQT.

```
set current refresh age=any

select year(pdate) as year, sum(ti.amount) as amount
  from cube.transitem as ti, cube.trans as t,
    cube.loc as loc, cube.pgroup as pg, cube.prodline as l
  where
    ti.transid = t.id and
    ti.pgid = pg.id and
    pg.lineid = l.id and
    t.locid = loc.id and
    year(pdate) in (1995, 1996)
  group by year(pdate);
```

Example of a query returns the total sales by product group for 1995 and 1996

The following query also benefits from using the aggregated data in the *dba.pg_salessum* MQT.

```
set current refresh age=any

select pg.id as "PRODUCT GROUP", sum(ti.amount) as amount
  from cube.transitem as ti, cube.trans as t,
    cube.loc as loc, cube.pgroup as pg, cube.prodline as l
```

```
where
  ti.transid = t.id and
  ti.pgid = pg.id and
  pg.lineid = l.id and
  t.locid = loc.id and
  year(pdate) in (1995, 1996)
group by pg.id;
```

Explain facility

The Db2 explain facility provides detailed information about the access plan that the optimizer chooses for an SQL or XQuery statement.

The information provided describes the decision criteria that are used to choose the access plan. The information can also help you to tune the statement or your instance configuration to improve performance. More specifically, explain information can help you with the following tasks:

- Understanding how the database manager accesses tables and indexes to satisfy your query.
- Evaluating your performance-tuning actions. After altering a statement or making a configuration change, examine the new explain information to determine how your action has affected performance.

The captured information includes the following information:

- The sequence of operations that were used to process the query
- Cost information
- Predicates and selectivity estimates for each predicate
- Statistics for all objects that were referenced in the SQL or XQuery statement at the time that the explain information was captured
- Values for host variables, parameter markers, or special registers that were used to reoptimize the SQL or XQuery statement

The explain facility is invoked by issuing the EXPLAIN statement, which captures information about the access plan chosen for a specific explainable statement and writes this information to explain tables. You must create the explain tables prior to issuing the EXPLAIN statement. You can also set CURRENT EXPLAIN MODE or CURRENT EXPLAIN SNAPSHOT, special registers that control the behavior of the explain facility.

For privileges and authorities that are required to use the explain utility, see the description of the EXPLAIN statement. The EXPLAIN authority can be granted to an individual who requires access to explain information but not to the data that is stored in the database. This authority is a subset of the database administrator authority and has no inherent privilege to access data stored in tables.

To display explain information, you can use a command-line tool. The tool that you use determines how you set the special registers that control the behavior of the explain facility. If you expect to perform detailed analysis with one of the command-line utilities or with custom SQL or XQuery statements against the explain tables, capture all explain information.

In IBM Data Studio Version 3.1 or later, you can generate a diagram of the current access plan for an SQL or XPATH statement. For more details, see Diagramming access plans with Visual Explain.

Tuning SQL statements using the explain facility:

The explain facility is used to display the query access plan that was chosen by the query optimizer to run an SQL statement.

It contains extensive details about the relational operations used to run the SQL statement, such as the plan operators, their arguments, order of execution, and costs. Because the query access plan is one of the most critical factors in query performance, it is important to understand explain facility output when diagnosing query performance problems.

Explain information is typically used to:

- Understand why application performance has changed
- Evaluate performance tuning efforts

Analyzing performance changes

To help you understand the reasons for changes in query performance, perform the following steps to obtain "before and after" explain information:

1. Capture explain information for the query before you make any changes, and save the resulting explain tables. Alternatively, you can save output from the **db2exfmt** utility. However, having explain information in the explain tables makes it easy to query them with SQL, and facilitates more sophisticated analysis. As well, it provides all of the obvious maintenance benefits of having data in a relational DBMS. The **db2exfmt** tool can be run at any time.
2. Save or print the current catalog statistics. You can also use the **db2look** command to help perform this task. In Db2 Version 9.7, you can collect an explain snapshot when the explain tables are populated. The explain snapshot contains all of the relevant statistics at the time that the statement is explained. The **db2exfmt** utility will automatically format the statistics that are contained in the snapshot. This is especially important when using automatic or real-time statistics collection, because the statistics used for query optimization might not yet be in the system catalog tables, or they might have changed between the time that the statement was explained and when the statistics were retrieved from the system catalog.
3. Save or print the data definition language (DDL) statements, including those for CREATE TABLE, CREATE VIEW, CREATE INDEX, and CREATE TABLESPACE. The **db2look** command will also perform this task.

The information that you collect in this way provides a reference point for future analysis. For dynamic SQL statements, you can collect this information when you run your application for the first time. For static SQL statements, you can also collect this information at bind time. It is especially important to collect this information before a major system change, such as the installation of a new service level or Db2 release, or before a significant configuration change, such as adding or dropping database partitions and redistributing data. This is because these types of system changes might result in an adverse change to access plans. Although access plan regression should be a rare occurrence, having this information available will help you to resolve performance regressions faster. To analyze a performance change, compare the information that you collected previously with information about the query and environment that you collect when you start your analysis.

As a simple example, your analysis might show that an index is no longer being used as part of an access plan. Using the catalog statistics information displayed by **db2exfmt**, you might notice that the number of index levels (NLEVELS column) is

now substantially higher than when the query was first bound to the database. You might then choose to perform one of the following actions:

- Reorganize the index
- Collect new statistics for your table and indexes
- Collect explain information when rebinding your query

After you perform one of these actions, examine the access plan again. If the index is being used once again, query performance might no longer be a problem. If the index is still not being used, or if performance is still a problem, try a second action and examine the results. Repeat these steps until the problem is resolved.

Evaluating performance tuning efforts

You can take a number of actions to help improve query performance, such as adjusting configuration parameters, adding containers, or collecting fresh catalog statistics.

After you make a change in any of these areas, you can use the explain facility to determine the affect, if any, that the change has had on the chosen access plan. For example, if you add an index or materialized query table (MQT) based on index guidelines, the explain data can help you to determine whether the index or materialized query table is actually being used as expected.

Although the explain output provides information that allows you to determine the access plan that was chosen and its relative cost, the only way to accurately measure the performance improvement for a query is to use benchmark testing techniques.

Explain tables and the organization of explain information:

An *explain instance* represents one invocation of the explain facility for one or more SQL or XQuery statements. The explain information that is captured in one explain instance includes information about the compilation environment and the access plan that is chosen to satisfy the SQL or XQuery statement that is being compiled.

For example, an explain instance might consist of any one of the following items:

- All eligible SQL or XQuery statements in one package, for static query statements. For SQL statements (including those that query XML data), you can capture explain information for CALL, compound SQL (dynamic), DELETE, INSERT, MERGE, REFRESH TABLE, SELECT, SELECT INTO, SET INTEGRITY, UPDATE, VALUES, and VALUES INTO statements. In the case of XQuery statements, you can obtain explain information for XQUERY db2-fn:xmlcolumn and XQUERY db2-fn:sqlquery statements.

  Note: REFRESH TABLE and SET INTEGRITY statements are compiled only dynamically.
- One particular SQL statement, for incremental bind SQL statements.
- One particular SQL statement, for dynamic SQL statements.
- Each EXPLAIN statement (dynamic or static).

The explain facility, which you can invoke by issuing the EXPLAIN statement or by using the section explain interfaces, captures information about the access plan that is chosen for a specific explainable statement and writes this information to explain tables. You must create the explain tables before issuing the EXPLAIN statement. To create the tables, use one of the following methods:

- Run the EXPLAIN.DDL script in the misc subdirectory of the sqllib subdirectory.
- Use the SYSPROC.SYSINSTALLOBJECTS procedure. You can also use this procedure to drop and validate explain tables.

You can create the tables under a specific schema and table space. You can find an example in the EXPLAIN.DDL file.

Explain tables can be common to more than one user. You can define tables for one user and then create aliases pointing to the defined tables for each additional user. Alternatively, you can define the explain tables under the SYSTOOLS schema. The explain facility uses the SYSTOOLS schema as the default if no explain tables or aliases are found under your session ID (for dynamic SQL or XQuery statements) or under the statement authorization ID (for static SQL or XQuery statements). Each user sharing common explain tables must hold the INSERT privilege on those tables.

The following table summarizes the purpose of each explain table.

Table 59. Summary of the explain tables

Table Name	Description
ADVISE_INDEX	Stores information about recommended indexes. The table can be populated by the query compiler or the **db2advis** command, or you can populate it. This table is used to get recommended indexes and to evaluate proposed indexes.
ADVISE_INSTANCE	Contains information about **db2advis** command execution, including start time. This table contains one row for each execution of the **db2advis** command.
ADVISE_MQT	Contains the following information: • The query that defines each recommended materialized query table (MQT) • The column statistics for each MQT, such as COLSTATS (in XML form) and NUMROWS • The sampling query to obtain detailed statistics for each MQT
ADVISE_PARTITION	Stores virtual database partitions that are generated and evaluated by the **db2advis** command.
ADVISE_TABLE	Stores the data definition language (DDL) statements for table creation, using the final Design Advisor recommendations for MQTs, multidimensional clustering tables (MDCs), and database partitioning.
ADVISE_WORKLOAD	Contains a row for each SQL or XQuery statement in a workload. The **db2advis** command uses this table to collect and store workload information.
EXPLAIN_ACTUALS	Contains the explain section actuals information.
EXPLAIN_ARGUMENT	Contains information about the unique characteristics of each operator, if any.
EXPLAIN_DIAGNOSTIC	Contains an entry for each diagnostic message that is produced for a particular instance of an explained statement in the EXPLAIN_STATEMENT table.

Table 59. Summary of the explain tables (continued)

Table Name	Description
EXPLAIN_DIAGNOSTIC_DATA	Contains message tokens for diagnostic messages that are recorded in the EXPLAIN_DIAGNOSTIC table. The message tokens provide additional information that is specific to the execution of the SQL statement that generated the message.
EXPLAIN_INSTANCE	Is the main control table for all explain information. Each row in the explain tables is linked to a unique row in this table. Basic information about the source of the SQL or XQuery statements being explained and environmental information are kept in this table.
EXPLAIN_OBJECT	Identifies the data objects that are required by the access plan that is generated to satisfy an SQL or XQuery statement.
EXPLAIN_OPERATOR	Contains all of the operators that the query compiler needs to satisfy an SQL or XQuery statement.
EXPLAIN_PREDICATE	Identifies the predicates that are applied by a specific operator.
EXPLAIN_STATEMENT	Contains the text of the SQL or XQuery statement for the different levels of explain information. The SQL or XQuery statement that you issued and the version that the optimizer uses to choose an access plan are stored in this table. When an explain snapshot is requested, additional explain information is recorded to describe the access plan that was selected by the query optimizer. This information is stored in the SNAPSHOT column of the EXPLAIN_STATEMENT table.
EXPLAIN_STREAM	Represents the input and output data streams between individual operators and data objects. The operators are represented in the EXPLAIN_OPERATOR table. The data objects are represented in the EXPLAIN_OBJECT table.
OBJECT_METRICS	Contains runtime statistics for each object that is referenced in a specific execution of a section at a specific time. If object statistics are collected on multiple members, this table contains a row for each member on which the object was referenced. If object statistics are collected for a partitioned object, this table contains a row for each data partition. This table contains information only if the activity event monitor captures section actuals.

Explain information for data objects:

A single access plan might use one or more data objects to satisfy an SQL or XQuery statement.

Object statistics

The explain facility records information about each object, such as the following:

- The creation time
- The last time that statistics were collected for the object
- Whether or not the data in the object is sorted (only table or index objects)
- The number of columns in the object (only table or index objects)
- The estimated number of rows in the object (only table or index objects)
- The number of pages that the object occupies in the buffer pool
- The total estimated overhead, in milliseconds, for a single random I/O to the specified table space where the object is stored
- The estimated transfer rate, in milliseconds, to read a 4-KB page from the specified table space
- Prefetch and extent sizes, in 4-KB pages
- The degree of data clustering within the index
- The number of leaf pages that are used by the index for this object, and the number of levels in the tree
- The number of distinct full key values in the index for this object
- The total number of overflow records in the table

Explain information for data operators:

A single access plan can perform several operations on the data to satisfy the SQL or XQuery statement and provide results back to you. The query compiler determines the operations that are required, such as a table scan, an index scan, a nested loop join, or a group-by operator.

In addition to showing information about each operator that is used in an access plan, explain output also shows the cumulative effects of the access plan.

Estimated cost information

The following cumulative cost estimates for operators are recorded. These costs are for the chosen access plan, up to and including the operator for which the information is captured.

- The total cost (in timerons)
- The number of page I/Os
- The number of processing instructions
- The cost (in timerons) of fetching the first row, including any required initial overhead
- The communication cost (in frames)

A *timeron* is an invented relative unit of measurement. Timeron values are determined by the optimizer, based on internal values such as statistics that change as the database is used. As a result, the timeron values for an SQL or XQuery statement are not guaranteed to be the same every time an estimated cost in timerons is determined.

If there is more than one network adapter involved, the cumulative communication cost for the adapter with the highest value is returned. These communication cost values only include the costs of network traffic between physical machines. They do not include the cost of passing frames between node partitions on the same

physical machine in a partitioned database environment, which therefore do not flow across the network.

Operator properties

The following information that describes the properties of each operator is recorded by the explain facility:

- The set of tables that have been accessed
- The set of columns that have been accessed
- The columns on which the data is ordered, if the optimizer has determined that this ordering can be used by subsequent operators
- The set of predicates that have been applied
- The estimated number of rows that will be returned (cardinality)

Explain information for instances:

Explain instance information is stored in the EXPLAIN_INSTANCE table. Additional specific information about each query statement in an instance is stored in the EXPLAIN_STATEMENT table.

Explain instance identification

The following information helps you to identify a specific explain instance and to associate the information about certain statements with a specific invocation of the explain facility:

- The user who requested the explain information
- When the explain request began
- The name of the package that contains the explained statement
- The SQL schema of the package that contains the explained statement
- The version of the package that contains the statement
- Whether snapshot information was collected

Environmental settings

Information about the database manager environment in which the query compiler optimized your queries is captured. The environmental information includes the following:

- The version and release number of the Db2 product
- The degree of parallelism under which the query was compiled

  The CURRENT DEGREE special register, the DEGREE bind option, the **SET RUNTIME DEGREE** command, and the **dft_degree** database configuration parameter determine the degree of parallelism under which a particular query is compiled.
- Whether the statement is dynamic or static
- The query optimization class used to compile the query
- The type of row blocking for cursors that occurs when compiling the query
- The isolation level under which the query runs
- The values of various configuration parameters when the query was compiled. Values for the following parameters are recorded when an explain snapshot is taken:
  - Sort heap size (**sortheap**)
  - Average number of active applications (**avg_appls**)

- Database heap (**dbheap**)
- Maximum storage for lock list (**locklist**)
- Maximum percent of lock list before escalation (**maxlocks**)
- CPU speed (**cpuspeed**)
- Communications bandwidth (**comm_bandwidth**)

Statement identification

More than one statement might have been explained for each explain instance. In addition to information that uniquely identifies the explain instance, the following information helps to identify individual query statements:

- The type of statement: SELECT, DELETE, INSERT, UPDATE, positioned DELETE, positioned UPDATE, or SET INTEGRITY
- The statement and section number of the package issuing the statement, as recorded in the SYSCAT.STATEMENTS catalog view

The QUERYTAG and QUERYNO fields in the EXPLAIN_STATEMENT table contain identifiers that are set as part of the explain process. When EXPLAIN MODE or EXPLAIN SNAPSHOT is active, and dynamic explain statements are submitted during a command line processor (CLP) or call-level interface (CLI) session, the QUERYTAG value is set to “CLP” or “CLI”, respectively. In this case, the QUERYNO value defaults to a number that is incremented by one or more for each statement. For all other dynamic explain statements that are not from the CLP or CLI, or that do not use the EXPLAIN statement, the QUERYTAG value is set to blanks and QUERYNO is always 1.

Cost estimation

For each explained statement, the optimizer records an estimate of the relative cost of executing the chosen access plan. This cost is stated in an invented relative unit of measure called a *timeron*. No estimate of elapsed times is provided, for the following reasons:

- The query optimizer does not estimate elapsed time but only resource consumption.
- The optimizer does not model all factors that can affect elapsed time. It ignores factors that do not affect the efficiency of the access plan. A number of runtime factors affect the elapsed time, including the system workload, the amount of resource contention, the amount of parallel processing and I/O, the cost of returning rows to the user, and the communication time between the client and server.

Statement text

Two versions of the statement text are recorded for each explained statement. One version is the code that the query compiler receives from the application. The other version is reverse-translated from the internal (compiler) representation of the query. Although this translation looks similar to other query statements, it does not necessarily follow correct query language syntax, nor does it necessarily reflect the actual content of the internal representation as a whole. This translation is provided only to enable you to understand the context in which the optimizer chose the access plan. To understand how the compiler has rewritten your query for better optimization, compare the user-written statement text to the internal representation of the query statement. The rewritten statement also shows you

other factors that affect your statement, such as triggers or constraints. Some keywords that are used in this "optimized" text include the following:

$C*n* The name of a derived column, where *n* represents an integer value.

$CONSTRAINT$
This tag identifies a constraint that was added to the original statement during compilation, and is seen in conjunction with the $WITH_CONTEXT$ prefix.

$DERIVED.T*n*
The name of a derived table, where *n* represents an integer value.

$INTERNAL_FUNC$
This tag indicates the presence of a function that is used by the compiler for the explained query but that is not available for general use.

$INTERNAL_PRED$
This tag indicates the presence of a predicate that was added during compilation of the explained query but that is not available for general use. An internal predicate is used by the compiler to satisfy additional context that is added to the original statement because of triggers or constraints.

$INTERNAL_XPATH$
This tag indicates the presence of an internal table function that takes a single annotated XPath pattern as an input parameter and returns a table with one or more columns that match that pattern.

RID This tag identifies the row identifier (RID) column for a particular row.

$TRIGGER$
This tag identifies a trigger that was added to the original statement during compilation, and is seen in conjunction with the $WITH_CONTEXT$ prefix.

$WITH_CONTEXT$(...)
This prefix appears at the beginning of the text when additional triggers or constraints have been added to the original query statement. A list of the names of any triggers or constraints that affect the compilation and resolution of the statement appears after this prefix.

Guidelines for capturing explain information:

Explain data can be captured by request when an SQL or XQuery statement is compiled.

If incremental bind SQL or XQuery statements are compiled at run time, data is placed in the explain tables at run time, not at bind time. For these statements, the inserted explain table qualifier and authorization ID are that of the package owner, not of the user running the package.

Explain information is captured only when an SQL or XQuery statement is compiled. After initial compilation, dynamic query statements are recompiled when a change to the environment requires it, or when the explain facility is active. If you issue the same PREPARE statement for the same query statement, the query is compiled and explain data is captured every time that this statement is prepared or executed.

If a package is bound using the **REOPT ONCE** or **REOPT ALWAYS** bind option, SQL or XQuery statements containing host variables, parameter markers, global variables,

or special registers are compiled, and the access path is created using real values for these variables if they are known, or default estimates if the values are not known at compilation time.

If the **REOPT ONCE** option is used, an attempt is made to match the specified SQL or XQuery statement with the same statement in the package cache. Values for this already re-optimized and cached query statement will be used to re-optimize the specified query statement. If the user has the required access privileges, the explain tables will contain the newly re-optimized access plan and the values that were used for re-optimization.

In a multi-partition database system, the statement should be explained on the same database partition on which it was originally compiled and re-optimized using **REOPT ONCE**; otherwise, an error is returned.

Capturing information in the explain tables

- Static or incremental bind SQL and XQuery statements

  Specify either **EXPLAIN ALL** or **EXPLAIN YES** options on the **BIND** or the **PREP** command, or include a static EXPLAIN statement in the source program.
- Dynamic SQL and XQuery statements

  Explain table information is captured in any of the following cases.
  - If the CURRENT EXPLAIN MODE special register is set to:
    - YES: The SQL and XQuery compiler captures explain data and executes the query statement.
    - EXPLAIN: The SQL and XQuery compiler captures explain data, but does not execute the query statement.
    - RECOMMEND INDEXES: The SQL and XQuery compiler captures explain data, and recommended indexes are placed in the ADVISE_INDEX table, but the query statement is not executed.
    - EVALUATE INDEXES: The SQL and XQuery compiler uses indexes that were placed by the user in the ADVISE_INDEX table for evaluation. In this mode, all dynamic statements are explained as though these virtual indexes were available. The query compiler then chooses to use the virtual indexes if they improve the performance of the statements. Otherwise, the indexes are ignored. To find out if proposed indexes are useful, review the EXPLAIN results.
    - REOPT: The query compiler captures explain data for static or dynamic SQL or XQuery statements during statement re-optimization at execution time, when actual values for host variables, parameter markers, global variables, or special registers are available.
- If the **EXPLAIN ALL** option has been specified on the **BIND** or **PREP** command, the query compiler captures explain data for dynamic SQL and XQuery statements at run time, even if the CURRENT EXPLAIN MODE special register is set to NO.

Capturing explain snapshot information

When an explain snapshot is requested, explain information is stored in the SNAPSHOT column of the EXPLAIN_STATEMENT table.

Explain snapshot data is captured when an SQL or XQuery statement is compiled and explain data has been requested, as follows:

- Static or incremental bind SQL and XQuery statements

An explain snapshot is captured when either the **EXPLSNAP ALL** or the **EXPLSNAP YES** clause is specified on the **BIND** or the **PREP** command, or when the source program includes a static EXPLAIN statement that uses a FOR SNAPSHOT or a WITH SNAPSHOT clause.

- Dynamic SQL and XQuery statements

  An explain snapshot is captured in any of the following cases.

  – You issue an EXPLAIN statement with a FOR SNAPSHOT or a WITH SNAPSHOT clause. With the former, only explain snapshot information is captured; with the latter, all explain information is captured.

  – If the CURRENT EXPLAIN SNAPSHOT special register is set to:

    - YES: The SQL and XQuery compiler captures explain snapshot data and executes the query statement.
    - EXPLAIN: The SQL and XQuery compiler captures explain snapshot data, but does not execute the query statement.

  – You specify the **EXPLSNAP ALL** option on the **BIND** or **PREP** command. The query compiler captures explain snapshot data at run time, even if the CURRENT EXPLAIN SNAPSHOT special register is set to NO.

Guidelines for capturing section explain information:

The section explain functionality captures (either directly or via tooling) explain information about a statement using only the contents of the runtime section. The section explain is similar to the functionality provided by the **db2expln** command, but the section explain gives a level of detail approaching that which is provided by the explain facility.

By explaining a statement using the contents of the runtime section, you can obtain information and diagnostics about what will actually be run (or was run, if the section was captured after execution), as opposed to issuing an EXPLAIN statement which might produce a different access plan (for example, in the case of dynamic SQL, the statistics might have been updated since the last execution of the statement resulting in a different access plan being chosen when the EXPLAIN statement compiles the statement being explained).

The section explain interfaces will populate the explain tables with information that is similar to what is produced by an EXPLAIN statement. However, there are some differences. After the data has been written to the explain tables, it may be processed by any of the existing explain tools you want to use (for example, the **db2exfmt** command).

Section explain interfaces

There are four interface procedures, in the following list, that can perform a section explain. The procedures differ by only the input that is provided (that is, the means by which the section is located):

EXPLAIN_FROM_ACTIVITY
: Takes application ID, activity ID, uow ID, and activity event monitor name as input. The procedure searches for the section corresponding to this activity in the activity event monitor (an SQL activity is a specific execution of a section). A section explain using this interface contains section actuals because a specific execution of the section is being performed.

EXPLAIN_FROM_CATALOG
Takes package name, package schema, unique ID, and section number as input. The procedure searches the catalog tables for the specific section.

EXPLAIN_FROM_DATA
Takes executable ID, section, and statement text as input.

EXPLAIN_FROM_SECTION
Takes executable ID and location as input, where location is specified by using one of the following:

- In-memory package cache
- Package cache event monitor name

The procedure searches for the section in the given location.

An executable ID uniquely and consistently identifies a section. The executable ID is an opaque, binary token generated at the data server for each section that has been executed. The executable ID is used as input to query monitoring data for the section, and to perform a section explain.

In each case, the procedure performs an explain, using the information contained in the identified runtime section, and writes the explain information to the explain tables identified by an *explain_schema* input parameter. It is the responsibility of the caller to perform a commit after invoking the procedure.

Investigating query performance using Explain information obtained from a section:

You can use the explain facility to examine the access plan for a specific statement *as it will actually run* (or as it was run) by generating the access plan from the section for the statement itself. By contrast, using the EXPLAIN statement creates the access plan by recompiling the statement. The resulting access plans from each of these two methods of creating access plans can be different. For example, if the statement in a section was compiled, say, 2 hours ago, then the access plan it uses might be different than the one produced by running the EXPLAIN statement against the statement.

If you have activity event monitor information available, you can generate the access plan for the section after it has run using the EXPLAIN_FROM_ACTIVITY procedure. (If section actuals are being collected, you can also view this information along with the estimates generated by the explain facility in the access plan. See “Capturing and accessing section actuals” on page 315 for more information.)

If there is no activity event monitor information available for the statement, you can use the EXPLAIN_FROM_SECTION procedure to generate the access plan for the statement as it will run based on the section stored in the package cache. This topic shows how to use EXPLAIN_FROM_SECTION to view access plan information for a statement based on section information in the package cache.

Before you begin

This task assumes that you have already created the explain tables required by the explain facility.

About this task

In this topic, assume that you want to use the explain facility to examine the most CPU-intensive statement in the package cache.

Procedure

The first part of this procedure shows how to identify the most CPU-intensive statement. Then, it shows how to use the EXPLAIN_FROM_SECTION procedure to view the access plan information for that statement as it will actually run.

1. Identify the statement that using the most processor time:

```
SELECT SECTION_TYPE,
    CASE
      WHEN SUM(NUM_COORD_EXEC_WITH_METRICS) > 0 THEN
           SUM(TOTAL_CPU_TIME)/SUM(NUM_COORD_EXEC_WITH_METRICS)
      ELSE
           0
    END as  AVG_CPU_TIME,
    EXECUTABLE_ID,
    VARCHAR(STMT_TEXT, 200) AS TEXT
    FROM TABLE(MON_GET_PKG_CACHE_STMT ( 'D', NULL, NULL, -2)) as T
    WHERE T.NUM_EXEC_WITH_METRICS <> 0 AND STMT_TYPE_ID LIKE 'DML%'
    GROUP BY SECTION_TYPE, EXECUTABLE_ID, VARCHAR(STMT_TEXT, 200)
    ORDER BY AVG_CPU_TIME DESC
```

   The preceding SQL is written to avoid division by 0 when calculating the average processor time across members. It also examines DML statements only, since the explain facility does not operate on DDL statements. The results of this query are as follows:

```
SECTION_TYPE AVG_CPU_TIME         EXECUTABLE_ID                                                                   TEXT
------------ -------------------- ------------------------------------------------------------------------------- --------------------------------------------------
D                          250000 x'0100000000000005F000000000000000000000000200201011081356293590000' select cust_last_name, cust_cc_number, cust_intere
SQL0445W  Value "SELECT POLICY FROM SYSTOOLS.POLICY WHERE MED='DB2TableMainte"
has been truncated.  SQLSTATE=01004

D                          15625 x'0100000000000001B000000000000000000000000200201011081350176250000' UPDATE SYSTOOLS.HMON_ATM_INFO SET STATS_LOCK = 'N'
D                          15625 x'0100000000000002000000000000000000000000200201011081350182960001' UPDATE SYSTOOLS.HMON_ATM_INFO AS ATM SET ATM.STATS
D                          15625 x'0100000000000002100000000000000000000000200201011081350183120001' UPDATE SYSTOOLS.HMON_ATM_INFO AS ATM SET STATS_FLA
D                           7812 x'0100000000000001500000000000000000000000200201011081350169840000' SELECT COLNAME, TYPENAME FROM  SYSCAT.COLUMNS WHER
D                              0 x'0100000000000001600000000000000000000000200201011081350171560000' SELECT TRIGNAME FROM  SYSCAT.TRIGGERS WHERE TABNAM
D                              0 x'0100000000000001900000000000000000000000200201011081350174840000' SELECT TABNAME FROM SYSCAT.TABLES  WHERE TABNAME='
D                              0 x'0100000000000001A000000000000000000000000200201011081350175000000' SELECT TABNAME FROM SYSCAT.TABLES  WHERE TABNAME='
D                              0 x'0100000000000001C000000000000000000000000200201011081350177500002' SELECT CREATOR, NAME, CTIME FROM SYSIBM.SYSTABLES
D                              0 x'0100000000000001D000000000000000000000000200201011081350178280001' SELECT CREATE_TIME FROM SYSTOOLS.HMON_ATM_INFO WHE
D                              0 x'0100000000000001E000000000000000000000000200201011081350180000001' DELETE FROM SYSTOOLS.HMON_ATM_INFO AS ATM WHERE NO
D                              0 x'0100000000000001F000000000000000000000000200201011081350180930000' SELECT POLICY FROM SYSTOOLS.POLICY WHERE MED='DB2T
D                              0 x'0100000000000002200000000000000000000000200201011081350183280001' SELECT IBM.TID, IBM.FID FROM SYSIBM.SYSTABLES AS I
D                              0 x'0100000000000002300000000000000000000000200201011081350183430001' SELECT POLICY FROM SYSTOOLS.POLICY WHERE MED='DB2C

  14 record(s) selected with 1 warning messages printed.
```

2. Based on the output of the preceding query, use the EXPLAIN_FROM_SECTION procedure to generate explain information from the section for the most CPU-intensive statement:

```
CALL EXPLAIN_FROM_SECTION (x'0100000000000005F000000000000000000000000200201011081356293590000' ,'M', NULL, 0, NULL, ?, ?, ?, ?, ? )
```

   The output of the EXPLAIN_FROM_SECTION procedure is as follows:

```
  Value of output parameters
  --------------------------
  Parameter Name  : EXPLAIN_SCHEMA
  Parameter Value : DB2DOCS

  Parameter Name  : EXPLAIN_REQUESTER
  Parameter Value : DB2DOCS

  Parameter Name  : EXPLAIN_TIME
  Parameter Value : 2010-11-08-13.57.52.984001

  Parameter Name  : SOURCE_NAME
  Parameter Value : SQLC2H21
```

```
  Parameter Name  : SOURCE_SCHEMA
  Parameter Value : NULLID

  Parameter Name  : SOURCE_VERSION
  Parameter Value :
```

3. You can now examine the explain information, either by examining the explain tables using SQL, or using the **db2exfmt** command to format the information for easier reading. For example, running **db2exfmt -d gsdb -e db2docs -w 2010-11-08-13.57.52.984001 -n SQLC2H21 -s NULLID -t -#0** against the explain information collected from the previous step generates the following output:

```
Connecting to the Database.
Db2 Universal Database Version 9.7, 5622-044 (c) Copyright IBM Corp. 1991, 2008
Licensed Material - Program Property of IBM
IBM DATABASE 2 Explain Table Format Tool



******************** EXPLAIN INSTANCE ********************

DB2_VERSION:       09.07.2
SOURCE_NAME:       SQLC2H21
SOURCE_SCHEMA:     NULLID
SOURCE_VERSION:
EXPLAIN_TIME:      2010-11-08-13.57.52.984001
EXPLAIN_REQUESTER: DB2DOCS

Database Context:
----------------
   Parallelism:          None
   CPU Speed:            8.029852e-007
   Comm Speed:           100
   Buffer Pool size:     21418
   Sort Heap size:       6590
   Database Heap size:   1196
   Lock List size:       21386
   Maximum Lock List:    97
   Average Applications: 1
   Locks Available:      663821

Package Context:
---------------
   SQL Type:           Dynamic
   Optimization Level: 5
   Blocking:           Block All Cursors
   Isolation Level:    Cursor Stability



---------------- STATEMENT 1  SECTION 201 ----------------
   QUERYNO:       0
   QUERYTAG:      CLP
   Statement Type:   Select
   Updatable:        No
   Deletable:        No
   Query Degree:     1

Original Statement:
------------------
select cust_last_name, cust_cc_number, cust_interest_code
from gosalesct.cust_crdt_card C, gosalesct.cust_customer D,
        gosalesct.cust_interest E
where C.cust_code=d.cust_code AND c.cust_code=e.cust_code
group by d.cust_last_name,    c.cust_cc_number, e.cust_interest_code
order by d.cust_last_name ASC, c.cust_cc_number DESC, e.cust_interest_code
        ASC


Optimized Statement:
-------------------
SELECT Q5.CUST_LAST_NAME AS "CUST_LAST_NAME", Q5.CUST_CC_NUMBER AS
        "CUST_CC_NUMBER", Q5.CUST_INTEREST_CODE AS "CUST_INTEREST_CODE"
FROM
   (SELECT Q4.CUST_LAST_NAME, Q4.CUST_CC_NUMBER, Q4.CUST_INTEREST_CODE
   FROM
      (SELECT Q2.CUST_LAST_NAME, Q3.CUST_CC_NUMBER, Q1.CUST_INTEREST_CODE
      FROM GOSALESCT.CUST_INTEREST AS Q1, GOSALESCT.CUST_CUSTOMER AS Q2,
```

```
            GOSALESCT.CUST_CRDT_CARD AS Q3
      WHERE (Q3.CUST_CODE = Q1.CUST_CODE) AND (Q1.CUST_CODE = Q2.CUST_CODE))
           AS Q4
   GROUP BY Q4.CUST_INTEREST_CODE, Q4.CUST_CC_NUMBER, Q4.CUST_LAST_NAME) AS
        Q5
ORDER BY Q5.CUST_LAST_NAME, Q5.CUST_CC_NUMBER DESC, Q5.CUST_INTEREST_CODE

Explain level:    Explain from section

Access Plan:
-----------
   Total Cost:       1255.29
   Query Degree:      1

                          Rows
                         RETURN
                         (   1)
                          Cost
                           I/O
                           |
                          31255
                         GRPBY
                         (   2)
                         1255.29
                           NA
                           |
                          31255
                         TBSCAN
                         (   3)
                         1249.02
                           NA
                           |
                          31255
                         SORT
                         (   4)
                         1242.74
                           NA
                           |
                          31255
                         ^HSJOIN
                         (   5)
                         1134.96
                           NA
                      /---------+---------\
                    31255                  31255
                   HSJOIN                 TBSCAN
                   (   6)                 (   9)
                   406.871                716.136
                     NA                     NA
                /------+-------\              |
              31255             31255        31255
             TBSCAN            IXSCAN   TABLE: GOSALESCT
             (   7)            (   8)     CUST_CUSTOMER
             235.505           159.488         Q2
               NA                NA
               |                 |
              31255             -1
 TABLE: GOSALESCT    INDEX: SYSIBM
  CUST_CRDT_CARD   SQL101108113609000
        Q3                 Q1


:
Objects Used in Access Plan:
---------------------------

   Schema: SYSIBM
   Name:    SQL101108113609000
   Type:    Index
         Last statistics update:    2010-11-08-13.29.58.531000
         Number of rows:       -1
         Number of buffer pool pages:    -1
         Distinct row values:       Yes
         Tablespace name:       GOSALES_TS
         Tablespace overhead:       7.500000
         Tablespace transfer rate:    0.060000
         Prefetch page count:       32
         Container extent page count:    32
         Index clustering statistic:    1.000000
         Index leaf pages:       37
         Index tree levels:       2
         Index full key cardinality:    31255
         Base Table Schema:       GOSALESCT
```

```
        Base Table Name:       CUST_INTEREST
        Columns in index:
           CUST_CODE(A)
           CUST_INTEREST_CODE(A)

   Schema: GOSALESCT
   Name:    CUST_CRDT_CARD
   Type:    Table
        Last statistics update:    2010-11-08-11.59.58.531000
        Number of rows:       31255
        Number of buffer pool pages:    192
        Distinct row values:       No
        Tablespace name:       GOSALES_TS
        Tablespace overhead:       7.500000
        Tablespace transfer rate:    0.060000
        Prefetch page count:       32
        Container extent page count:    32
        Table overflow record count:    0
        Table Active Blocks:       -1
        Average Row Compression Ratio:    0
        Percentage Rows Compressed:    0
        Average Compressed Row Size:    0

   Schema: GOSALESCT
   Name:    CUST_CUSTOMER
   Type:    Table
        Last statistics update:    2010-11-08-11.59.59.437000
        Number of rows:       31255
        Number of buffer pool pages:    672
        Distinct row values:       No
        Tablespace name:       GOSALES_TS
        Tablespace overhead:       7.500000
        Tablespace transfer rate:    0.060000
        Prefetch page count:       32
        Container extent page count:    32
        Table overflow record count:    0
        Table Active Blocks:       -1
        Average Row Compression Ratio:    0
        Percentage Rows Compressed:    0
        Average Compressed Row Size:    0

Base Table For Index Not Already Shown:
---------------------------------------

   Schema: GOSALESCT
   Name:    CUST_INTEREST
        Time of creation:       2010-11-08-11.30.28.203002
        Last statistics update:    2010-11-08-13.29.58.531000
        Number of rows:       31255
        Number of pages:       128
        Number of pages with rows:    124
        Table overflow record count:    0
        Indexspace name:       GOSALES_TS
        Tablespace name:       GOSALES_TS
        Tablespace overhead:       7.500000
        Tablespace transfer rate:    0.060000
        Prefetch page count:       -1
        Container extent page count:    32

        Long tablespace name:       GOSALES_TS
```

(The preceding output has had several lines removed for presentation purposes.)

What to do next

Analyze the explain output to see where there are opportunities to tune the query.

Differences between section explain and EXPLAIN statement output:

The results obtained after issuing a section explain are similar to those collected after running the EXPLAIN statement. There are slight differences which are described per affected explain table and by the implications, if any, to the output generated by the **db2exfmt** utility.

The stored procedure output parameters EXPLAIN_REQUESTER, EXPLAIN_TIME, SOURCE_NAME, SOURCE_SCHEMA, and SOURCE_VERSION comprise the key used to look up the information for the section in the explain tables. Use these parameters with any existing explain tools (for example, **db2exfmt**) to format the explain information retrieved from the section.

EXPLAIN_INSTANCE table

The following columns are set differently for the row generated by a section explain:

- EXPLAIN_OPTION is set to value `S`
- SNAPSHOT_TAKEN is always set to `N`
- REMARKS is always NULL

EXPLAIN_STATEMENT table

When a section explain has generated an explain output, the EXPLAIN_LEVEL column is set to value `S`. It is important to note that the EXPLAIN_LEVEL column is part of the primary key of the table and part of the foreign key of most other EXPLAIN tables; hence, this EXPLAIN_LEVEL value will also be present in those other tables.

In the EXPLAIN_STATEMENT table, the remaining column values that are usually associated with a row with EXPLAIN_LEVEL = `P`, are instead present when EXPLAIN_LEVEL = `S`, with the exception of SNAPSHOT. SNAPSHOT is always NULL when EXPLAIN_LEVEL is `S`.

If the original statement was not available at the time the section explain was generated (for example, if the statement text was not provided to the EXPLAIN_FROM_DATA procedure), STATEMENT_TEXT is set to the string `UNKNOWN` when EXPLAIN_LEVEL is set to `O`.

In the **db2exfmt** output for a section explain, the following extra line is shown after the optimized statement:

```
Explain level:    Explain from section
```

EXPLAIN_OPERATOR table

Considering all of the columns recording a cost, only the TOTAL_COST and FIRST_ROW_COST columns are populated with a value after a section explain. All the other columns recording cost have a value of `-1`.

In the **db2exfmt** output for a section explain, the following differences are obtained:

- In the access plan graph, the I/O cost is shown as `NA`
- In the details for each operator, the only costs shown are `Cumulative Total Cost` and `Cumulative First Row Cost`

EXPLAIN_PREDICATE table

No differences.

EXPLAIN_ARGUMENT table

A small number of argument types are not written to the EXPLAIN_ARGUMENT table when a section explain is issued.

EXPLAIN_STREAM table

The following columns do not have values after a section explain:
- SINGLE_NODE
- PARTITION_COLUMNS
- SEQUENCE_SIZES

The following column always has a value of -1 after a section explain:
- PREDICATE_ID

The following columns will have values only for streams originating from a base table object or default to no value and -1 respectively after a section explain:
- COLUMN_NAMES
- COLUMN_COUNT

In the **db2exfmt** output for a section explain, the information from these listed columns is omitted from the `Input Streams` and `Output Streams` section for each operator when they do not have values, or have a value of -1.

EXPLAIN_OBJECT table

After issuing a section explain, the STATS_SRC column is always set to an empty string and the CREATE_TIME column is set to NULL.

The following columns always have values of -1 after a section explain:
- COLUMN_COUNT
- WIDTH
- FIRSTKEYCARD
- FIRST2KEYCARD
- FIRST3KEYCARD
- FIRST4KEYCARD
- SEQUENTIAL_PAGES
- DENSITY
- AVERAGE_SEQUENCE_GAP
- AVERAGE_SEQUENCE_FETCH_GAP
- AVERAGE_SEQUENCE_PAGES
- AVERAGE_SEQUENCE_FETCH_PAGES
- AVERAGE_RANDOM_PAGES
- AVERAGE_RANDOM_FETCH_PAGES
- NUMRIDS
- NUMRIDS_DELETED
- NUM_EMPTY_LEAFS
- ACTIVE_BLOCKS
- NUM_DATA_PART

The following columns will also have values of -1 after a section explain for partitioned objects:
- OVERHEAD
- TRANSFER_RATE

- PREFETCHSIZE

In the **db2exfmt** output for a section explain, the information from these listed columns is omitted from the per-table and per-index statistical information found near the end of the output.

Section explain does not include compiler-referenced objects in its output (that is, rows where OBJECT_TYPE starts with a +). These objects are not shown in the **db2exfmt** output.

Capturing and accessing section actuals:

Section actuals are runtime statistics collected during the execution of the section for an access plan. To capture a section with actuals, you use the activity event monitor. To access the section actuals, you perform a section explain using the EXPLAIN_FROM_ACTIVITY stored procedure.

To be able to view section actuals, you must perform a section explain on a section for which section actuals were captured (that is, both the section and the section actuals are the inputs to the explain facility). Information about enabling, capturing, and accessing section actuals is provided here.

Enabling section actuals

Section actuals will only be updated at runtime if they have been enabled. Enable section actuals for the entire database using the **section_actuals** database configuration parameter or for a specific application using the WLM_SET_CONN_ENV procedure.

Section actuals will only be updated at runtime if they have been enabled. Enable section actuals using the **section_actuals** database configuration parameter. To enable section actuals, set the parameter to BASE (the default value is NONE). For example:

```
db2 update database configuration using section_actuals base
```

To enable section actuals for a specific application, use the WLM_SET_CONN_ENV procedure and specify BASE for the **section_actuals** element. For example:

```
CALL WLM_SET_CONN_ENV(NULL,
     '<collectactdata>WITH DETAILS, SECTION</collectactdata>
      <collectsectionactuals>BASE</collectsectionactuals>
     ')
```

Note:

1. The setting of the **section_actuals** database configuration parameter that was in effect at the start of the unit of work is applied to all statements in that unit of work. When the **section_actuals** database configuration parameter is changed dynamically, the new value will not be seen by an application until the next unit of work.
2. The **section_actuals** setting specified by the **WLM_SET_CONN_ENV** procedure for an application takes effect immediately. Section actuals will be collected for the next statement issued by the application.
3. Section actuals cannot be enabled if automatic statistics profile generation is enabled (SQLCODE -5153).

Capturing section actuals

The mechanism for capturing a section, with section actuals, is the activity event monitor. An activity event monitor writes out details of an activity when the activity completes execution, if collection of activity information is enabled. Activity information collection is enabled using the COLLECT ACTIVITY DATA clause on a workload, service class, threshold, or work action. To specify collection of a section and actuals (if the latter is enabled), the SECTION option of the COLLECT ACTIVITY DATA clause is used. For example, the following statement indicates that any SQL statement, issued by a connection associated with the `WL1` workload, will have information (including section and actuals) collected by any active activity event monitor when the statement completes:

```
ALTER WORKLOAD WL1 COLLECT ACTIVITY DATA WITH DETAILS,SECTION
```

In a partitioned database environment, section actuals are captured by an activity event monitor on all partitions where the activity was executed, if the statement being executed has a COLLECT ACTIVITY DATA clause applied to it and the COLLECT ACTIVITY DATA clause specifies both the SECTION keyword and the ON ALL DATABASE PARTITIONS clause. If the ON ALL DATABASE PARTITIONS clause is not specified, then actuals are captured on only the coordinator partition. In addition, besides the COLLECT ACTIVITY DATA clause on a workload, service class, threshold, or work action, activity collection can be enabled (for an individual application) using the WLM_SET_CONN_ENV procedure with a second argument that includes the collectactdata tag with a value of "WITH DETAILS, SECTION".

Limitations

The limitations, with respect to the capture of section actuals, are the following:

- Section actuals will not be captured when the WLM_CAPTURE_ACTIVITY_IN_PROGRESS stored procedure is used to send information about a currently executing activity to an activity event monitor. Any activity event monitor record generated by the WLM_CAPTURE_ACTIVITY_IN_PROGRESS stored procedure will have a value of 1 in its partial_record column.
- When a reactive threshold has been violated, section actuals will be captured on only the coordinator partition.
- Explain tables must be migrated to Db2 Version 9.7 Fix Pack 1, or later, before section actuals can be accessed using a section explain. If the explain tables have not been migrated, the section explain will work, but section actuals information will not be populated in the explain tables. In this case, an entry will be written to the EXPLAIN_DIAGNOSTIC table.
- Existing Db2 V9.7 activity event monitor tables (in particular, the activity table) must be recreated before section actuals data can be captured by the activity event monitor. If the activity logical group does not contain the SECTION_ACTUALS column, a section explain may still be performed using a section captured by the activity event monitor, but the explain will not contain any section actuals data.

Accessing section actuals

Section actuals can be accessed using the EXPLAIN_FROM_ACTIVITY procedure. When you perform a section explain on an activity for which section actuals were captured, the EXPLAIN_ACTUALS explain table will be populated with the actuals information.

Note: Section actuals are only available when a section explain is performed using the EXPLAIN_FROM_ACTIVITY procedure.

The EXPLAIN_ACTUALS table is the child table of the existing EXPLAIN_OPERATOR explain table. When EXPLAIN_FROM_ACTIVITY is invoked, if the section actuals are available, the EXPLAIN_ACTUALS table will be populated with the actuals data. If the section actuals are collected on multiple database partitions, there is one row per database partition for each operator in the EXPLAIN_ACTUALS table.

Obtaining a section explain with actuals to investigate poor query performance:

To resolve a SQL query performance slow down, you can begin by obtaining a section explain that includes section actuals information. The section actuals values can then be compared with the estimated access plan values generated by the optimizer to assess the validity of the access plan. This task takes you through the process of obtaining section actuals to investigate poor query performance.

Before you begin

You have completed the diagnosis phase of your investigation and determined that indeed you have a SQL query performance slow down and you have determined which statement is suspected to be involved in the performance slow down.

About this task

This task takes you through the process of obtaining section actuals to investigate poor query performance. The information contained in the sections actuals, when compared with the estimated values generated by the optimizer, can help to resolve the query performance slow down.

Restrictions

See the limitations in "Capturing and accessing section actuals".

Procedure

To investigate poor query performance for a query executed by the `myApp.exe` application, complete the following steps:

1. Enable section actuals:
   ```
   DB2 UPDATE DATABASE CONFIGURATION USING SECTION_ACTUALS BASE
   ```
2. Create the EXPLAIN tables in the MYSCHEMA schema using the SYSINSTALLOBJECTS procedure:
   ```
   CALL SYSINSTALLOBJECTS( 'EXPLAIN', 'C', NULL, 'MYSCHEMA' )
   ```

   Note: This step can be skipped if you have already created the EXPLAIN tables.

3. Create a workload MYCOLLECTWL to collect activities submitted by the myApp.exe application and enable collection of section data for those activities by issuing the following two commands:

   ```
   CREATE WORKLOAD MYCOLLECTWL APPLNAME( 'MYAPP.EXE')
   COLLECT ACTIVITY DATA WITH DETAILS,SECTION
   ```

   Followed by:

   ```
   GRANT USAGE ON WORKLOAD MYCOLLECTWL TO PUBLIC
   ```

   Note: Choosing to use a separate workload limits the amount of information captured by the activity event monitor

4. Create an activity event monitor, called ACTEVMON, by issuing the following statement:

   ```
   CREATE EVENT MONITOR ACTEVMON FOR ACTIVITIES WRITE TO TABLE
   ```

5. Activate the activity event monitor ACTEVMON by executing the following statement:

   ```
   SET EVENT MONITOR ACTEVMON STATE 1
   ```

6. Run the myApp.exe application. All statements, issued by the application, are captured by the activity event monitor.
7. Query the activity event monitor tables to find the identifier information for the statement of interest by issuing the following statement:

   ```
   SELECT APPL_ID,
          UOW_ID,
          ACTIVITY_ID,
          STMT_TEXT
   FROM ACTIVITYSTMT_ACTEVMON
   ```

   The following is an example of the output that was generated as a result of the issued select statement:

   ```
   APPL_ID                    UOW_ID    ACTIVITY_ID     STMT_TEXT
   -------------------------  --------  --------------  --------------
   *N2.DB2INST1.0B5A12222841         1               1  SELECT * FROM ...
   ```

8. Use the activity identifier information as input to the EXPLAIN_FROM_ACTIVITY procedure to obtain a section explain with actuals, as shown in the following call statement:

   ```
   CALL EXPLAIN_FROM_ACTIVITY( '*N2.DB2INST1.0B5A12222841', 1, 1, 'ACTEVMON',
    'MYSCHEMA', ?, ?, ?, ?, ? )
   ```

   The following is a sample output resulting from the EXPLAIN_FROM_ACTIVITY call:

   ```
   Value of output parameters
   --------------------------
   Parameter Name : EXPLAIN_SCHEMA
   Parameter Value : MYSCHEMA

   Parameter Name : EXPLAIN_REQUESTER
   Parameter Value : SWALKTY

   Parameter Name : EXPLAIN_TIME
   Parameter Value : 2009-08-24-12.33.57.525703

   Parameter Name : SOURCE_NAME
   Parameter Value : SQLC2H20

   Parameter Name : SOURCE_SCHEMA
   Parameter Value : NULLID
   ```

```
Parameter Name : SOURCE_VERSION
Parameter Value :

Return Status = 0
```

9. Format the explain data using the **db2exfmt** command and specifying, as input, the explain instance key that was returned as output from the EXPLAIN_FROM_ACTIVITY procedure, such as the following:

```
db2exfmt -d test -w 2009-08-24-12.33.57.525703 -n SQLC2H20 -s NULLID -# 0 -t
```

The explain instance output was the following:

```
  ******************** EXPLAIN INSTANCE ********************

DB2_VERSION:       09.07.1
SOURCE_NAME:       SQLC2H20
SOURCE_SCHEMA:     NULLID
SOURCE_VERSION:
EXPLAIN_TIME:      2009-08-24-12.33.57.525703
EXPLAIN_REQUESTER: SWALKTY

Database Context:
----------------
        Parallelism:          None
        CPU Speed:            4.000000e-05
        Comm Speed:           0
        Buffer Pool size:     198224
        Sort Heap size:       1278
        Database Heap size:   2512
        Lock List size:       6200
        Maximum Lock List:    60
        Average Applications: 1
        Locks Available:      119040

Package Context:
---------------
        SQL Type:           Dynamic
        Optimization Level: 5
        Blocking:           Block All Cursors
        Isolation Level:    Cursor Stability



---------------- STATEMENT 1  SECTION 201 ----------------
        QUERYNO:       0
        QUERYTAG:      CLP
        Statement Type:   Select
        Updatable:        No
        Deletable:        No
        Query Degree:     1

Original Statement:
------------------
select *
from syscat.tables

Optimized Statement:
-------------------
SELECT Q10.$C67 AS "TABSCHEMA", Q10.$C66 AS "TABNAME", Q10.$C65 AS "OWNER",
        Q10.$C64 AS "OWNERTYPE", Q10.$C63 AS "TYPE", Q10.$C62 AS "STATUS",
        Q10.$C61 AS "BASE_TABSCHEMA", Q10.$C60 AS "BASE_TABNAME", Q10.$C59 AS
        "ROWTYPESCHEMA", Q10.$C58 AS "ROWTYPENAME", Q10.$C57 AS "CREATE_TIME",
        Q10.$C56 AS "ALTER_TIME", Q10.$C55 AS "INVALIDATE_TIME", Q10.$C54 AS
        "STATS_TIME", Q10.$C53 AS "COLCOUNT", Q10.$C52 AS "TABLEID", Q10.$C51
        AS "TBSPACEID", Q10.$C50 AS "CARD", Q10.$C49 AS "NPAGES", Q10.$C48 AS
        "FPAGES", Q10.$C47 AS "OVERFLOW", Q10.$C46 AS "TBSPACE", Q10.$C45 AS
        "INDEX_TBSPACE", Q10.$C44 AS "LONG_TBSPACE", Q10.$C43 AS "PARENTS",
        Q10.$C42 AS "CHILDREN", Q10.$C41 AS "SELFREFS", Q10.$C40 AS
```

```
        "KEYCOLUMNS", Q10.$C39 AS "KEYINDEXID", Q10.$C38 AS "KEYUNIQUE",
        Q10.$C37 AS "CHECKCOUNT", Q10.$C36 AS "DATACAPTURE", Q10.$C35 AS
        "CONST_CHECKED", Q10.$C34 AS "PMAP_ID", Q10.$C33 AS "PARTITION_MODE",
        '0' AS "LOG_ATTRIBUTE", Q10.$C32 AS "PCTFREE", Q10.$C31 AS
        "APPEND_MODE", Q10.$C30 AS "REFRESH", Q10.$C29 AS "REFRESH_TIME",
...

Explain level:    Explain from section

Access Plan:
-----------
        Total Cost:             154.035
        Query Degree:           1

                                                  Rows
                                              Rows Actual
                                                RETURN
                                                (   1)
                                                  Cost
                                                  I/O
                                                  |
                                                  54
                                                  396
                                               >^HSJOIN
                                                (   2)
                                                 153.056
                                                   NA
                                          /----------+----------\
                                        54                      20
                                        396                      0
                                     >^HSJOIN                 TBSCAN
                                     (   3)                   (  12)
                                      140.872                 11.0302
                                        NA                      NA
                                   (continued below)              |
                                                                 20
                                                                 NA
                                                             TABLE: SYSIBM
                                                            SYSAUDITPOLICIES


                                              (continued from above)
                                          /---------+---------\
                                        54                      6
                                        396                     0
                                     >^HSJOIN                IXSCAN
                                     (   4)                  (  11)
                                      138.033                2.01136
                                        NA                     NA
                                  /---------+---------\          |
                                54                      6        -1
                                396                     0        NA
                             >^HSJOIN                IXSCAN    INDEX: SYSIBM
                             (   5)                  (  10)   INDCOLLATIONS04
                              135.193                2.01136
                                NA                     NA
                          /-------+--------\             |
                        54                    6          -1
                        396                   0          NA
                     >^HSJOIN              IXSCAN    INDEX: SYSIBM
                     (   6)                (   9)   INDCOLLATIONS04
                      132.354              2.01136
                        NA                   NA
                   /-----+-----\             |
                 54              6          -1
                 396             0          NA
```

```
      TBSCAN          IXSCAN    INDEX: SYSIBM
      (   7)          (   8)   INDCOLLATIONS04
      129.57          2.01136
        NA              NA
        |               |
        54              -1
        NA              NA
   TABLE: SYSIBM   INDEX: SYSIBM
     SYSTABLES    INDCOLLATIONS04
```

...

10. Examine the section actuals information in the explain output. Compare the section actuals values with the estimated values of the access plan generated by the optimizer. If a discrepancy occurs between the section actuals and estimated values for the access plan, ascertain what is causing the discrepancy and take the appropriate action. As an example for the purposes of discussion, you ascertain that the table statistics are out of date for one of the tables being queried. This leads the optimizer to select an incorrect access plan which might account for the query performance slow down. The course of action to take, in this case, is to run the **RUNSTATS** command on the table to update the table statistics.
11. Retry the application to determine if the query slow down persists.

Analysis of section actuals information in explain output:

Section actuals, when available, are displayed in different parts of the explain output. Where to find section actuals information, operator details, and object statistics in explain output is described here.

Section actuals in db2exfmt command graph output

If explain actuals are available, they are displayed in the graph under the estimated rows. Graph output includes actuals only for operators, not for objects. NA (not applicable) is displayed for objects in the graph.

An example of graph output from the**db2exfmt** command is as follows:

```
      Rows
    Rows Actual
     RETURN
     (   1)
      Cost
       I/O
       |
     3.21948  << The estimated rows that are used by the optimizer
       301    << The actuals rows that are collected in run time
      DTQ
     (   2)
     75.3961
       NA
       |
     3.21948
       130
     HSJOIN
     (   3)
     72.5927
       NA
     /--+---\
   674        260
   220        130
 TBSCAN     TBSCAN
```

```
(   4)     (   5)
40.7052    26.447
  NA         NA
  |          |
  337        130
  NA         NA  << Graph output does not include actuals for objects
TABLE: FF  TABLE: FF
  T1         T2
```

In a partitioned database environment, the cardinality that is displayed in the graph is the average cardinality for the database partitions where the actuals are collected. The average is displayed because that is the value that is estimated by the optimizer. The actual average is a meaningful value to compare against the estimated average. In addition, a breakdown of section actuals per database partition is provided in the operator details output. You can examine these details to determine other information, such as total (across all partitions), minimum, and maximum.

Operator details in db2exfmt command output

The actual cardinality for an operator is displayed in the stream section following the line containing `Estimated number of rows` (`Actual number of rows` in the explain output). In a partitioned database environment, if the operator is running on more than one database member, the actual cardinality that is displayed is the average cardinality for the environment. The values per database partition are displayed under a separate section, `Explain Actuals`. This section is shown only for a partitioned database environment, but not in the serial mode. If the actuals are not available for a particular database partition, `NA` is displayed in the list of values per database partition next to the partition number. `Actual number of rows` in the section `Output Streams` is also shown as `NA`.

An example of operator details output from the **db2exfmt** command is as follows:

```
9) UNION : (Union)
   Cumulative Total Cost:     10.6858
   Cumulative First Row Cost:    9.6526

   Arguments:
   ---------
   UNIONALL: (UnionAll Parameterized Base Table)
      DISJOINT

   Input Streams:
   -------------
      5) From Operator #10

         Estimated number of rows:  30
         Actual number of rows:     63
         Partition Map ID:          3

      7) From Operator #11

         Estimated number of rows:  16
         Actual number of rows:     99
         Partition Map ID:          3


   Output Streams:
   --------------
      8) To Operator #8

         Estimated number of rows:  30
         Actual number of rows:     162
```

```
      Partition Map ID:          3

Explain Actuals:      << This section is shown only show
                              in a partitioned database environment
----------------
   DB Partition number   Cardinality
   -------------------   -----------
            1               193
            2               131
```

Object statistics in db2exfmt command output

The explain output includes statistics for each object that is used in the access plan. For partitioned tables and indexes, the statistics are per data partition. In a partitioned database environment or Db2 pureScale environment, the statistics are per member. If the statistics are not available for a particular member, NA is displayed in the values list for that member next to the member number.

The following example shows how object statistics are displayed in the output of the **db2exfmt** command:

```
Runtime statistics for objects Used in Access Plan:
---------------------------------------------------

        Schema: GOSALES
        Name:   ORDER_DETAILS
        Type:   Table

                      Member 0
                      ---------

                             Metrics
                             -----------------
                             lock_wait_time:85899
                             lock_wait_time_global:25769
                             lock_waits_local:21474
                             lock_waits_global:85899
                             lock_escals_local:17179
                             lock_escals_global:2
                             direct_writes:12884
                             direct_read_reqs:1
                             pool_data_gbp_invalid_pages:446
                             pool_data_lbp_pages_found:445
                             pool_xda_l_reads:446
                             pool_xda_p_reads:15
```

Guidelines for using explain information:

You can use explain information to understand why application performance has changed or to evaluate performance tuning efforts.

Analysis of performance changes

To help you understand the reasons for changes in query performance, you need "before and after" explain information, which you can obtain by performing the following steps:

1. Capture explain information for the query before you make any changes and save the resulting explain tables. Alternatively, save output from the **db2exfmt** explain tool.
2. Save or print the current catalog statistics. You could use the **db2look** productivity tool to help you perform this task.

3. Save or print the data definition language (DDL) statements, including CREATE TABLE, CREATE VIEW, CREATE INDEX, or CREATE TABLESPACE.

The information that you collect in this way provides a reference point for future analysis. For dynamic SQL or XQuery statements, you can collect this information when you run your application for the first time. For static SQL and XQuery statements, you can collect this information at bind time. To analyze a performance change, compare the information that you collect with this reference information that was collected previously.

For example, your analysis might show that an index is no longer being used when determining an access path. Using the catalog statistics information, you might notice that the number of index levels (the NLEVELS column) is now substantially higher than when the query was first bound to the database. You might then choose to perform one of the following actions:

- Reorganize the index
- Collect new statistics for your table and indexes
- Collect explain information when rebinding your query

After you perform one of these actions, examine the access plan again. If the index is being used, query performance might no longer be a problem. If the index is still not being used, or if performance is still a problem, choose another action from this list and examine the results. Repeat these steps until the problem is resolved.

Evaluation of performance tuning efforts

You can take a number of actions to help improve query performance, such as updating configuration parameters, adding containers, collecting fresh catalog statistics, and so on.

After you make a change in any of these areas, use the explain facility to determine what affect, if any, the change has had on the chosen access plan. For example, if you add an index or materialized query table (MQT) based on the index guidelines, the explain data can help you to determine if the index or MQT is actually being used as expected.

Although the explain output enables you to determine the access plan that was chosen and its relative cost, the only way to accurately measure the performance improvement for a specific query is to use benchmark testing techniques.

Guidelines for analyzing explain information:

The primary use for explain information is the analysis of access paths for query statements. There are a number of ways in which analyzing the explain data can help you to tune your queries and environment.

Consider the following kinds of analysis:

- Index use

  The proper indexes can significantly benefit performance. Using explain output, you can determine whether the indexes that you have created to help a specific set of queries are being used. Look for index usage in the following areas:

  – Join predicates
  – Local predicates
  – GROUP BY clause

- ORDER BY clause
- WHERE XMLEXISTS clause
- The select list

You can also use the explain facility to evaluate whether a different index or no index at all might be better. After you create a new index, use the **RUNSTATS** command to collect statistics for that index, and then recompile your query. Over time, you might notice (through explain data) that a table scan is being used instead of an index scan. This can result from a change in the clustering of the table data. If the index that was previously being used now has a low cluster ratio, you might want to:

- Reorganize the table to cluster its data according to that index
- Use the **RUNSTATS** command to collect statistics for both index and table
- Recompile the query

To determine whether reorganizing the table has improved the access plan, examine explain output for the recompiled query.

- Access type

  Analyze the explain output, and look for data access types that are not usually optimal for the type of application that you are running. For example:

  - Online transaction processing (OLTP) queries

    OLTP applications are prime candidates for index scans with range-delimiting predicates, because they tend to return only a few rows that are qualified by an equality predicate against a key column. If your OLTP queries are using a table scan, you might want to analyze the explain data to determine why an index scan is not being used.

  - Browse-only queries

    The search criteria for a "browse" type query can be very vague, resulting in a large number of qualifying rows. If users usually look at only a few screens of output data, you might specify that the entire answer set need not be computed before some results are returned. In this case, the goals of the user are different than the basic operating principle of the optimizer, which attempts to minimize resource consumption for the entire query, not just the first few screens of data.

    For example, if the explain output shows that both merge scan join and sort operators were used in the access plan, the entire answer set will be materialized in a temporary table before any rows are returned to the application. In this case, you can attempt to change the access plan by using the OPTIMIZE FOR clause on the SELECT statement. If you specify this option, the optimizer can attempt to choose an access plan that does not produce the entire answer set in a temporary table before returning the first rows to the application.

- Join methods

  If a query joins two tables, check the type of join being used. Joins that involve more rows, such as those in decision-support queries, usually run faster with a hash join or a merge join. Joins that involve only a few rows, such as those in OLTP queries, typically run faster with nested-loop joins. However, there might be extenuating circumstances in either case-such as the use of local predicates or indexes-that could change how these typical joins work.

Using access plans to self-diagnose performance problems with REFRESH TABLE and SET INTEGRITY statements:

Invoking the explain utility against REFRESH TABLE or SET INTEGRITY statements enables you to generate access plans that can be used to self-diagnose performance problems with these statements. This can help you to better maintain your materialized query tables (MQTs).

To get the access plan for a REFRESH TABLE or a SET INTEGRITY statement, use either of the following methods:

- Use the EXPLAIN PLAN FOR REFRESH TABLE or EXPLAIN PLAN FOR SET INTEGRITY option on the EXPLAIN statement.
- Set the CURRENT EXPLAIN MODE special register to EXPLAIN before issuing the REFRESH TABLE or SET INTEGRITY statement, and then set the CURRENT EXPLAIN MODE special register to NO afterwards.

Restrictions

- The REFRESH TABLE and SET INTEGRITY statements do not qualify for re-optimization; therefore, the REOPT explain mode (or explain snapshot) is not applicable to these two statements.
- The WITH REOPT ONCE clause of the EXPLAIN statement, which indicates that the specified explainable statement is to be re-optimized, is not applicable to the REFRESH TABLE and SET INTEGRITY statements.

Scenario

This scenario shows how you can generate and use access plans from EXPLAIN and REFRESH TABLE statements to self-diagnose the cause of your performance problems.

1. Create and populate your tables. For example:

```
create table t (
  i1 int not null,
  i2 int not null,
  primary key (i1)
);

insert into t values (1,1), (2,1), (3,2), (4,2);

create table mqt as (
  select i2, count(*) as cnt from t group by i2
)
data initially deferred
refresh deferred;
```

2. Issue the EXPLAIN and REFRESH TABLE statements, as follows:

```
explain plan for refresh table mqt;
```

   This step can be replaced by setting the EXPLAIN mode on the SET CURRENT EXPLAIN MODE special register, as follows:

```
set current explain mode explain;
refresh table mqt;
set current explain mode no;
```

3. Use the **db2exfmt** command to format the contents of the explain tables and obtain the access plan. This tool is located in the `misc` subdirectory of the instance `sqllib` directory.

```
db2exfmt -d dbname -o refresh.exp -1
```

4. Analyze the access plan to determine the cause of the performance problem. In the previous example, if T is a large table, a table scan would be very expensive. Creating an index might improve the performance of the query.

Tools for collecting and analyzing explain information:

The Db2 database server has a comprehensive explain facility that provides detailed information about the access plan that the optimizer chooses for an SQL or XQuery statement.

The tables that store explain data are accessible on all supported platforms and contain information for both static and dynamic SQL and XQuery statements. Several tools are available to give you the flexibility that you need to capture, display, and analyze explain information.

Detailed query optimizer information that enables the in-depth analysis of an access plan is stored in explain tables that are separate from the actual access plan itself. Use one or more of the following methods to get information from the explain tables:

- Use the **db2exfmt** tool to display explain information in formatted output.
- Write your own queries against the explain tables. Writing your own queries enables the easy manipulation of output, comparisons among different queries, or comparisons among executions of the same query over time.

Use the **db2expln** tool to see the access plan information that is available for one or more packages of static SQL or XQuery statements. This utility shows the actual implementation of the chosen access plan; it does not show optimizer information. By examining the generated access plan, the **db2expln** tool provides a relatively compact, verbal overview of the operations that will occur at run time.

The command line explain tools can be found in the `misc` subdirectory of the `sqllib` directory.

The following table summarizes the different tools that are available with the Db2 explain facility. Use this table to select the tool that is most suitable for your environment and needs.

Table 60. Explain Facility Tools

Desired characteristics	Explain tables	db2expln	db2exfmt
Text output		Yes	Yes
"Quick and dirty" static SQL and XQuery analysis		Yes	
Static SQL and XQuery support	Yes	Yes	Yes
Dynamic SQL and XQuery support	Yes	Yes	Yes
CLI application support	Yes		Yes
Available to DRDA Application Requesters	Yes		
Detailed optimizer information	Yes		Yes
Suited for analysis of multiple statements	Yes	Yes	Yes
Information is accessible from within an application	Yes		

In addition to these tools, you can use IBM Data Studio Version 3.1 or later to generate a diagram of the current access plan for SQL or XPATH statements. For more details, see Diagramming access plans with Visual Explain.

Displaying catalog statistics that are in effect at explain time

The explain facility captures the statistics that are in effect when a statement is explained. These statistics might be different than those that are stored in the system catalog, especially if real-time statistics gathering is enabled. If the explain tables are populated, but an explain snapshot was not created, only some statistics are recorded in the EXPLAIN_OBJECT table.

To capture all catalog statistics that are relevant to the statement being explained, create an explain snapshot at the same time that explain tables are being populated, then use the SYSPROC.EXPLAIN_FORMAT_STATS scalar function to format the catalog statistics in the snapshot.

If the **db2exfmt** tool is used to format the explain information, and an explain snapshot was collected, the tool automatically uses the SYSPROC.EXPLAIN_FORMAT_STATS function to display the catalog statistics.

Explain information for column-organized tables:

Explain information is captured to support column-organized tables. You can use this information to determine how your application performs when it uses this functionality.

The CTQ plan operator represents the transition between column-organized data processing and row-organized data processing.

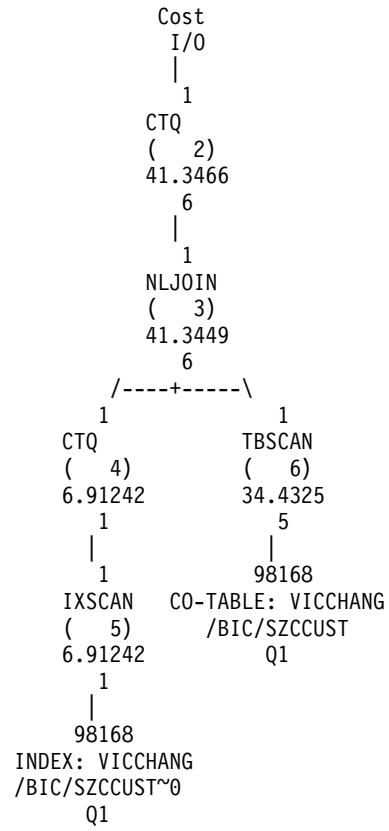
The steps that you use to capture the explain information for column-organized tables are the same steps that you use for running queries against row-organized tables.

- Set the EXPLAIN mode on by using the CURRENT EXPLAIN MODE special register as follows:
  ```
  db2 SET CURRENT EXPLAIN MODE YES
  ```
- Issue your query against column-organized tables.
- Issue the **db2exfmt** command to format the contents of the explain tables and obtain the access plan. The following example shows you how to use this command against the SAMPLE database:
  ```
  db2exfmt -d sample -1 -o output.exfmt
  ```

Improving the performance of queries on column-organized tables by using indexes

Some queries that have selective search conditions might run faster when accessing column-organized tables if indexes are used. Unique indexes are implicitly created to support enforced primary and unique key constraints. Starting in Db2 Version 11.1 Mod Pack 3 and Fix Pack 3 (11.1.3.3), indexes can also be explicitly created by using the CREATE INDEX statement. Before Db2 11.1.3.3, unique indexes were used for select, update, or delete operations that affect only one row in a column-organized table. An example is shown in the following index access plan:

```
          Rows
         RETURN
         (   1)
```



This plan is equivalent to a FETCH-IXSCAN combination that is used to access row-organized data. For index access to column-organized data, row-organized data processing retrieves the rowid from the index by using IXSCAN(5) and passes it to column-organized data processing using CTQ(4). CTQ(4) represents a column-organized table queue that passes data from row-organized data processing to column-organized data processing. TBSCAN(6) locates the columns that are identified by the rowid. TBSCAN(6) might apply additional predicates if necessary, or reapply the IXSCAN predicates in some situations. Specifically, if the table is being accessed under the UR isolation level, or the access is in support of an update or delete operation, the TBSCAN needs to apply only those predicates that were not already applied by the IXSCAN. Otherwise, the TBSCAN needs to reapply all of the IXSCAN predicates. NLJOIN(3) represents the process of retrieving the rowid from row-organized data processing and passing it to the column-organized TBSCAN.

Starting in Db2 11.1.3.3, the FETCH operator for column-organized index scans replaces the use of the previous nested-loop join method for isolation level CS, when a modification state index exists for the table. The nested-loop join method is still used for isolation level UR, if the following conditions are met:

- The result of the index access is joined to another column-organized table
- The index scan returns at most one row

Otherwise, the FETCH operator is used. A column-organized FETCH can process any number of rows, while the nested-loop join representation of a fetch operation is limited to processing no more than one row. A column-organized FETCH can apply sargable and residual predicates just like a row-organized FETCH. A column-organized FETCH can also be used in all of the same contexts as a row-organized FETCH, including the inner loop of a nested-loop join and in correlated subselect queries.

The FETCH operator runs by using row-organized processing even though it is accessing column-organized data. Data that is returned cannot be used for subsequent column-organized processing. The Db2 query optimizer also considers accessing the column-organized table by using a table scan (TBSCAN operator), if that is a less expensive option for processing such as:

- Joins
- Aggregation
- Removal of duplicate rows
- Sorting by using column-organized processing

The explain representation for a column-organized FETCH is similar to that of a row-organized FETCH, except for arguments that are not applicable to column-organized processing. The example that is shown previously for a query that uses an index before Db2 11.1.3.3 would appear as the following when a FETCH operator is used:

```
            Rows
           RETURN
           (   1)
            Cost
             I/O
             |
              1
           FETCH
           (   2)
           17.0787
              1
         /----+----\
        1           98168
    IXSCAN  CO-TABLE: VICCHANG
    (   3)     /BIC/SZCCUST
    6.91242          Q1
       1
       |
     98168
INDEX: VICCHANG
/BIC/SZCCUST~0
      Q1
```

Indexes on column-organized tables are not supported for the following index operations:

- Jump scans
- Deferred fetch index plans (index ANDing, ORing, and list prefetch)
- Star join and zigzag join
- Scan Sharing

Intra-partition parallel index scans are not supported for column-organized tables.

Update and delete operations that use an index scan on a column-organized table are not supported by the FETCH operator. Update and delete operations that affect only a single row are supported by using either index-only access or the nested-loop fetch approach.

Common table expression

A *common table expression* defines the result of a table that you can specify in the FROM clause of an SQL Statement. Statements with common table expressions against column-organized tables can have more efficient execution plans.

Consider the following query against a column-organized table T1 that has two columns, C1 and C2:

```
WITH cse(c1,c2) AS (SELECT  t1.c1, MAX(c2) FROM t1 GROUP BY t1.c1)
  SELECT a.c2 FROM cse a, cse b
    WHERE a.c1 = b.c2;
```

The following sample execution plan correspond to that query.

```
           Rows
          RETURN
          (   1)
           Cost
           I/O
            |
            25
           LTQ
          (   2)
          2078.07
            30
            |
            25
           CTQ
          (   3)
           2074
            30
            |
            25
         ^HSJOIN
          (   4)
          2073.79
             30
           /-+--\
         25       25
       TBSCAN   TBSCAN
       (   5)   (   9)
       1036.85  1036.85
         15       15
         |        |
         25       25
        TEMP     TEMP
       (   6)   (   6)
       1033.24  1033.24
         15       15
         |
         25
       GRPBY
       (   7)
       1032.45
         15
         |
       50000
       TBSCAN
       (   8)
       792.149
         15
         |
       50000
 CO-TABLE: URSU
         T1
         Q1
```

The execution plan includes the TEMP(6) operator, which materializes the results of the common table expression during column-organized data processing. Operators TBSCAN(5) and TBSCAN(9) scan the output of the TEMP(6) operator and send the data to the HSJN(4) operator. Afterward, the CTQ(3) operator sends

the results of the join operation from column-organized data processing to row-organized data processing.

Column-organized sorts

The query optimizer determines where SORT operators are placed in the access plan based on query semantics and costing. Column-organized sorting will be done to satisfy ORDER BY requirements on sub-selects and within OLAP specifications. They will also be used to partition data for OLAP specifications that include the PARTITION BY clause, to allow the OLAP function to be computed in parallel using multiple database agents. For example:

```
SELECT * FROM tc1 ORDER BY c1,c2
SELECT rank() OVER (PARTITION BY c1 ORDER BY c2) AS rnk, c2 FROM tc1
```

A column-organized sort is typically executed in parallel using multiple database agents and can use different methods to distribute the data among the agents, depending on the semantics of the SQL statement. The type of parallel sorting method is indicated by the SORTTYPE argument of the SORT operator along with the sort key columns and the sort partitioning columns. The SORTTYPE argument can have the values GLOBAL, PARTITIONED, or MERGE for a column-organized SORT.

A global sort is used when the SQL statement semantics require that the data be globally ordered to satisfy an ORDER BY request, for example. The sorting will be performed by multiple database agents but the final result will be produced by a single agent. For example, the following query has an ORDER BY on columns C1 and C2.

```
SELECT * FROM tc1 ORDER BY c1, c2
```

The access plan for this query has one SORT operator:

```
 Rows
RETURN
(   1)
 Cost
 I/O
  |
 1000
 LMTQ
(   2)
351.462
  10
  |
 1000
 CTQ
(   3)
289.471
  10
  |
 1000
TBSCAN
(   4)
288.271
  10
  |
 1000
SORT
(   5)
278.209
  10
  |
 1000
```

```
      TBSCAN
      (   6)
      85.5805
        10
        |
       1000
CO-TABLE: DB2USER
        TC1
        Q1
```

The SORT operator arguments have SORTTYPE GLOBAL, indicating that it will produce a single stream of sorted data. (Only arguments relevant to this discussion are shown):

```
 Arguments:
 ---------
       SORTKEY : (Sort Key column)
           1: Q1.C1(A)
           2: Q1.C2(A)
       SORTTYPE: (Intra-Partition parallelism sort type)
           GLOBAL
```

A partitioned sort is used when the sorted data can be used by multiple SQL operations, such as an ORDER BY request and OLAP functions that require partitioned or ordered data. For example, the following query contains 2 OLAP functions that require the data to be partitioned by (C2) and (C2,C3) and the query also has an ORDER BY clause.

```
 SELECT
     c1,
     c2,
     c3,
     MAX(c1) OVER (PARTITION BY c2),
     MAX(c1) OVER (PARTITION BY c2, c3)
 FROM
     tc1
 ORDER BY
     c2
```

The following access plan containing 2 SORT operators is chosen:

```
       Rows
      RETURN
      (   1)
       Cost
        I/O
        |
       1000
      LMTQ
      (   2)
      466.38
        10
        |
       1000
      CTQ
      (   3)
      411.188
        10
        |
       1000
      TBSCAN
      (   4)
      409.588
        10
        |
       1000
      SORT
```

```
      (   5)
      399.527
        10
        |
       1000
      TBSCAN
      (   6)
      288.271
        10
        |
       1000
      SORT
      (   7)
      278.209
        10
        |
       1000
      TBSCAN
      (   8)
      85.5805
        10
        |
       1000
CO-TABLE: DB2USER
        TC1
        Q1
```

SORT(7) is executed first and it has the following explain arguments:

```
       Arguments:
       ---------
       PARTCOLS: (Table partitioning columns)
           1: Q2.C2
       SORTKEY : (Sort Key column)
           1: Q2.C2(A)
           2: Q2.C3(A)
       SORTTYPE: (Intra-Partition parallelism sort type)
           PARTITIONED
```

SORT(7) is partitioned by ranges of values of C2. Within each sort partition, the data is sorted by columns C2 and C3 This allows both MAX functions to be computed in parallel by multiple agents. The parallel streams with the MAX results are able to maintain order on C2, allowing the ORDER BY C2 to be satisfied by simply merging the parallel streams. SORT(5) merges the streams from each agent to produce one ordered stream, rather than performing a global sort of the input streams. This is indicated by SORTTYPE MERGE in the explain arguments for SORT(5):

```
       Arguments:
       ---------
       SORTKEY : (Sort Key column)
           1: Q3.C2(A)
       SORTTYPE: (Intra-Partition parallelism sort type)
           MERGE
```

If this same query didn't have an ORDER BY, a different type of parallel sort method is used.

```
 SELECT
     c1,
     c2,
     c3,
     MAX(c1) OVER (PARTITION BY c2),
     MAX(c1) OVER (PARTITION BY c2, c3)
 FROM
     tc1
```

The following access plan with a single SORT is chosen:

```
        Rows
       RETURN
       (   1)
        Cost
         I/O
         |
        1000
       LTQ
       (   2)
       421.526
         10
         |
        1000
       CTQ
       (   3)
       383.134
         10
         |
        1000
       TBSCAN
       (   4)
       288.271
         10
         |
        1000
       SORT
       (   5)
       278.209
         10
         |
        1000
       TBSCAN
       (   6)
       85.5805
         10
         |
        1000
CO-TABLE: DB2USER
        TC1
        Q1
```

The explain arguments for SORT(5) indicate that it is a partitioned sort, however the partitioning is done differently than the previous example:

```
        Arguments:
        ---------
        PARTCOLS: (Table partitioning columns)
            1: Q2.C2
        SORTKEY : (Sort Key column)
            1: Q2.C2(R)
            2: Q2.C3(A)
        SORTTYPE: (Intra-Partition parallelism sort type)
            PARTITIONED
```

Each sort output stream read by each agent contains a range of values for C2, but the data is not ordered on C2 within each stream. Instead, the data is ordered on C3 within each distinct value of C2. For example:

Table 61. Pertitioned SORT output

Agent 1	**Agent 1**	**Agent 1**	**Agent 1**
C2	C3	C2	C3
5	1	8	6

Table 61. Pertitioned SORT output (continued)

Agent 1	Agent 1	Agent 1	Agent 1
5	1	8	6
5	2	8	7
5	2	8	7
1	4	2	1
1	5	2	2

Since the window specification for both OLAP functions is PARTITION BY rather than ORDER BY, and the outer sub-select doesn't have an ORDER BY clause, strict order on C2 and C3 does not need to be produced by the SORT. Distinct values of C2 just need to be processed by the same database agent. For example, all rows with values C2 = 5 must be processed by the same agent in order to properly determine the maximum value of C1 for that group of values. Limiting the sorting to values of C3 within distinct values of C2 reduces the memory required to perform the sort. Performance might also be improved because partitions can be emitted out of sequence, avoiding the synchronization overhead to emit them in order.

Since the data is not strictly ordered on C2, the order indicator is set to "R" indicating that it is randomly ordered.

A single SORT operation can provide the partitioning needed for multiple OLAP functions if the partitioning specifications are identical or are proper subsets of each other. When this is the case, the SORT will be partitioned on the the smallest set of columns that is a proper subset of all other OLAP partitioning specifications.

SQL and XQuery explain tool:

The **db2expln** command describes the access plan selected for SQL or XQuery statements.

You can use this tool to obtain a quick explanation of the chosen access plan when explain data was not captured. For static SQL and XQuery statements, **db2expln** examines the packages that are stored in the system catalog. For dynamic SQL and XQuery statements, **db2expln** examines the sections in the query cache.

The explain tool is located in the `bin` subdirectory of your instance `sqllib` directory. If **db2expln** is not in your current directory, it must be in a directory that appears in your PATH environment variable.

The **db2expln** command uses the `db2expln.bnd`, `db2exsrv.bnd`, and `db2exdyn.bnd` files to bind itself to a database the first time the database is accessed.

Description of db2expln output:

Explain output from the **db2expln** command includes both package information and section information for each package.

- Package information includes the date of the bind operation and relevant bind options
- Section information includes the section number and the SQL or XQuery statement being explained

Explain output pertaining to the chosen access plan for the SQL or XQuery statement appears under the section information.

The steps of an access plan, or section, are presented in the order that the database manager executes them. Each major step is shown as a left-aligned heading with information about that step indented under it. Indentation bars are displayed in the left margin of the explain output for an access plan. These bars also mark the scope of each operation. Operations at a lower level of indentation, farther to the right, are processed before those that appear in the previous level of indentation.

The chosen access plan is based on an augmented version of the original SQL statement, the *effective SQL statement* if statement concentrator is enabled, or the XQuery statement that is shown in the output. Because the query rewrite component of the compiler might convert the SQL or XQuery statement into an equivalent but more efficient format, the access plan shown in explain output might differ substantially from what you expect. The explain facility, which includes the explain tables, and the SET CURRENT EXPLAIN MODE statement, shows the actual SQL or XQuery statement that was used for optimization in the form of an SQL- or XQuery-like statement that is created by reverse-translating the internal representation of the query.

When you compare output from **db2expln** to output from the explain facility, the operator ID option (-opids) can be useful. Each time that **db2expln** begins processing a new operator from the explain facility, the operator ID number is printed to the left of the explained plan. The operator IDs can be used to compare steps in the different representations of the access plan. Note that there is not always a one-to-one correspondence between the operators in explain facility output and the operations shown by **db2expln**.

Table access information:

A statement in **db2expln** output provides the name and type of table being accessed.

Information about regular tables includes one of the following table access statements:

```
Access Table Name = schema.name  ID = ts,n
Access Hierarchy Table Name = schema.name  ID = ts,n
Access Materialized Query Table Name = schema.name  ID = ts,n
```

where:

- *schema.name* is the fully qualified name of the table that is being accessed
- ID is the corresponding TABLESPACEID and TABLEID from the SYSCAT.TABLES catalog view entry for the table

Information about temporary tables includes one of the following table access statements:

```
Access Temp Table  ID = tn
Access Global Temp Table  ID = ts,tn
```

where ID is the corresponding TABLESPACEID from the SYSCAT.TABLES catalog view entry for the table (*ts*) or the corresponding identifier that is assigned by **db2expln** (*tn*).

After the table access statement, the following more statements are provided to further describe the access.

- Number of columns
- Block access
- Parallel scan
- Scan direction
- Row access
- Lock intent
- Predicate
- Miscellaneous

Number of columns statement

The following statement indicates the number of columns that are being used from each row of the table:

```
#Columns = n
```

Block access statement

The following statement indicates that the table has one or more dimension block indexes that are defined on it:

```
Clustered by Dimension for Block Index Access
```

If this statement does not appear, the table was created without the ORGANIZE BY DIMENSIONS clause.

Parallel scan statement

The following statement indicates that the database manager uses several subagents to read the table in parallel:

```
Parallel Scan
```

If this statement does not appear, the table is read by only one agent (or subagent).

Scan direction statement

The following statement indicates that the database manager reads rows in reverse order:

```
Scan Direction = Reverse
```

If this statement does not appear, the scan direction is forward, which is the default.

Row access statements

One of the following statements indicates how qualifying rows in the table are being accessed.

- The `Relation Scan` statement indicates that the table is being sequentially scanned for qualifying rows.
  - The following statement indicates that no prefetching of data is done:

    ```
    Relation Scan
    |  Prefetch: None
    ```

  - The following statement indicates that the optimizer determined the number of pages that are prefetched:

```
Relation Scan
|  Prefetch: n Pages
```

– The following statement indicates that data must be prefetched:

```
Relation Scan
|  Prefetch: Eligible
```

- The following statement indicates that qualifying rows are being identified and accessed through an index:

```
Index Scan:  Name = schema.name  ID = xx
| Index type
| Index Columns:
```

where:

– *schema.name* is the fully qualified name of the index that is being scanned

– ID is the corresponding IID column in the SYSCAT.INDEXES catalog view

– Index type is one of:

```
Regular index (not clustered)
Regular index (clustered)
Dimension block index
Composite dimension block index
Index over XML data
```

This is followed by one line of output for each column in the index. Valid formats for this information are as follows:

```
n: column_name (Ascending)
n: column_name (Descending)
n: column_name (Include Column)
```

The following statements are provided to clarify the type of index scan.

– The range-delimiting predicates for the index are shown by the following statements:

```
#Key Columns = n
|  Start Key: xxxxx
|  Stop Key: xxxxx
```

where *xxxxx* is one of:

- `Start of Index`
- `End of Index`
- `Inclusive Value:` or `Exclusive Value:`

  An inclusive key value is included in the index scan. An exclusive key value is not be included in the scan. The value of the key is determined by one of the following items for each part of the key:

  ```
  n: 'string'
  n: nnn
  n: yyyy-mm-dd
  n: hh:mm:ss
  n: yyyy-mm-dd hh:mm:ss.uuuuuu
  n: NULL
  n: ?
  ```

  Only the first 20 characters of a literal string are displayed. A string is longer than 20 characters is indicated by an ellipsis (...) at the end of the string. Some keys cannot be determined until the section is run and is indicated by a question mark (?) as the value.

– `Index-Only Access`

  If all of the needed columns can be obtained from the index key, this statement displays and no table data are accessed.

- The following statement indicates that no prefetching of index pages is done:
  ```
  Index Prefetch: None
  ```
- The following statement indicates that for index prefetching sequential detection prefetching is enabled and it shows the MAXPAGES value for this type of prefetching denoted by *x*:
  ```
  Index Prefetch: Sequential (x)
  ```
- The following statement indicates that for index prefetching readahead prefetching is enabled:
  ```
  Index Prefetch: Readahead
  ```
- The following statement indicates that for index prefetching sequential detection and readahead prefetching are enabled. It also shows the MAXPAGES value for sequential detection prefetching that is denoted by *x*:
  ```
  Index Prefetch: Sequential (x), Readahead
  ```
- The following statement indicates that no prefetching of data pages is done:
  ```
  Data Prefetch: None
  ```
- The following statement indicates that for data prefetching sequential detection prefetching is enabled and it shows the MAXPAGES value for this type of prefetching denoted by *x*:
  ```
  Data Prefetch: Sequential (x)
  ```
- The following statement indicates that for data prefetching readahead prefetching is enabled:
  ```
  Data Prefetch: Readahead
  ```
- The following statement indicates that for data prefetching sequential detection and readahead prefetching are enabled. It also shows the MAXPAGES value for sequential detection prefetching that is denoted by *x*:
  ```
  Data Prefetch: Sequential (x), Readahead
  ```
- If there are predicates that can be passed to the index manager to help qualify index entries, the following statement is used to show the number of these predicates:
  ```
  Sargable Index Predicate(s)
  |  #Predicates = n
  ```

- When a statement indicates that qualifying rows are being identified and accessed through an index with an expression-based key, **db2explain** shows detailed information about the index.
  - This is followed by one line of output for each column in the index. Valid formats for this information are as follows:
    ```
    n: column_name (Ascending)
    n: column_name (Descending)
    n: column_name (Include Column)
    ```

  For example, if an index is created using the expression upper(name), salary+bonus, id, then the expression return the following **db2expln** output:
  ```
  |  |  Index Columns:
  |  |  |  1: K00[UPPER(NAME)] (Ascending)
  |  |  |  2: K01[SALARY+BONUS] (Ascending)
  |  |  |  3: ID (Ascending)
  ```
- If the qualifying rows are being accessed through row IDs (RIDs) that were prepared earlier in the access plan, this is indicated by the following statement:
  ```
  Fetch Direct Using Row IDs
  ```

  If the table has one or more block indexes that are defined on it, rows can be accessed by either block or row IDs. This is indicated by the following statement:

```
Fetch Direct Using Block or Row IOs
```

Lock intent statements

For each table access, the type of lock that are acquired at the table and row levels is shown with the following statement:

```
Lock Intents
|  Table: xxxx
|  Row  : xxxx
```

Possible values for a table lock are:

- Exclusive
- Intent Exclusive
- Intent None
- Intent Share
- Share
- Share Intent Exclusive
- Super Exclusive
- Update

Possible values for a row lock are:

- Exclusive
- Next Key Weak Exclusive
- None
- Share
- Update

Predicate statements

There are three types of statement that provide information about the predicates that are used in an access plan.

- The following statement indicates the number of predicates that are evaluated for each block of data that is retrieved from a blocked index:

  ```
  Block Predicates(s)
  |  #Predicates = n
  ```

- The following statement indicates the number of predicates that are evaluated while the data is being accessed. This number does not include pushdown operations, such as aggregation or sort:

  ```
  Sargable Predicate(s)
  |  #Predicates = n
  ```

- The following statement indicates the number of predicates that will be evaluated after the data is returned:

  ```
  Residual Predicate(s)
  |  #Predicates = n
  ```

The number of predicates that are shown in these statements might not reflect the number of predicates that are provided in the query statement, because predicates can be:

- Applied more than once within the same query
- Transformed and extended with the addition of implicit predicates during the query optimization process

- Transformed and condensed into fewer predicates during the query optimization process

Miscellaneous table statements

- The following statement indicates that only one row are accessed:

  ```
  Single Record
  ```

- The following statement appears when the isolation level that is used for table access is different from the isolation level for the statement:

  ```
  Isolation Level: xxxx
  ```

  There are a number of possible reasons for this. For example:

  – A package that was bound with the repeatable read (RR) isolation level is impacting certain referential integrity constraints; access to the parent table for checking these constraints is downgraded to the cursor stability (CS) isolation level to avoid holding unnecessary locks on this table.
  – A package that was bound with the uncommitted read (UR) isolation level includes a DELETE statement; access to the table for the delete operation is upgraded to CS.

- The following statement indicates that some or all of the rows that are read from a temporary table are cached outside of the buffer pool if sufficient **sortheap** memory is available:

  ```
  Keep Rows In Private Memory
  ```

- The following statement indicates that the table has the volatile cardinality attribute set:

  ```
  Volatile Cardinality
  ```

Temporary table information:

A temporary table is used as a work table during access plan execution. Generally, temporary tables are used when subqueries need to be evaluated early in the access plan, or when intermediate results will not fit into the available memory.

If a temporary table is needed, one of the following statements will appear in **db2expln** command output.

```
Insert Into Temp Table  ID = tn     --> ordinary temporary table
Insert Into Shared Temp Table  ID = tn     --> ordinary temporary table will be created
                                               by multiple subagents in parallel
Insert Into Sorted Temp Table  ID = tn     --> sorted temporary table
Insert Into Sorted Shared Temp Table  ID = tn     --> sorted temporary table will be created
                                                      by multiple subagents in parallel

Insert Into Global Temp Table  ID = ts,tn     --> declared global temporary table
Insert Into Shared Global Temp Table  ID = ts,tn     --> declared global temporary table
                                                          will be created by multiple subagents
                                                          in parallel
Insert Into Sorted Global Temp Table  ID = ts,tn     --> sorted declared global temporary table
Insert Into Sorted Shared Global Temp Table  ID = ts,tn     --> sorted declared global temporary
                                                                table will be created by
                                                                multiple subagents in parallel
```

The ID is an identifier that is assigned by **db2expln** for convenience when referring to the temporary table. This ID is prefixed with the letter 't' to indicate that the table is a temporary table.

Each of these statements is followed by:

```
#Columns = n
```

which indicates how many columns there are in each row that is being inserted into the temporary table.

Sorted temporary tables

Sorted temporary tables can result from such operations as:

- ORDER BY
- DISTINCT
- GROUP BY
- Merge join
- '= ANY' subquery
- '<> ALL' subquery
- INTERSECT or EXCEPT
- UNION (without the ALL keyword)

A number of statements that are associated with a sorted temporary table can appear in **db2expln** command output.

- The following statement indicates the number of key columns that are used in the sort:

  ```
  #Sort Key Columns = n
  ```

  One of the following lines is displayed for each column in the sort key:

  ```
  Key n: column_name (Ascending)
  Key n: column_name (Descending)
  Key n: (Ascending)
  Key n: (Descending)
  ```

- The following statements provide estimates of the number of rows and the row size so that the optimal sort heap can be allocated at run time:

  ```
  Sortheap Allocation Parameters:
  |  #Rows     = n
  |  Row Width = n
  ```

- The following statement is displayed if only the first rows of the sorted result are needed:

  ```
  Sort Limited To Estimated Row Count
  ```

- For sorts that are performed in a symmetric multiprocessor (SMP) environment, the type of sort that is to be performed is indicated by one of the following statements:

  ```
  Use Partitioned Sort
  Use Shared Sort
  Use Replicated Sort
  Use Round-Robin Sort
  ```

- The following statements indicate whether or not the sorted result will be left in the sort heap:

  ```
  Piped
  Not Piped
  ```

  If a piped sort is indicated, the database manager will keep the sorted output in memory, rather than placing it in another temporary table.

- The following statement indicates that duplicate values will be removed during the sort operation:

  ```
  Duplicate Elimination
  ```

- If aggregation is being performed during the sort operation, one of the following statements is displayed:

```
Partial Aggregation
Intermediate Aggregation
Buffered Partial Aggregation
Buffered Intermediate Aggregation
```

Temporary table completion

A completion statement is displayed whenever a temporary table is created within the scope of a table access. This statement can be one of the following:

```
Temp Table Completion  ID = tn
Shared Temp Table Completion  ID = tn
Sorted Temp Table Completion  ID = tn
Sorted Shared Temp Table Completion  ID = tn
```

Table functions

Table functions are user-defined functions (UDFs) that return data to the statement in the form of a table. A table function is indicated by the following statements, which detail the attributes of the function. The specific name uniquely identifies the table function that is invoked.

```
Access User Defined Table Function
|  Name = schema.funcname
|  Specific Name = specificname
|  SQL Access Level = accesslevel
|  Language = lang
|  Parameter Style = parmstyle
|  Fenced                      Not Deterministic
|  Called on NULL Input        Disallow Parallel
|  Not Federated               Not Threadsafe
```

Binary join information:

Output from the **db2expln** command can contain information about joins in an explained statement.

Whenever a binary join is performed, one of the following statements is displayed:

```
Hash Join
Merge Join
Nested Loop Join
```

A left outer join is indicated by one of the following statements:

```
Left Outer Hash Join
Left Outer Merge Join
Left Outer Nested Loop Join
```

In the case of a merge or nested loop join, the outer table of the join is the table that was referenced in the previous access statement (shown in the output). The inner table of the join is the table that was referenced in the access statement that is contained within the scope of the join statement. In the case of a hash join, the access statements are reversed: the outer table is contained within the scope of the join, and the inner table appears before the join.

In the case of a hash or merge join, the following additional statements might appear:

- `Early Out: Single Match Per Outer Row`

  In some circumstances, a join simply needs to determine whether any row in the inner table matches the current row in the outer table.

- `Residual Predicate(s)`
  `| #Predicates = n`

  It is possible to apply predicates after a join has completed. This statement displays the number of predicates being applied.

In the case of a hash join, the following additional statements might appear:

- `Process Hash Table For Join`

  The hash table is built from the inner table. This statement displays if the building of the hash table was pushed down into a predicate during access to the inner table.
- `Process Probe Table For Hash Join`

  While accessing the outer table, a probe table can be built to improve the performance of the join. This statement displays if a probe table was built during access to the outer table.
- `Estimated Build Size: n`

  This statement displays the estimated number of bytes that are needed to build the hash table.
- `Estimated Probe Size: n`

  This statement displays the estimated number of bytes that are needed to build the probe table.

In the case of a nested loop join, the following statement might appear immediately after the join statement:

```
Piped Inner
```

This statement indicates that the inner table of the join is the result of another series of operations. This is also referred to as a *composite inner*.

If a join involves more than two tables, the explain steps should be read from top to bottom. For example, suppose the explain output has the following flow:

```
Access ..... W
Join
|  Access ..... X
Join
|  Access ..... Y
Join
|  Access ..... Z
```

The steps of execution would be:

1. Take a qualifying row from table W.
2. Join a row from W with the next row from table X and call the result P1 (for partial join result number 1).
3. Join P1 with the next row from table Y to create P2.
4. Join P2 with the next row from table Z to create one complete result row.
5. If there are more rows in Z, go to step 4.
6. If there are more rows in Y, go to step 3.
7. If there are more rows in X, go to step 2.
8. If there are more rows in W, go to step 1.

Data stream information:

Within an access plan, there is often a need to control the creation and flow of data from one series of operations to another. The data stream concept enables a group of operations within an access plan to be controlled as a unit.

The start of a data stream is indicated by the following statement in **db2expln** output:

```
Data Stream n
```

where *n* is a unique identifier assigned by **db2expln** for ease of reference.

The end of a data stream is indicated by:

```
End of Data Stream n
```

All operations between these statements are considered to be part of the same data stream.

A data stream has a number of characteristics, and one or more statements can follow the initial data stream statement to describe these characteristics:

- If the operation of the data stream depends on a value that is generated earlier in the access plan, the data stream is marked with:

  ```
  Correlated
  ```

- Similar to a sorted temporary table, the following statements indicate whether or not the results of the data stream will be kept in memory:

  ```
  Piped
  Not Piped
  ```

  A piped data stream might be written to disk if there is insufficient memory at execution time. The access plan provides for both possibilities.

- The following statement indicates that only a single record is required from this data stream:

  ```
  Single Record
  ```

When a data stream is accessed, the following statement will appear in the output:

```
Access Data Stream n
```

Insert, update, and delete information:

The explain text for the INSERT, UPDATE, or DELETE statement is self-explanatory.

Statement text for these SQL operations in **db2expln** output can be:

```
Insert: Table Name = schema.name ID = ts,n
Update: Table Name = schema.name ID = ts,n
Delete: Table Name = schema.name ID = ts,n
Insert: Hierarchy Table Name = schema.name ID = ts,n
Update: Hierarchy Table Name = schema.name ID = ts,n
Delete: Hierarchy Table Name = schema.name ID = ts,n
Insert: Materialized Query Table = schema.name ID = ts,n
Update: Materialized Query Table = schema.name ID = ts,n
Delete: Materialized Query Table = schema.name ID = ts,n
Insert: Global Temporary Table ID = ts, tn
Update: Global Temporary Table ID = ts, tn
Delete: Global Temporary Table ID = ts, tn
```

Block and row identifier preparation information:

For some access plans, it is more efficient if the qualifying row and block identifiers are sorted and duplicates are removed (in the case of index ORing), or if a technique is used to determine which identifiers appear in all of the indexes being accessed (in the case of index ANDing) before the table is accessed.

There are three main uses of the identifier preparation information that is shown in explain output:

- Either of the following statements indicates that Index ORing was used to prepare the list of qualifying identifiers:

  ```
  Index ORing Preparation
  Block Index ORing Preparation
  ```

  Index ORing refers to the technique of accessing more than one index and combining the results to include the distinct identifiers that appear in any of the indexes. The optimizer considers index ORing when predicates are connected by OR keywords or there is an IN predicate.
- Either of the following statements indicates that input data was prepared for use during list prefetching:

  ```
  List Prefetch Preparation
  Block List Prefetch RID Preparation
  ```
- *Index ANDing* refers to the technique of accessing more than one index and combining the results to include the identifiers that appear in all of the accessed indexes. Index ANDing begins with either of the following statements:

  ```
  Index ANDing
  Block Index ANDing
  ```

  If the optimizer has estimated the size of the result set, the estimate is shown with the following statement:

  ```
  Optimizer Estimate of Set Size: n
  ```

  Index ANDing filter operations process identifiers and use bit filter techniques to determine the identifiers that appear in every accessed index. The following statements indicate that identifiers were processed for index ANDing:

  ```
  Index ANDing Bitmap Build Using Row IDs
  Index ANDing Bitmap Probe Using Row IDs
  Index ANDing Bitmap Build and Probe Using Row IDs
  Block Index ANDing Bitmap Build Using Block IDs
  Block Index ANDing Bitmap Build and Probe Using Block IDs
  Block Index ANDing Bitmap Build and Probe Using Row IDs
  Block Index ANDing Bitmap Probe Using Block IDs and Build Using Row IDs
  Block Index ANDing Bitmap Probe Using Block IDs
  Block Index ANDing Bitmap Probe Using Row IDs
  ```

  If the optimizer has estimated the size of the result set for a bitmap, the estimate is shown with the following statement:

  ```
  Optimizer Estimate of Set Size: n
  ```

If list prefetching can be performed for any type of identifier preparation, it will be so indicated with the following statement:

```
Prefetch: Enabled
```

Aggregation information:

Aggregation is performed on rows satisfying criteria that are represented by predicates in an SQL statement.

If an aggregate function executes, one of the following statements appears in **db2expln** output:

```
Aggregation
Predicate Aggregation
Partial Aggregation
Partial Predicate Aggregation
Hashed Partial Aggregation
Hashed Partial Predicate Aggregation
Intermediate Aggregation
Intermediate Predicate Aggregation
Final Aggregation
Final Predicate Aggregation
```

Predicate aggregation means that the aggregation operation was processed as a predicate when the data was accessed.

The aggregation statement is followed by another statement that identifies the type of aggregate function that was performed:

```
Group By
Column Function(s)
Single Record
```

The specific column function can be derived from the original SQL statement. A single record is fetched from an index to satisfy a MIN or MAX operation.

If predicate aggregation has been performed, there is an aggregation completion operation and corresponding output:

```
Aggregation Completion
Partial Aggregation Completion
Hashed Partial Aggregation Completion
Intermediate Aggregation Completion
Final Aggregation Completion
```

Parallel processing information:

Executing an SQL statement in parallel (using either intrapartition or interpartition parallelism) requires some special access plan operations.

- When running an intrapartition parallel plan, portions of the plan are executed simultaneously using several subagents. The creation of these subagents is indicated by the statement in output from the **db2expln** command:

  ```
  Process Using n Subagents
  ```

- When running an interpartition parallel plan, the section is broken into several subsections. Each subsection is sent to one or more database partitions to be run. An important subsection is the *coordinator subsection*. The coordinator subsection is the first subsection in every plan. It acquires control first, and is responsible for distributing the other subsections and returning results to the calling application.
  - The distribution of subsections is indicated by the following statement:

    ```
    Distribute Subsection #n
    ```

  - The following statement indicates that the subsection will be sent to a database partition within the database partition group, based on the value of the columns.

```
Directed by Hash
|  #Columns = n
|  Partition Map ID = n, Nodegroup = ngname, #Nodes = n
```

- The following statement indicates that the subsection will be sent to a predetermined database partition. (This is common when the statement uses the DBPARTITIONNUM() scalar function.)

  ```
  Directed by Node Number
  ```

- The following statement indicates that the subsection will be sent to the database partition that corresponds to a predetermined database partition number in the database partition group. (This is common when the statement uses the HASHEDVALUE scalar function.)

  ```
  Directed by Partition Number
  |  Partition Map ID = n, Nodegroup = ngname, #Nodes = n
  ```

- The following statement indicates that the subsection will be sent to the database partition that provided the current row for the application's cursor.

  ```
  Directed by Position
  ```

- The following statement indicates that only one database partition, determined when the statement was compiled, will receive the subsection.

  ```
  Directed to Single Node
  |  Node Number = n
  ```

- Either of the following statements indicates that the subsection will be executed on the coordinator database partition.

  ```
  Directed to Application Coordinator Node
  Directed to Local Coordinator Node
  ```

- The following statement indicates that the subsection will be sent to all of the listed database partitions.

  ```
  Broadcast to Node List
  |  Nodes = n1, n2, n3, ...
  ```

- The following statement indicates that only one database partition, determined as the statement is executing, will receive the subsection.

  ```
  Directed to Any Node
  ```

- Table queues are used to move data between subsections in a partitioned database environment or between subagents in a symmetric multiprocessor (SMP) environment.
  - The following statements indicate that data is being inserted into a table queue:

    ```
    Insert Into Synchronous Table Queue  ID = qn
    Insert Into Asynchronous Table Queue  ID = qn
    Insert Into Synchronous Local Table Queue  ID = qn
    Insert Into Asynchronous Local Table Queue  ID = qn
    ```

  - For database partition table queues, the destination for rows that are inserted into the table queue is described by one of the following statements:

    Each row is sent to the coordinator database partition:

    ```
    Broadcast to Coordinator Node
    ```

    Each row is sent to every database partition on which the given subsection is running:

    ```
    Broadcast to All Nodes of Subsection n
    ```

    Each row is sent to a database partition that is based on the values in the row:

    ```
    Hash to Specific Node
    ```

Each row is sent to a database partition that is determined while the statement is executing:

```
Send to Specific Node
```

Each row is sent to a randomly determined database partition:

```
Send to Random Node
```

- In some situations, a database partition table queue will have to overflow some rows to a temporary table. This possibility is identified by the following statement:

```
Rows Can Overflow to Temporary Table
```

- After a table access that includes a pushdown operation to insert rows into a table queue, there is a "completion" statement that handles rows that could not be sent immediately. In this case, one of the following lines is displayed:

```
Insert Into Synchronous Table Queue Completion  ID = qn
Insert Into Asynchronous Table Queue Completion  ID = qn
Insert Into Synchronous Local Table Queue Completion  ID = qn
Insert Into Asynchronous Local Table Queue Completion  ID = qn
```

- The following statements indicate that data is being retrieved from a table queue:

```
Access Table Queue  ID = qn
Access Local Table Queue  ID = qn
```

These statements are always followed by the number of columns being retrieved.

```
#Columns = n
```

- If the table queue sorts the rows at the receiving end, one of the following statements appears:

```
Output Sorted
Output Sorted and Unique
```

These statements are followed by the number of keys being used for the sort operation.

```
#Key Columns = n
```

For each column in the sort key, one of the following statements is displayed:

```
Key n: (Ascending)
Key n: (Descending)
```

- If predicates will be applied to rows at the receiving end of the table queue, the following statement appears:

```
Residual Predicate(s)
|  #Predicates = n
```

- Some subsections in a partitioned database environment explicitly loop back to the start of the subsection, and the following statement is displayed:

```
Jump Back to Start of Subsection
```

Federated query information:

Executing an SQL statement in a federated database requires the ability to perform portions of the statement on other data sources.

The following output from the **db2expln** command indicates that a data source will be read:

```
Ship Distributed Subquery #n
|   #Columns = n
```

If predicates are applied to data that is returned from a distributed subquery, the number of predicates being applied is indicated by the following statements:

```
Residual Predicate(s)
|   #Predicates = n
```

An insert, update, or delete operation that occurs at a data source is indicated by one of the following statements:

```
Ship Distributed Insert #n
Ship Distributed Update #n
Ship Distributed Delete #n
```

If a table is explicitly locked at a data source, the following statement appears:

```
Ship Distributed Lock Table #n
```

Data definition language (DDL) statements against a data source are split into two parts. The part that is invoked at the data source is indicated by the following statement:

```
Ship Distributed DDL Statement #n
```

If the federated server is a partitioned database, part of the DDL statement must be run at the catalog database partition. This is indicated by the following statement:

```
Distributed DDL Statement #n Completion
```

The details for each distributed sub-statement are displayed separately.

- The data source for the subquery is indicated by one of the following statements:

  ```
  Server: server_name (type, version)
  Server: server_name (type)
  Server: server_name
  ```

- If the data source is relational, the SQL for the sub-statement is displayed as follows:

  ```
  SQL Statement:
       statement
  ```

  Non-relational data sources are indicated with:

  ```
  Non-Relational Data Source
  ```

- Nicknames that are referenced in the sub-statement are listed as follows:

  ```
  Nicknames Referenced:
     schema.nickname ID = n
  ```

  If the data source is relational, the base table for the nickname is displayed as follows:

  ```
  Base = baseschema.basetable
  ```

  If the data source is non-relational, the source file for the nickname is displayed as follows:

  ```
  Source File = filename
  ```

- If values are passed from the federated server to the data source before executing the sub-statement, the number of values is indicated by the following statement:

  ```
  #Input Columns: n
  ```

- If values are passed from the data source to the federated server after executing the sub-statement, the number of values is indicated by the following statement:

  ```
  #Output Columns: n
  ```

Miscellaneous explain information:

Output from the **db2expln** command contains additional useful information that cannot be readily classified.

- Sections for data definition language (DDL) statements are indicated in the output with the following statement:

  ```
  DDL Statement
  ```

  No additional explain output is provided for DDL statements.
- Sections for SET statements pertaining to updatable special registers, such as CURRENT EXPLAIN SNAPSHOT, are indicated in the output with the following statement:

  ```
  SET Statement
  ```

  No additional explain output is provided for SET statements.
- If the SQL statement contains a DISTINCT clause, the following statement might appear in the output:

  ```
  Distinct Filter  #Columns = n
  ```

  where *n* is the number of columns involved in obtaining distinct rows. To retrieve distinct row values, the rows must first be sorted to eliminate duplicates. This statement will not appear if the database manager does not have to explicitly eliminate duplicates, as in the following cases:
  - A unique index exists and all of the columns in the index key are part of the DISTINCT operation
  - Duplicates can be eliminated during sorting
- The following statement appears if a partial early distinct (PED) operation was performed to remove many, if not all, duplicates. This reduces the amount of data that must be processed later in the query evaluation.

  ```
  Hashed partial distinct filter
  ```
- The following statement appears if the next operation is dependent on a specific record identifier:

  ```
  Positioned Operation
  ```

  If the positioned operation is against a federated data source, the statement becomes:

  ```
  Distributed Positioned Operation
  ```

  This statement appears for any SQL statement that uses the WHERE CURRENT OF syntax.
- The following statement appears if there are predicates that must be applied to the result but that could not be applied as part of another operation:

  ```
  Residual Predicate Application
  |  #Predicates = n
  ```
- The following statement appears if the SQL statement contains a UNION operator:

  ```
  UNION
  ```
- The following statement appears if there is an operation in the access plan whose sole purpose is to produce row values for use by subsequent operations:

  ```
  Table Constructor
  |  n-Row(s)
  ```

Table constructors can be used for transforming values in a set into a series of rows that are then passed to subsequent operations. When a table constructor is prompted for the next row, the following statement appears:

```
Access Table Constructor
```

- The following statement appears if there is an operation that is only processed under certain conditions:

```
Conditional Evaluation
|  Condition #n:
|  #Predicates = n
|  Action #n:
```

  Conditional evaluation is used to implement such activities as the CASE statement, or internal mechanisms such as referential integrity constraints or triggers. If no action is shown, then only data manipulation operations are processed when the condition is true.

- One of the following statements appears if an ALL, ANY, or EXISTS subquery is being processed in the access plan:

```
ANY/ALL Subquery
EXISTS Subquery
EXISTS SINGLE Subquery
```

- Prior to certain update or delete operations, it is necessary to establish the position of a specific row within the table. This is indicated by the following statement:

```
Establish Row Position
```

- The following statement appears if rows are being returned to the application:

```
Return Data to Application
|  #Columns = n
```

  If the operation was pushed down into a table access, a completion phase statement appears in the output:

```
Return Data Completion
```

- The following statements appear if a stored procedure is being invoked:

```
Call Stored Procedure
|   Name = schema.funcname
|   Specific Name = specificname
|   SQL Access Level = accesslevel
|   Language = lang
|   Parameter Style = parmstyle
|   Expected Result Sets = n
|   Fenced                      Not Deterministic
|   Called on NULL Input        Disallow Parallel
|   Not Federated               Not Threadsafe
```

- The following statement appears if one or more large object (LOB) locators are being freed:

```
Free LOB Locators
```

Access plan optimization

Access plans can be optimized in an attempt to improve query performance. The degree of improvement depends on the type of optimization chosen. Optimizing access plans is one of the best ways to ensure that the query compiler behaves the way you expect and design it to.

Statement concentrator:

The statement concentrator modifies dynamic SQL statements at the database server so that similar SQL statements can share the access plan, thus improving performance.

In online transaction processing (OLTP), simple statements might repeatedly be generated with different literal values. In such workloads, the cost of recompiling the statements can add significant memory usage. The statement concentrator avoids this memory usage by allowing compiled statements to be reused, regardless of the values of the literals. The memory usage that is associated with modifying the incoming SQL statements for a OLTP workload is small for the statement concentrator, when compared to the savings that are realized by reusing statements that are in the package cache.

The statement concentrator is disabled by default. You can enable it for all dynamic statements in a database by setting the **stmt_conc** database configuration parameter to LITERALS.

If a dynamic statement is modified as a result of statement concentration, both the original statement and the modified statement are displayed in the explain output. The event monitor logical monitor elements and output from the MON_GET_ACTIVITY_DETAILS table function show the original statement if the statement concentrator modified the original statement text. Other monitoring interfaces show only the modified statement text.

Consider the following example, in which the **stmt_conc** database configuration parameter is set to LITERALS and the following two statements are issued:

```
select firstnme, lastname from employee where empno='000020'
select firstnme, lastname from employee where empno='000070'
```

These statements share the entry in the package cache, and that entry uses the following statement:

```
select firstnme, lastname from employee where empno=:L0
```

The data server provides a value for :L0 (either '000020' or '000070'), based on the literal that was used in the original statements.

The statement concentrator requires that the length attributes for VARCHAR and VARGRAPHIC string literals to be greater than the lengths of the string literals.

The statement concentrator might cause some built-in functions to return different result types. For example, REPLACE can return a different type when statement concentrator is used. The WORKDEPT column is defined as CHAR(3), the following query returns VARCHAR(3) when statement concentrator is disabled:

```
SELECT REPLACE(WORKDEPT,'E2','E9') FROM EMPLOYEE
```

When **stmt_conc**=LITERALS, the two string literals are replaced with parameter markers and the return type is VARCHAR(6).

Because statement concentration alters the statement text, statement concentration impacts access plan selection. The statement concentrator works best when similar statements in the package cache have similar access plans. If different literal values in a statement result in different access plans or the value of a literal makes a significant difference in plan selection and execution time (for example, if the

presence of the literal allows an expression to match an expression-based index key), then do not enable the statement concentrator for that statement.

Access plan reuse:

You can request that the access plans that are chosen for static SQL statements in a package stay the same as, or be very similar to, existing access plans across several bind or rebind operations.

Access plan reuse can prevent significant plan changes from occurring without your explicit approval. Although this can mean that your queries do not benefit from potential access plan improvements, the control that access plan reuse provides will give you the ability to test and implement those improvements when you are ready to do so. Until then, you can continue to use existing access plans for stable and predictable performance.

Access plan reuse is also referred to as plan lockdown.

Enable access plan reuse through the ALTER PACKAGE statement, or by using the APREUSE option on the **BIND**, **REBIND**, or **PRECOMPILE** command. Packages that are subject to access plan reuse have the value Y in the APREUSE column of the SYSCAT.PACKAGES catalog view.

The ALTER_ROUTINE_PACKAGE procedure is a convenient way to enable access plan reuse for compiled SQL objects, such as SQL procedures. However, access plans cannot be reused during compiled object revalidation, because the object is dropped before being rebound. In this case, APREUSE will only take effect the next time that the package is bound or rebound.

Access plan reuse is most effective when changes to the schema and compilation environment are kept to a minimum. If significant changes are made, it might not be possible to recreate the previous access plan. Examples of such significant changes include dropping an index that is being used in an access plan, or recompiling an SQL statement at a different optimization level. Significant changes to the query compiler's analysis of the statement can also result in the previous access plan no longer being reusable.

You can combine access plan reuse with optimization guidelines. A statement-level guideline takes precedence over access plan reuse for the static SQL statement to which it applies. Access plans for static statements that do not have statement-level guidelines can be reused if they do not conflict with any general optimization guidelines that have been specified. A statement profile with an empty guideline can be used to disable access plan reuse for a specific statement, while leaving plan reuse available for the other static statements in the package.

Note: Access plans from packages that were produced by releases prior to Version 9.7 cannot be reused.

If an access plan cannot be reused, compilation continues, but a warning (SQL20516W) is returned with a reason code that indicates why the attempt to reuse the access plan was not successful. Additional information is sometimes provided in the diagnostic messages that are available through the explain facility.

Optimization classes:

When you compile an SQL or XQuery statement, you can specify an optimization class that determines how the optimizer chooses the most efficient access plan for that statement.

The optimization classes differ in the number and type of optimization strategies that are considered during the compilation of a query. Although you can specify optimization techniques individually to improve runtime performance for the query, the more optimization techniques that you specify, the more time and system resources query compilation will require.

You can specify one of the following optimization classes when you compile an SQL or XQuery statement.

0 This class directs the optimizer to use minimal optimization when generating an access plan, and has the following characteristics:

- Frequent-value statistics are not considered by the optimizer.
- Only basic query rewrite rules are applied.
- Greedy join enumeration is used.
- Only nested loop join and index scan access methods are enabled.
- List prefetch is not used in generated access methods.
- The star-join strategy is not considered.

This class should only be used in circumstances that require the lowest possible query compilation overhead. Query optimization class 0 is appropriate for an application that consists entirely of very simple dynamic SQL or XQuery statements that access well-indexed tables.

1 This optimization class has the following characteristics:

- Frequent-value statistics are not considered by the optimizer.
- Only a subset of query rewrite rules are applied.
- Greedy join enumeration is used.
- List prefetch is not used in generated access methods.

Optimization class 1 is similar to class 0, except that merge scan joins and table scans are also available.

2 This class directs the optimizer to use a degree of optimization that is significantly higher than class 1, while keeping compilation costs for complex queries significantly lower than class 3 or higher. This optimization class has the following characteristics:

- All available statistics, including frequent-value and quantile statistics, are used.
- All query rewrite rules (including materialized query table routing) are applied, except computationally intensive rules that are applicable only in very rare cases.
- Greedy join enumeration is used.
- A wide range of access methods is considered, including list prefetch and materialized query table routing.
- The star-join strategy is considered, if applicable.

Optimization class 2 is similar to class 5, except that it uses greedy join enumeration instead of dynamic programming join enumeration. This class has the most optimization of all classes that use the greedy join

enumeration algorithm, which considers fewer alternatives for complex queries, and therefore consumes less compilation time than class 3 or higher. Class 2 is recommended for very complex queries in a decision support or online analytic processing (OLAP) environment. In such environments, a specific query is not likely to be repeated in exactly the same way, so that an access plan is unlikely to remain in the cache until the next occurrence of the query.

3 This class represents a moderate amount of optimization, and comes closest to matching the query optimization characteristics of Db2 for z/OS. This optimization class has the following characteristics:

- Frequent-value statistics are used, if available.
- Most query rewrite rules are applied, including subquery-to-join transformations.
- Dynamic programming join enumeration is used, with:
  - Limited use of composite inner tables
  - Limited use of Cartesian products for star schemas involving lookup tables
- A wide range of access methods is considered, including list prefetch, index ANDing, and star joins.

This class is suitable for a broad range of applications, and improves access plans for queries with four or more joins.

5 This class directs the optimizer to use a significant amount of optimization to generate an access plan, and has the following characteristics:

- All available statistics, including frequent-value and quantile statistics, are used.
- All query rewrite rules (including materialized query table routing) are applied, except computationally intensive rules that are applicable only in very rare cases.
- Dynamic programming join enumeration is used, with:
  - Limited use of composite inner tables
  - Limited use of Cartesian products for star schemas involving lookup tables
- A wide range of access methods is considered, including list prefetch, index ANDing, and materialized query table routing.

Optimization class 5 (the default) is an excellent choice for a mixed environment with both transaction processing and complex queries. This optimization class is designed to apply the most valuable query transformations and other query optimization techniques in an efficient manner.

If the optimizer detects that additional resources and processing time for complex dynamic SQL or XQuery statements are not warranted, optimization is reduced. The extent of the reduction depends on the machine size and the number of predicates. When the optimizer reduces the amount of query optimization, it continues to apply all of the query rewrite rules that would normally be applied. However, it uses greedy join enumeration and it reduces the number of access plan combinations that are considered.

7 This class directs the optimizer to use a significant amount of optimization to generate an access plan. It is similar to optimization class 5, except that

in this case, the optimizer never considers reducing the amount of query optimization for complex dynamic SQL or XQuery statements.

9 This class directs the optimizer to use all available optimization techniques. These include:

- All available statistics
- All query rewrite rules
- All possibilities for join enumeration, including Cartesian products and unlimited composite inners
- All access methods

This class increases the number of possible access plans that are considered by the optimizer. You might use this class to determine whether more comprehensive optimization would generate a better access plan for very complex or very long-running queries that use large tables. Use explain and performance measurements to verify that a better plan has actually been found.

Choosing an optimization class:

Setting the optimization class can provide some of the advantages of explicitly specifying optimization techniques.

This is true, particularly when:

- Managing very small databases or very simple dynamic queries
- Accommodating memory limitations on your database server at compile time
- Reducing query compilation time; for example, during statement preparation

Most statements can be adequately optimized with a reasonable amount of resource by using the default optimization class 5. Query compilation time and resource consumption are primarily influenced by the complexity of a query; in particular, by the number of joins and subqueries. However, compilation time and resource consumption are also affected by the amount of optimization that is performed.

Query optimization classes 1, 2, 3, 5, and 7 are all suitable for general use. Consider class 0 only if you require further reductions in query compilation time, and the SQL and XQuery statements are very simple.

Tip: To analyze a long-running query, run the query with **db2batch** to determine how much time is spent compiling and executing the query. If compilation time is excessive, reduce the optimization class. If execution time is a problem, consider a higher optimization class.

When you select an optimization class, consider the following general guidelines:

- Start by using the default query optimization class 5.
- When choosing a class other than the default, try class 1, 2, or 3 first. Classes 0, 1, and 2 use the greedy join enumeration algorithm.
- Use optimization class 1 or 2 if you have many tables with many join predicates on the same column, and if compilation time is a concern.
- Use a low optimization class (0 or 1) for queries that have very short run times of less than one second. Such queries tend to:
  - Access a single table or only a few tables
  - Fetch a single row or only a few rows

  - Use fully qualified and unique indexes
  - Be involved in online transaction processing (OLTP)
- Use a higher optimization class (3, 5, or 7) for queries that have longer run times of more than 30 seconds.
- Class 3 or higher uses the dynamic programming join enumeration algorithm, which considers many more alternative plans, and might incur significantly more compilation time than classes 0, 1, or 2, especially as the number of tables increases.
- Use optimization class 9 only if you have extraordinary optimization requirements for a query.

Complex queries might require different amounts of optimization to select the best access plan. Consider using higher optimization classes for queries that have:

- Access to large tables
- A large number of views
- A large number of predicates
- Many subqueries
- Many joins
- Many set operators, such as UNION or INTERSECT
- Many qualifying rows
- GROUP BY and HAVING operations
- Nested table expressions

Decision support queries or month-end reporting queries against fully normalized databases are good examples of complex queries for which at least the default query optimization class should be used.

Use higher query optimization classes for SQL and XQuery statements that were produced by a query generator. Many query generators create inefficient queries. Poorly written queries require additional optimization to select a good access plan. Using query optimization class 2 or higher can improve such queries.

For SAP applications, always use optimization class 5. This optimization class enables many Db2 features optimized for SAP, such as setting the **DB2_REDUCED_OPTIMIZATION** registry variable.

In a federated database, the optimization class does not apply to the remote optimizer.

Setting the optimization class:

When you specify an optimization level, consider whether a query uses static or dynamic SQL and XQuery statements, and whether the same dynamic query is repeatedly executed.

About this task

For static SQL and XQuery statements, the query compilation time and resources are expended only once, and the resulting plan can be used many times. In general, static SQL and XQuery statements should always use the default query optimization class (5). Because dynamic statements are bound and executed at run time, consider whether the overhead of additional optimization for dynamic statements improves overall performance. However, if the same dynamic SQL or

XQuery statement is executed repeatedly, the selected access plan is cached. Such statements can use the same optimization levels as static SQL and XQuery statements.

If you are not sure whether a query might benefit from additional optimization, or you are concerned about compilation time and resource consumption, consider benchmark testing.

Procedure

To specify a query optimization class:

1. Analyze performance factors.
   - For a dynamic query statement, tests should compare the average run time for the statement. Use the following formula to estimate the average run time:

     ```
     compilation time + sum of execution times for all iterations
     ------------------------------------------------------------
                          number of iterations
     ```

     The number of iterations represents the number of times that you expect the statement might be executed each time that it is compiled.

     Note: After initial compilation, dynamic SQL and XQuery statements are recompiled whenever a change to the environment requires it. If the environment does not change after a statement is cached, subsequent PREPARE statements reuse the cached statement.
   - For static SQL and XQuery statements, compare the statement run times.

     Although you might also be interested in the compilation time of static SQL and XQuery statements, the total compilation and execution time for a static statement is difficult to assess in any meaningful context. Comparing the total times does not recognize the fact that a static statement can be executed many times whenever it is bound, and that such a statement is generally not bound during run time.
2. Specify the optimization class.
   - Dynamic SQL and XQuery statements use the optimization class that is specified by the CURRENT QUERY OPTIMIZATION special register. For example, the following statement sets the optimization class to 1:

     ```
     SET CURRENT QUERY OPTIMIZATION = 1
     ```

     To ensure that a dynamic SQL or XQuery statement always uses the same optimization class, include a SET statement in the application program.

     If the CURRENT QUERY OPTIMIZATION special register has not been set, dynamic statements are bound using the default query optimization class. The default value for both dynamic and static queries is determined by the value of the **dft_queryopt** database configuration parameter, whose default value is 5. The default values for the bind option and the special register are also read from the **dft_queryopt** database configuration parameter.
   - Static SQL and XQuery statements use the optimization class that is specified on the **PREP** and **BIND** commands. The QUERYOPT column in the SYSCAT.PACKAGES catalog view records the optimization class that is used to bind a package. If the package is rebound, either implicitly or by using the **REBIND PACKAGE** command, this same optimization class is used for static statements. To change the optimization class for such static SQL and XQuery statements, use the **BIND** command. If you do not specify the optimization

class, the data server uses the default optimization class, as specified by the **dft_queryopt** database configuration parameter.

Using optimization profiles if other tuning options do not produce acceptable results:

If you have followed best practices recommendations, but you believe that you are still getting less than optimal performance, you can provide explicit optimization guidelines to the Db2 optimizer.

These optimization guidelines are contained in an XML document called the optimization profile. The profile defines SQL statements and their associated optimization guidelines.

If you use optimization profiles extensively, they require a lot of effort to maintain. More importantly, you can only use optimization profiles to improve performance for existing SQL statements. Following best practices consistently can help you to achieve query performance stability for all queries, including future ones.

Optimization profiles and guidelines:

An optimization profile is an XML document that can contain optimization guidelines for one or more SQL statements. The correspondence between each SQL statement and its associated optimization guidelines is established using the SQL text and other information that is needed to unambiguously identify an SQL statement.

The Db2 optimizer is one of the most sophisticated cost-based optimizers in the industry. However, in rare cases the optimizer might select a less than optimal execution plan. As a DBA familiar with the database, you can use utilities such as **db2advis,** runstats, and **db2expln**, as well as the optimization class setting to help you tune the optimizer for better database performance. If you do not receive expected results after all tuning options have been exhausted, you can provide explicit optimization guidelines to the Db2 optimizer.

An optimization profile is an XML document that can contain optimization guidelines for one or more SQL statements. The correspondence between each SQL statement and its associated optimization guidelines is established using the SQL text and other information that is needed to unambiguously identify an SQL statement.

For example, suppose that even after you had updated the database statistics and performed all other tuning steps, the optimizer still did not choose the I_SUPPKEY index to access the SUPPLIERS table in the following subquery:

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
FROM PARTS P, SUPPLIERS S, PARTSUPP PS
WHERE P_PARTKEY = PS.PS_PARTKEY
  AND S.S_SUPPKEY = PS.PS_SUPPKEY
  AND P.P_SIZE = 39
  AND P.P_TYPE = 'BRASS'
  AND S.S_NATION = 'MOROCCO'
  AND S.S_NATION IN ('MOROCCO', 'SPAIN')
  AND PS.PS_SUPPLYCOST = (SELECT MIN(PS1.PS_SUPPLYCOST)
                          FROM PARTSUPP PS1, SUPPLIERS S1
                          WHERE P.P_PARTKEY = PS1.PS_PARTKEY
                            AND S1.S_SUPPKEY = PS1.PS_SUPPKEY
                            AND S1.S_NATION = S.S_NATION))
```

In this case, an explicit optimization guideline can be used to influence the optimizer. For example:

```
<OPTGUIDELINES><IXSCAN TABLE="S" INDEX="I_SUPPKEY"/></OPTGUIDELINES>
```

Optimization guidelines are specified using a simple XML specification. Each element within the OPTGUIDELINES element is interpreted as an optimization guideline by the Db2 optimizer. There is one optimization guideline element in this example. The IXSCAN element requests that the optimizer use index access. The TABLE attribute of the IXSCAN element indicates the target table reference (using the exposed name of the table reference) and the INDEX attribute specifies the index.

The following example is based on the previous query, and shows how an optimization guideline can be passed to the Db2 optimizer using an optimization profile.

```
<?xml version="1.0" encoding="UTF-8"?>

<OPTPROFILE VERSION="9.1.0.0">
<STMTPROFILE ID="Guidelines for SAMP Q9">
  <STMTKEY SCHEMA="SAMP">
    SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
    FROM PARTS P, SUPPLIERS S, PARTSUPP PS
    WHERE P_PARTKEY = PS.PS_PARTKEY
      AND S.S_SUPPKEY = PS.PS_SUPPKEY
      AND P.P_SIZE = 39
      AND P.P_TYPE = 'BRASS'
      AND S.S_NATION = 'MOROCCO'
      AND S.S_NATION IN ('MOROCCO', 'SPAIN')
      AND PS.PS_SUPPLYCOST = (SELECT MIN(PS1.PS_SUPPLYCOST)
                              FROM PARTSUPP PS1, SUPPLIERS S1
                              WHERE P.P_PARTKEY = PS1.PS_PARTKEY
                                AND S1.S_SUPPKEY = PS1.PS_SUPPKEY
                                AND S1.S_NATION = S.S_NATION))
  </STMTKEY>
  <OPTGUIDELINES><IXSCAN TABLE="S" INDEX="I_SUPPKEY"/></OPTGUIDELINES>
</STMTPROFILE>
</OPTPROFILE>
```

Each STMTPROFILE element provides a set of optimization guidelines for one application statement. The targeted statement is identified by the STMTKEY subelement. The optimization profile is then given a schema-qualified name and inserted into the database. The optimization profile is put into effect for the statement by specifying this name on the **BIND** or **PRECOMPILE** command.

Optimization profiles allow optimization guidelines to be provided to the optimizer without application or database configuration changes. You simply compose the simple XML document, insert it into the database, and specify the name of the optimization profile on the **BIND** or **PRECOMPILE** command. The optimizer automatically matches optimization guidelines to the appropriate statement.

Optimization guidelines do not need to be comprehensive, but should be targeted to a desired execution plan. The Db2 optimizer still considers other possible access plans using the existing cost-based methods. Optimization guidelines targeting specific table references cannot override general optimization settings. For example, an optimization guideline specifying the merge join between tables A and B is not valid at optimization class 0.

You can also specify optimization guidelines through embedded optimization guidelines at the end of SQL statements. For example:

```
/*<OPTGUIDELINES><IXSCAN TABLE="S" INDEX="I_SUPPKEY"/></OPTGUIDELINES>*/
```

This method of specifying optimization guidelines requires no additional configuration.

The optimizer ignores invalid or inapplicable optimization guidelines. If any optimization guidelines are ignored, an execution plan is created and SQL0437W

with reason code 13 is returned. You can then use the EXPLAIN statement to get detailed diagnostic information regarding optimization guidelines processing.

Access plan and query optimization profiles:

Optimization profiles store optimization guidelines that you create to control the decisions made by the optimization stage of the query compiler.

Anatomy of an optimization profile:

An optimization profile can contain global optimization guidelines, and it can contain specific optimization guidelines that apply to individual DML statements in a package. Global optimization guidelines apply to all data manipulation language (DML) statements that are executed while the profile is in effect.

For example:

- You could write a global optimization guideline requesting that the optimizer refer to the materialized query tables (MQTs) Test.SumSales and Test.AvgSales whenever a statement is processed while the current optimization profile is active.
- You could write a statement-level optimization guideline requesting that the optimizer use the I_SUPPKEY index to access the SUPPLIERS table whenever the optimizer encounters the specified statement.

You can specify these two types of guidelines in the two major sections of an optimization profile:

- The global optimization guidelines section can contain one OPTGUIDELINES element
- The statement profiles section that can contain any number of STMTPROFILE elements

An optimization profile must also contain an OPTPROFILE element, which includes metadata and processing directives.

The following code is an example of a valid optimization profile. The optimization profile contains a global optimization guidelines section and a statement profile section with one STMTPROFILE element.

```
<?xml version="1.0" encoding="UTF-8"?>
<OPTPROFILE>

  <!--
       Global optimization guidelines section.
       Optional but at most one.
  -->
  <OPTGUIDELINES>
    <MQT NAME="Test.AvgSales"/>
    <MQT NAME="Test.SumSales"/>
  </OPTGUIDELINES>

  <!--
       Statement profile section.
       Zero or more.
  -->
  <STMTPROFILE ID="Guidelines for SAMP Q9">
    <STMTKEY SCHEMA="SAMP">
      <![CDATA[SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE,
S.S_COMMENT FROM PARTS P, SUPPLIERS S, PARTSUPP PS
WHERE P_PARTKEY = PS.PS_PARTKEY AND S.S_SUPPKEY = PS.PS_SUPPKEY
AND P.P_SIZE = 39 AND P.P_TYPE = 'BRASS'
```

```
AND S.S_NATION = 'MOROCCO' AND S.S_NATION IN ('MOROCCO', 'SPAIN')
AND PS.PS_SUPPLYCOST = (SELECT MIN(PS1.PS_SUPPLYCOST)
FROM PARTSUPP PS1, SUPPLIERS S1
WHERE P.P_PARTKEY = PS1.PS_PARTKEY AND S1.S_SUPPKEY = PS1.PS_SUPPKEY
AND S1.S_NATION = S.S_NATION)]]>
    </STMTKEY>
    <OPTGUIDELINES>
      <IXSCAN TABID="Q1" INDEX="I_SUPPKEY"/>
    </OPTGUIDELINES>
  </STMTPROFILE>

</OPTPROFILE>
```

The OPTPROFILE element

An optimization profile begins with the OPTPROFILE element. In the preceding example, this element consists of a VERSION attribute specifying that the optimization profile version is 9.1.

The global optimization guidelines section

Global optimization guidelines apply to all statements for which the optimization profile is in effect. The global optimization guidelines section is defined in the global OPTGUIDELINES element. In the preceding example, this section contains a single global optimization guideline telling the optimizer to consider the MQTs Test.AvgSales and Test.SumSales when processing any statements for which the optimization profile is in effect.

The statement profile section

A statement profile defines optimization guidelines that apply to a specific statement. There can be zero or more statement profiles in an optimization profile. The statement profile section is defined in the STMTPROFILE element. In the preceding example, this section contains guidelines for a specific statement for which the optimization profile is in effect.

Each statement profile contains a statement key and statement-level optimization guidelines, represented by the STMTKEY and OPTGUIDELINES elements, respectively:

- The statement key identifies the statement to which the statement-level optimization guidelines apply. In the example, the STMTKEY element contains the original statement text and other information that is needed to unambiguously identify the statement. Using the statement key, the optimizer matches a statement profile with the appropriate statement. This relationship enables you to provide optimization guidelines for a statement without having to modify the application.
- The statement-level optimization guidelines section of the statement profile is represented by the OPTGUIDELINES element. This section is made up of one or more access or join requests, which specify methods for accessing or joining tables in the statement. After a successful match with the statement key in a statement profile, the optimizer refers to the associated statement-level optimization guidelines when optimizing the statement. The example contains one access request, which specifies that the SUPPLIERS table referenced in the nested subselect use an index named I_SUPPKEY.

Elements common to both the global optimization guidelines and statement profile sections

Other than the OPTGUIDELINES element the REGISTRY and STMTMATCH element are other elements available for both of these sections:

- The REGISTRY element can set certain registry variables at either the statement or global level. The REGISTRY element is nested in the OPTGUIDELINES element.

  The REGISTRY element contains an OPTION element. The OPTION element has NAME and VALUE attributes which are used to set the value of the named registry variable.

  If you specify a value for a registry variable at the global level, that value applies to all the statements in the connection on which the profile is applied. If you specify a value for a registry variable at the statement level, that value applies only to that statement within the STMTKEY. This value at the statement level takes precedence over the value at the global level.
- The STMTMATCH element sets the matching type used when the compiling statement is matched to the statements in the optimization profile.

  The STMTMATCH element has an EXACT attribute which can be set to either TRUE or FALSE. The default value of STMTMATCH EXACT is TRUE.

  If STMTMATCH EXACT is set to TRUE, exact matching is applied. If STMTMATCH EXACT is set to FALSE, inexact matching is applied.

Creating an optimization profile:

An optimization profile is an XML document that contains optimization guidelines for one or more Data Manipulation Language (DML) statements.

About this task

Because an optimization profile can contain many combinations of guidelines, the following information specifies only those steps that are common to creating any optimization profile.

Procedure

To create an optimization profile:

1. Launch an XML editor. If possible, use one that has schema validation capability. The optimizer does not perform XML validation. An optimization profile must be valid according to the current optimization profile schema.
2. Create an XML document by using a name that makes sense to you. You might want to give it a name that describes the scope of statements to which it applies. For example: `inventory_db.xml`
3. Add the XML declaration to the document. If you do not specify an encoding format, UTF-8 is assumed. Save the document with UTF-16 encoding, if possible. The data server is more efficient when processing this encoding.

   ```
   <?xml version="1.0" encoding="UTF-16"?>
   ```
4. Add an optimization profile section to the document.

   ```
   <OPTPROFILE>
   </OPTPROFILE>
   ```
5. Within the OPTPROFILE element, create global or statement-level optimization guidelines, as appropriate, and save the file.

What to do next

After you have the XML document created, configure the data server to use the optimization profile by inserting the optimization profile into the SYSTOOLS.OPT_PROFILE table.

SQL compiler registry variables in an optimization profile:

Optimization profiles can have different registry variable values applied to a specific query statement or to many query statements used in an application.

Setting registry variables in an optimization profile can increase the flexibility you have in using different query statements for different applications. When you use the **db2set** command to set registry variables, the registry variable values are applied to the entire instance. In optimization profiles, the registry variable values apply only to the statements specified in the optimization profile. By setting registry variables in an optimization profile, you can tailor specific statements for applications without worrying about the registry variable settings of other query statements.

Only a subset of registry variables can be set in an optimization profile. The following registry variables can be set in an optimization profile:

- **DB2_ANTIJOIN**
- **DB2_EXTENDED_OPTIMIZATION** (Only the `ON`, `OFF`, and `IXOR` values are supported)
- **DB2_INLIST_TO_NLJN**
- **DB2_MINIMIZE_LISTPREFETCH**
- **DB2_NEW_CORR_SQ_FF**
- **DB2_OPT_MAX_TEMP_SIZE**
- **DB2_REDUCED_OPTIMIZATION**
- **DB2_RESOLVE_CALL_CONFLICT**
- **DB2_SELECTIVITY**
- **DB2_SELUDI_COMM_BUFFER**
- **DB2_SORT_AFTER_TQ**

Registry variables can be set at both the global level and statement level. If the registry variable is set at the global level, it uses the registry variable settings for all the statements in the optimization profile. If the registry variable is set at the statement level the setting for that registry variable applies only to that specific statement. If the same registry variable is set at both the global and statement levels, the registry variable value at the statement level takes precedence.

Syntax for setting registry variables

Each registry variable is defined and set in an OPTION XML element, with a NAME and a VALUE attribute, all of which are nested in a REGISTRY element. For example:

```
<REGISTRY>
     <OPTION NAME='DB2_SELECTIVITY'  VALUE='YES'/>
     <OPTION NAME='DB2_REDUCED_OPTIMIZATION'  VALUE='NO'/>
</REGISTRY>
```

To have OPTION elements apply to all statements in the application that uses this profile, include the REGISTRY and OPTION elements in the global OPTGUIDELINES element.

To have OPTION elements apply to just a specific SQL statement, include the REGISTRY and OPTION elements in the applicable statement-level STMTPROFILE element. Different STMTPROFILE elements can have different OPTION element settings.

The following example shows registry variable settings at the application and statement level:

```
<?xml version="1.0" encoding="UTF-8"?>
<OPTPROFILE>

   <!--Global section -->
   <OPTGUIDELINES>
      <!-- New: registry variables that apply to all SQL statements that
           use this profile -->
      <REGISTRY>
         <OPTION NAME='DB2_SELECTIVITY'  VALUE='YES'/>
         <OPTION NAME='DB2_REDUCED_OPTIMIZATION'  VALUE='NO'/>
      </REGISTRY>
   </OPTGUIDELINES>

  <!-- First statement level profile -->
  <STMTPROFILE ID='S1'>
      <STMTKEY>
         <![CDATA[select t1.c1, count(*) from t1,t2 where t1.c1 = t2.c1
                  group by t1.c1]]>
      </STMTKEY>
      <OPTGUIDELINES>
         <!-- New: registry variables that JUST applies to the above
              SQL statement when using this profile -->
         <REGISTRY>
            <OPTION  NAME='DB2_REDUCED_OPTIMIZATION'  VALUE='NO_SORT_NLJOIN'/>
         </REGISTRY>
         <NLJOIN>
            <TBSCAN TABLE='T1'/>
            <TBSCAN TABLE='T2'/>
         </NLJOIN>
      </OPTGUIDELINES>
  </STMTPROFILE>

  <!-- Second statement level profile -->
  <STMTPROFILE ID='S2'>
      <STMTKEY><![CDATA[select * from T1 where c1 in( 10,20)]]></STMTKEY>
      <OPTGUIDELINES>
         <!-- New: registry variables that JUST applies to the above
              SQL statement when using this profile -->
         <REGISTRY>
            <OPTION  NAME='DB2_REDUCED_OPTIMIZATION'  VALUE='YES'/>
         </REGISTRY>
      </OPTGUIDELINES>
  </STMTPROFILE>

</OPTPROFILE>
```

Order of precedence

The example sets the same registry variables in multiple places. In addition to these settings, the **db2set** command can also have been used to set registry variables. Registry variables are applied in the following order of precedence, from highest to lowest:

1. Statement level optimization profile settings, which are defined in a statement-level optimization guideline.
2. Overall optimization profile settings, which are defined in a global optimization guideline.

3. Registry variables set by the **db2set** command.

Explain facility

The explain facility captures information about all SQL compiler registry variables that affect data manipulation language (DML) compilation and writes this information to explain tables.

You can use the Explain facility to determine which registry variables are being used for a particular statement. Activate the Explain facility and run the query. Two strings in the ENVVAR sections indicate where the setting has come from:

- `[Global Optimization Guideline]`
- `[Statement-level Optimization Guideline]`

For example:

```
ENVVAR : (Environment Variable)
    DB2_EXTENDED_OPTIMIZATION = ON

ENVVAR : (Environment Variable)
    DB2_EXTENDED_OPTIMIZATION = ON [Global Optimization  Guideline]

ENVVAR : (Environment Variable)
    DB2_EXTENDED_OPTIMIZATION = ON [Statement-level Optimization Guideline]
```

If the registry variables are set in different places, the registry variable with the highest precedence is the only one displayed in the explain output.

Configuring the data server to use an optimization profile:

After an optimization profile is created and its contents are validated against the current optimization profile schema (COPS), the contents must be associated with a unique schema-qualified name and stored in the `SYSTOOLS.OPT_PROFILE` table.

Procedure

To configure the data server to use an optimization profile:

1. Create the optimization profile table (`systools.opt_profile`). Each row of the optimization profile table can contain one optimization profile: the SCHEMA and NAME columns identify the optimization profile, and the PROFILE column contains the text of the optimization profile. The following example calls the `SYSINSTALLOBJECTS` procedure to create the optimization profile table:

   ```
   call sysinstallobjects('opt_profiles','c','','')
   ```

2. Optional: You can grant any authority or privilege on the `systools.opt_profile` table that satisfies your database security requirements. Granting authority or privilege on the `systools.opt_profile` table has no effect on the optimizer's ability to read the table.
3. Create an input data file that contains the three comma-separated string values that are enclosed in double quotation marks. The first string value is the profile schema name. The second string value is the profile name. The third string value is the optimization profile file name. For example, you can create an input data file named `PROFILEDATA` that contains the following three string values:

   ```
   "DBUSER", "PROFILE1", "inventory_db.xml"
   ```

4. Populate the `SYSTOOLS.OPT_PROFILE` table with the optimization profile. The following **IMPORT** command example populates the `SYSTOOLS.OPT_PROFILE` table

with the PROFILEDATA input data file, which contains the profile schema name, profile name and the optimization profile name.

```
db2 import from profiledata of del modified by lobsinfile insert into systools.opt_profile
```

5. Enable the optimization profile with the CURRENT OPTIMIZATION PROFILE special register. For example, you can incorporate SET CURRENT OPTIMIZATION PROFILE statement in your application:

```
stmt.execute( "set current optimization profile = DBUSER.PROFILE1");
```

STMTKEY field in optimization profiles:

Within a STMTPROFILE, the targeted statement is identified by the STMTKEY subelement. The statement that is defined in the STMTKEY field must match exactly to the statement being run by the application, allowing Db2 to unambiguously identify the targeted statement. However, 'white space' within the statement is tolerated.

After Db2 finds a statement key that matches the current compilation key, it stops looking. Therefore, if there were multiple statement profiles in an optimization profile whose statement key matches the current compilation key, only the first such statement profile is used. Statement file precedence is based on document order. No error or warning is issued in this case.

The statement key matches the statement ⌂select * from orders where *foo(orderkey)*>20," provided the compilation key has a default schema of ⌂COLLEGE⌂ and a function path of SYSIBM,SYSFUN,SYSPROC,DAVE.

```
<STMTKEY SCHEMA='COLLEGE' FUNCPATH='SYSIBM,SYSFUN,SYSPROC,DAVE'>
<![CDATA[select * from orders where foo(orderkey)>20]]>
</stmtkey>
```

Specifying which optimization profile the optimizer is to use:

Use the **OPTPROFILE** bind option to specify that an optimization profile is to be used at the package level, or use the CURRENT OPTIMIZATION PROFILE special register to specify that an optimization profile is to be used at the statement level.

This special register contains the qualified name of the optimization profile used by statements that are dynamically prepared for optimization. For CLI applications, you can use the **CURRENTOPTIMIZATIONPROFILE** client configuration option to set this special register for each connection.

The **OPTPROFILE** bind option setting also specifies the default optimization profile for the CURRENT OPTIMIZATION PROFILE special register. The order of precedence for defaults is as follows:

- The **OPTPROFILE** bind option applies to all static statements, regardless of any other settings.
- For dynamic statements, the value of the CURRENT OPTIMIZATION PROFILE special register is determined by the following, in order of lowest to highest precedence:
  - The **OPTPROFILE** bind option
  - The **CURRENTOPTIMIZATIONPROFILE** client configuration option
  - The most recent SET CURRENT OPTIMIZATION PROFILE statement in the application

Setting an optimization profile within an application:

You can control the setting of the current optimization profile for dynamic statements in an application by using the SET CURRENT OPTIMIZATION PROFILE statement.

About this task

The optimization profile name that you provide in the statement must be a schema-qualified name. If you do not provide a schema name, the value of the CURRENT SCHEMA special register is used as the implicit schema qualifier.

The optimization profile that you specify applies to all subsequent dynamic statements until another SET CURRENT OPTIMIZATION PROFILE statement is encountered. Static statements are not affected, because they are preprocessed and packaged before this setting is evaluated.

Procedure

To set an optimization profile within an application:

- Use the SET CURRENT OPTIMIZATION PROFILE statement anywhere within your application. For example, the last statement in the following sequence is optimized according to the JON.SALES optimization profile.

```
EXEC SQL  SET CURRENT OPTIMIZATION PROFILE = 'NEWTON.INVENTDB';

/* The following statements are both optimized with 'NEWTON.INVENTDB' */
EXEC SQL PREPARE stmt FROM SELECT ... ;
EXEC SQL EXECUTE stmt;

EXEC SQL EXECUTE IMMEDIATE SELECT ... ;

EXEC SQL SET CURRENT OPTIMIZATION PROFILE = 'JON.SALES';

/* This statement is optimized with 'JON.SALES' */
EXEC SQL EXECUTE IMMEDIATE SELECT ... ;
```

- If you want the optimizer to use the default optimization profile that was in effect when the application started running, specify the null value. For example:

```
EXEC SQL SET CURRENT OPTIMIZATION PROFILE = NULL;
```

- If you don't want the optimizer to use optimization profiles, specify the empty string. For example:

```
EXEC SQL SET CURRENT OPTIMIZATION PROFILE = '';
```

- If you are using a call level interface (CLI) application, you can add the CURRENTOPTIMIZATIONPROFILE parameter to the `db2cli.ini` file, using the **`UPDATE CLI CONFIGURATION`** command. For example:

```
update cli cfg for section sanfran using currentoptimizationprofile jon.sales
```

  This results in the following entry in the `db2cli.ini` file:

```
[SANFRAN]
CURRENTOPTIMIZATIONPROFILE=JON.SALES
```

  Note: Any SET CURRENT OPTIMIZATION PROFILE statements in the application override this setting.

Binding an optimization profile to a package:

When you prepare a package by using the **`BIND`** or **`PRECOMPILE`** command, you can use the **`OPTPROFILE`** command parameter to specify the optimization profile for the package.

About this task

This method is the only way to apply an optimization profile to static statements, and the specified profile applies to all static statements in the package. An optimization profile that is specified in this manner is also the default optimization profile that is used for dynamic statements within the package.

Procedure

You can bind an optimization profile in SQLJ or embedded SQL using APIs (for example, sqlaprep) or the command line processor (CLP).
For example, the following code shows how to bind an inventory database optimization profile to an inventory application from the CLP:

```
db2 prep inventapp.sqc bindfile optprofile newton.inventdb
db2 bind inventapp.bnd
db2 connect reset
db2 terminate
xlc -I$HOME/sqllib/include -c inventapp.c -o inventapp.o
xlc -o inventapp inventapp.o -ldb2 -L$HOME/sqllib/lib
```

If you do not specify a schema name for the optimization profile, the **QUALIFIER** command parameter is used as the implicit qualifier.

Modifying an optimization profile:

When you make a change to an optimization profile there are certain steps that need to be taken in order for the changes in the optimization profiles to take effect.

About this task

When an optimization profile is referenced, it is compiled and cached in memory; therefore, these references must also be removed. Use the FLUSH OPTIMIZATION PROFILE CACHE statement to remove the old profile from the optimization profile cache. The statement also invalidates any statement in the dynamic plan cache that was prepared by using the old profile (*logical invalidation*).

Procedure

To modify an optimization profile:

1. Edit the optimization profile XML file on disk, make the necessary changes, save the file to disk.
2. Validate that the changes made to the file are well formed XML as defined in the current optimization profile schema (COPS) in the DB2OptProfile.xsd file, which is located in the misc subdirectory of the sqllib directory.
3. Update the existing row in the SYSTOOLS.OPT_PROFILE table with the new profile.
4. Ensure the new optimization profile is used:
   - If you did not create triggers to flush the optimization profile cache, issue the FLUSH OPTIMIZATION PROFILE CACHE statement. The statement removes any versions of the optimization profile that might be contained in the optimization profile cache.

     When you flush the optimization profile cache, any dynamic statements that were prepared with the old optimization profile are also invalidated in the dynamic plan cache.

- If you have bound an optimization profile to a package of static statements, then you will need to re-bind the package, using the OPTPROFILE command parameter again to specify the modified optimization profile.

Results

Any subsequent reference to the optimization profile causes the optimizer to read the new profile and to reload it into the optimization profile cache. Statements prepared under the old optimization profile are logically invalidated. Calls made to those statements are prepared under the new optimization profile and recached in the dynamic plan cache.

Deleting an optimization profile:

You can remove an optimization profile that is no longer needed by deleting it from the SYSTOOLS.OPT_PROFILE table. When an optimization profile is referenced, it is compiled and cached in memory; therefore, if the original profile has already been used, you must also flush the deleted optimization profile from the optimization profile cache.

Procedure

To delete an optimization profile:

1. Delete the optimization profile from the SYSTOOLS.OPT_PROFILE table. For example:

```
delete from systools.opt_profile
  where schema = 'NEWTON' and name = 'INVENTDB'
```

2. If you did not create triggers to flush the optimization profile cache, issue the FLUSH OPTIMIZATION PROFILE CACHE statement to remove any versions of the optimization profile that might be contained in the optimization profile cache. See “Triggers to flush the optimization profile cache” on page 429.

   Note: When you flush the optimization profile cache, any dynamic statements that were prepared with the old optimization profile are also invalidated in the dynamic plan cache.

Results

Any subsequent reference to the optimization profile causes the optimizer to return SQL0437W with reason code 13.

Access plan and query optimization guidelines:

Optimization guidelines are rules that you create to control the decisions made during the optimization stages of query compilation. These guidelines are recorded and used by an optimization profile. Guidelines can be written at the global level or statement level.

Many query compiler decisions can be controlled through the creation of optimization guidelines.

Types of optimization guidelines:

The Db2 optimizer processes a statement in two phases: the query rewrite optimization phase and the plan optimization phase.

The optimized statement is determined by the *query rewrite optimization phase*, which transforms the original statement into a semantically equivalent statement that can be more easily optimized in the plan optimization phase. The *plan optimization phase* determines the optimal access methods, join methods, and join orders for the *optimized statement* by enumerating a number of alternatives and choosing the alternative that minimizes an execution cost estimate.

The query transformations, access methods, join methods, join orders, and other optimization alternatives that are considered during the two optimization phases are governed by various Db2 parameters, such as the CURRENT QUERY OPTIMIZATION special register, the **REOPT** bind option, and the **DB2_REDUCED_OPTIMIZATION** registry variable. The set of optimization alternatives is known as the *search space*.

The following types of statement optimization guidelines are supported:

- *General optimization guidelines*, which can be used to affect the setting of general optimization parameters, are applied first, because they can affect the search space.
- *Query rewrite guidelines*, which can be used to affect the transformations that are considered during the query rewrite optimization phase, are applied next, because they can affect the statement that is optimized during the plan optimization phase.
- *Plan optimization guidelines*, which can be used to affect the access methods, join methods, and join orders that are considered during the plan optimization phase, are applied last.

General optimization guidelines:

General optimization guidelines can be used to set general optimization parameters.

Each of these guidelines has statement-level scope.

Query rewrite optimization guidelines:

Query rewrite guidelines can be used to affect the transformations that are considered during the query rewrite optimization phase, which transforms the original statement into a semantically equivalent optimized statement.

Query rewrite guidelines can be used to affect the transformations that are considered during the query rewrite optimization phase, which transforms the original statement into a semantically equivalent optimized statement.

The optimal execution plan for the optimized statement is determined during the plan optimization phase. Consequently, query rewrite optimization guidelines can affect the applicability of plan optimization guidelines.

Each query rewrite optimization guideline corresponds to one of the optimizer's query transformation rules. The following query transformation rules can be affected by query rewrite optimization guidelines:

- IN-LIST-to-join
- Subquery-to-join
- NOT-EXISTS-subquery-to-antijoin
- NOT-IN-subquery-to-antijoin

Query rewrite optimization guidelines are not always applicable. Query rewrite rules are enforced one at a time. Consequently, query rewrite rules that are enforced before a subsequent rule can affect the query rewrite optimization guideline that is associated with that rule. The environment configuration can affect the behavior of some rewrite rules which, in turn, affects the applicability of a query rewrite optimization guideline to a specific rule.

To get the same results each time, query rewrite rules have certain conditions that are applied before the rules are enforced. If the conditions that are associated with a rule are not satisfied when the query rewrite component attempts to apply the rule to the query, the query rewrite optimization guideline for the rule is ignored. If the query rewrite optimization guideline is not applicable, and the guideline is an enabling guideline, SQL0437W with reason code 13 is returned. If the query rewrite optimization guideline is not applicable, and the guideline is a disabling guideline, no message is returned. The query rewrite rule is not applied in this case, because the rule is treated as if it were disabled.

Query rewrite optimization guidelines can be divided into two categories: statement-level guidelines and predicate-level guidelines. All of the query rewrite optimization guidelines support the statement-level category. Only INLIST2JOIN supports the predicate-level category. The statement-level query rewrite optimization guidelines apply to the entire query. The predicate-level query rewrite optimization guideline applies to the specific predicate only. If both statement-level and predicate-level query rewrite optimization guidelines are specified, the predicate-level guideline overrides the statement-level guideline for the specific predicate.

Each query rewrite optimization guideline is represented by a corresponding rewrite request element in the optimization guideline schema.

The following example illustrates an IN-LIST-to-join query rewrite optimization guideline, as represented by the INLIST2JOIN rewrite request element.

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
FROM "Samp".PARTS P, "Samp".SUPPLIERS S, "Samp".PARTSUPP PS
WHERE P_PARTKEY = PS.PS_PARTKEY
  AND S.S_SUPPKEY = PS.PS_SUPPKEY
  AND P_SIZE IN (35, 36, 39, 40)
  AND S.S_NATION IN ('INDIA', 'SPAIN')
  AND PS.PS_SUPPLYCOST = (SELECT MIN(PS1.PS_SUPPLYCOST)
                          FROM "Samp".PARTSUPP PS1, "Samp".SUPPLIERS S1
                          WHERE P.P_PARTKEY = PS1.PS_PARTKEY
                            AND S1.S_SUPPKEY = PS1.PS_SUPPKEY
                            AND S1.S_NATION = S.S_NATION)
ORDER BY S.S_NAME
<OPTGUIDELINES><INLIST2JOIN TABLE='P'/></OPTGUIDELINES>;
```

This particular query rewrite optimization guideline specifies that the list of constants in the predicate `P_SIZE IN (35, 36, 39, 40)` should be transformed into a table expression. This table expression would then be eligible to drive an indexed nested-loop join access to the PARTS table in the main subselect. The TABLE attribute is used to identify the target IN-LIST predicate by indicating the table reference to which this predicate applies. If there are multiple IN-LIST predicates for the identified table reference, the INLIST2JOIN rewrite request element is considered ambiguous and is ignored.

In such cases, a COLUMN attribute can be added to further qualify the target IN-LIST predicate. For example:

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
FROM "Samp".PARTS P, "Samp".SUPPLIERS S, "Samp".PARTSUPP PS
WHERE P_PARTKEY = PS.PS_PARTKEY
  AND S.S_SUPPKEY = PS.PS_SUPPKEY
  AND P_SIZE IN (35, 36, 39, 40)
  AND P_TYPE IN ('BRASS', 'COPPER')
  AND S.S_NATION IN ('INDIA', 'SPAIN')
  AND PS.PS_SUPPLYCOST = (SELECT MIN(PS1.PS_SUPPLYCOST)
                          FROM "Samp".PARTSUPP PS1, "Samp".SUPPLIERS S1
                          WHERE P.P_PARTKEY = PS1.PS_PARTKEY
                            AND S1.S_SUPPKEY = PS1.PS_SUPPKEY
                            AND S1.S_NATION = S.S_NATION)
ORDER BY S.S_NAME
<OPTGUIDELINES><INLIST2JOIN TABLE='P' COLUMN='P_SIZE'/></OPTGUIDELINES>;
```

The TABLE attribute of the INLIST2JOIN element identifies the PARTS table reference in the main subselect. The COLUMN attribute is used to identify the IN-LIST predicate on the P_SIZE column as the target. In general, the value of the COLUMN attribute can contain the unqualified name of the column referenced in the target IN-LIST predicate. If the COLUMN attribute is provided without the TABLE attribute, the query rewrite optimization guideline is considered invalid and is ignored.

The OPTION attribute can be used to enable or disable a particular query rewrite optimization guideline. Because the OPTION attribute is set to DISABLE in the following example, the list of constants in the predicate `P_SIZE IN (35, 36, 39, 40)` will not be transformed into a table expression. The default value of the OPTION attribute is ENABLE. ENABLE and DISABLE must be specified in uppercase characters.

```
<OPTGUIDELINES>
  <INLIST2JOIN TABLE='P' COLUMN='P_SIZE' OPTION='DISABLE'/>
</OPTGUIDELINES>
```

In the following example, the INLIST2JOIN rewrite request element does not have a TABLE attribute. The optimizer interprets this as a request to disable the IN-LIST-to-join query transformation for all IN-LIST predicates in the statement.

```
<OPTGUIDELINES><INLIST2JOIN OPTION='DISABLE'/></OPTGUIDELINES>
```

The following example illustrates a subquery-to-join query rewrite optimization guideline, as represented by the SUBQ2JOIN rewrite request element. A subquery-to-join transformation converts a subquery into an equivalent table expression. The transformation applies to subquery predicates that are quantified by EXISTS, IN, =SOME, =ANY, <>SOME, or <>ANY. The subquery-to-join query rewrite optimization guideline does not ensure that a subquery will be merged. A particular subquery cannot be targeted by this query rewrite optimization guideline. The transformation can only be enabled or disabled at the statement level.

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
FROM "Samp".PARTS P, "Samp".SUPPLIERS S, "Samp".PARTSUPP PS
WHERE P_PARTKEY = PS.PS_PARTKEY
  AND S.S_SUPPKEY = PS.PS_SUPPKEY
  AND P_SIZE IN (35, 36, 39, 40)
  AND P_TYPE = 'BRASS'
  AND S.S_NATION IN ('INDIA', 'SPAIN')
  AND PS.PS_SUPPLYCOST = (SELECT MIN(PS1.PS_SUPPLYCOST)
                          FROM "Samp".PARTSUPP PS1, "Samp".SUPPLIERS S1
                          WHERE P.P_PARTKEY = PS1.PS_PARTKEY
                            AND S1.S_SUPPKEY = PS1.PS_SUPPKEY
```

```
                              AND S1.S_NATION = S.S_NATION)
ORDER BY S.S_NAME
<OPTGUIDELINES><SUBQ2JOIN OPTION='DISABLE'/></OPTGUIDELINES>;
```

The following example illustrates a NOT-EXISTS-to-anti-join query rewrite optimization guideline, as represented by the NOTEX2AJ rewrite request element. A NOT-EXISTS-to-anti-join transformation converts a subquery into a table expression that is joined to other tables using anti-join semantics (only nonmatching rows are returned). The NOT-EXISTS-to-anti-join query rewrite optimization guideline applies to subquery predicates that are quantified by NOT EXISTS. The NOT-EXISTS-to-anti-join query rewrite optimization guideline does not ensure that a subquery will be merged. A particular subquery cannot be targeted by this query rewrite optimization guideline. The transformation can only be enabled or disabled at the statement level.

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
FROM "Samp".PARTS P, "Samp".SUPPLIERS S, "Samp".PARTSUPP PS
WHERE P_PARTKEY = PS.PS_PARTKEY
  AND S.S_SUPPKEY = PS.PS_SUPPKEY
  AND P_SIZE IN (35, 36, 39, 40)
  AND P_TYPE = 'BRASS'
  AND S.S_NATION IN ('INDIA', 'SPAIN')
  AND NOT EXISTS (SELECT 1
                  FROM "Samp".SUPPLIERS S1
                  WHERE S1.S_SUPPKEY = PS.PS_SUPPKEY)
ORDER BY S.S_NAME
<OPTGUIDELINES><NOTEX2AJ OPTION='ENABLE'/></OPTGUIDELINES>;
```

Note: The enablement of a query transformation rule at the statement level does not ensure that the rule will be applied to a particular part of the statement. The usual criteria are used to determine whether query transformation will take place. For example, if there are multiple NOT EXISTS predicates in the query block, the optimizer will not consider converting any of them into anti-joins. Explicitly enabling query transformation at the statement level does not change this behavior.

The following example illustrates a NOT-IN-to-anti-join query rewrite optimization guideline, as represented by the NOTIN2AJ rewrite request element. A NOT-IN-to-anti-join transformation converts a subquery into a table expression that is joined to other tables using anti-join semantics (only nonmatching rows are returned). The NOT-IN-to-anti-join query rewrite optimization guideline applies to subquery predicates that are quantified by NOT IN. The NOT-IN-to-anti-join query rewrite optimization guideline does not ensure that a subquery will be merged. A particular subquery cannot be targeted by this query rewrite optimization guideline. The transformation can only be enabled or disabled at the statement level.

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
FROM "Samp".PARTS P, "Samp".SUPPLIERS S, "Samp".PARTSUPP PS
WHERE P_PARTKEY = PS.PS_PARTKEY
  AND S.S_SUPPKEY = PS.PS_SUPPKEY
  AND P_SIZE IN (35, 36, 39, 40)
  AND P_TYPE = 'BRASS'
  AND S.S_NATION IN ('INDIA', 'SPAIN')
  AND PS.PS_SUPPKEY NOT IN (SELECT S1.S_SUPPKEY
                            FROM "Samp".SUPPLIERS S1
                            WHERE S1.S_NATION = 'CANADA')
ORDER BY S.S_NAME
<OPTGUIDELINES><NOTIN2AJ OPTION='ENABLE'/></OPTGUIDELINES>
```

A particular query rewrite optimization guideline might not be applicable when considered within the context of other query rewrite transformations being applied to the statement. That is, if a guideline request to enable a transform cannot be

applied, a warning is returned. For example, an INLIST2JOIN rewrite enable request element targeting a predicate that is eliminated from the query by another query transformation would not be applicable. Moreover, the successful application of a query rewrite optimization guideline might change the applicability of other query rewrite transformation rules. For example, a request to transform an IN-LIST to a table expression might prevent a different IN-LIST from being transformed to a table expression, because the optimizer will only apply a single IN-LIST-to-join transformation per query block.

Plan optimization guidelines:

Plan optimization guidelines are applied during the cost-based phase of optimization, where access methods, join methods, join order, and other details of the execution plan for the statement are determined.

Plan optimization guidelines are applied during the cost-based phase of optimization, where access methods, join methods, join order, and other details of the execution plan for the statement are determined.

Plan optimization guidelines need not specify all aspects of an execution plan. Unspecified aspects of the execution plan are determined by the optimizer in a cost-based fashion.

There are two categories of plan optimization guidelines:

- accessRequest – An access request specifies an access method for satisfying a table reference in a statement.
- joinRequest – A join request specifies a method and sequence for performing a join operation. Join requests are composed of access or other join requests.

Access request optimization guidelines correspond to the optimizer's data access methods, such as table scan, index scan, and list prefetch. Join request guidelines correspond to the optimizer's join methods, such as nested-loop join, hash join, and merge join. Each access request and join request is represented by a corresponding access request element and join request element in the statement optimization guideline schema.

The following example illustrates an index scan access request, as represented by the IXSCAN access request element. This particular request specifies that the optimizer is to use the I_SUPPKEY index to access the SUPPLIERS table in the main subselect of the statement. The optional INDEX attribute identifies the desired index. The TABLE attribute identifies the table reference to which the access request is applied. A TABLE attribute must identify the target table reference using its exposed name, which in this example is the correlation name S.

```
SQL statement:

  select s.s_name, s.s_address, s.s_phone, s.s_comment
    from "Samp".parts, "Samp".suppliers s, "Samp".partsupp ps
    where p_partkey = ps.ps_partkey and
      s.s_suppkey = ps.ps_suppkey and
      p_size = 39 and
      p_type = 'BRASS' and
      s.s_nation in ('MOROCCO', 'SPAIN') and
      ps.ps_supplycost = (select min(ps1.ps_supplycost)
                            from "Samp".partsupp ps1, "Samp".suppliers s1
                            where "Samp".parts.p_partkey = ps1.ps_partkey and
                            s1.s_suppkey = ps1.ps_suppkey and
                            s1.s_nation = s.s_nation)
    order by s.s_name
```

```
Optimization guideline:

   <OPTGUIDELINES>
     <IXSCAN TABLE='S' INDEX='I_SUPPKEY'/>
   </OPTGUIDELINES>
```

The following index scan access request element specifies that the optimizer is to use index access to the PARTS table in the main subselect of the statement. The optimizer will choose the index in a cost-based fashion, because there is no INDEX attribute. The TABLE attribute uses the qualified table name to refer to the target table reference, because there is no associated correlation name.

```
   <OPTGUIDELINES>
     <IXSCAN TABLE='"Samp".PARTS'/>
   </OPTGUIDELINES>
```

The following list prefetch access request is represented by the LPREFETCH access request element. This particular request specifies that the optimizer is to use the I_SNATION index to access the SUPPLIERS table in the nested subselect of the statement. The TABLE attribute uses the correlation name S1, because that is the exposed name identifying the SUPPLIERS table reference in the nested subselect.

```
   <OPTGUIDELINES>
     <LPREFETCH TABLE='S1' INDEX='I_SNATION'/>
   </OPTGUIDELINES>
```

The following index scan access request element specifies that the optimizer is to use the I_SNAME index to access the SUPPLIERS table in the main subselect. The FIRST attribute specifies that this table is to be the first table that is accessed in the join sequence chosen for the corresponding FROM clause. The FIRST attribute can be added to any access or join request; however, there can be at most one access or join request with the FIRST attribute referring to tables in the same FROM clause.

```
SQL statement:

   select s.s_name, s.s_address, s.s_phone, s.s_comment
     from "Samp".parts, "Samp".suppliers s, "Samp".partsupp ps
     where p_partkey = ps.ps_partkey
       s.s_suppkey = ps.ps_suppkey and
       p_size = 39 and
       p_type = 'BRASS' and
       s.s_nation in ('MOROCCO', 'SPAIN') and
       ps.ps_supplycost = (select min(ps1.ps_supplycost)
                             from "Samp".partsupp ps1, "Samp".suppliers s1
                             where "Samp".parts.p_partkey = ps1.ps_partkey and
                               s1.s_suppkey = ps1.ps_suppkey and
                               s1.s_nation = s.s_nation)
     order by s.s_name
     optimize for 1 row

Optimization guidelines:

   <OPTGUIDELINES>
     <IXSCAN TABLE='S' INDEX='I_SNAME' FIRST='TRUE'/>
   </OPTGUIDELINES>
```

The following example illustrates how multiple access requests are passed in a single statement optimization guideline. The TBSCAN access request element represents a table scan access request. This particular request specifies that the SUPPLIERS table in the nested subselect is to be accessed using a full table scan. The LPREFETCH access request element specifies that the optimizer is to use the I_SUPPKEY index during list prefetch index access to the SUPPLIERS table in the main subselect.

```
<OPTGUIDELINES>
  <TBSCAN TABLE='S1'/>
  <LPREFETCH TABLE='S' INDEX='I_SUPPKEY'/>
</OPTGUIDELINES>
```

The following example illustrates a nested-loop join request, as represented by the NLJOIN join request element. In general, a join request element contains two child elements. The first child element represents the desired outer input to the join operation, and the second child element represents the desired inner input to the join operation. The child elements can be access requests, other join requests, or combinations of access and join requests. In this example, the first IXSCAN access request element specifies that the PARTS table in the main subselect is to be the outer table of the join operation. It also specifies that PARTS table access be performed using an index scan. The second IXSCAN access request element specifies that the PARTSUPP table in the main subselect is to be the inner table of the join operation. It, too, specifies that the table is to be accessed using an index scan.

```
<OPTGUIDELINES>
  <NLJOIN>
    <IXSCAN TABLE='"Samp".Parts'/>
    <IXSCAN TABLE="PS"/>
  </NLJOIN>
</OPTGUIDELINES>
```

The following example illustrates a hash join request, as represented by the HSJOIN join request element. The ACCESS access request element specifies that the SUPPLIERS table in the nested subselect is to be the outer table of the join operation. This access request element is useful in cases where specifying the join order is the primary objective. The IXSCAN access request element specifies that the PARTSUPP table in the nested subselect is to be the inner table of the join operation, and that the optimizer is to choose an index scan to access that table.

```
<OPTGUIDELINES>
  <HSJOIN>
    <ACCESS TABLE='S1'/>
    <IXSCAN TABLE='PS1'/>
  </HSJOIN>
</OPTGUIDELINES>
```

The following example illustrates how larger join requests can be constructed by nesting join requests. The example includes a merge join request, as represented by the MSJOIN join request element. The outer input of the join operation is the result of joining the PARTS and PARTSUPP tables of the main subselect, as represented by the NLJOIN join request element. The inner input of the join request element is the SUPPLIERS table in the main subselect, as represented by the IXSCAN access request element.

```
<OPTGUIDELINES>
  <MSJOIN>
    <NLJOIN>
      <IXSCAN TABLE='"Samp".Parts'/>
      <IXSCAN TABLE="PS"/>
    </NLJOIN>
    <IXSCAN TABLE='S'/>
  </MSJOIN>
</OPTGUIDELINES>
```

If a join request is to be valid, all access request elements that are nested either directly or indirectly inside of it must reference tables in the same FROM clause of the optimized statement.

Optimization guidelines for MQT matching

Users can override the optimizer's decision and force it to choose specific materialized query tables (MQTs) with the MQTENFORCE element. The MQTENFORCE element, can be specified at both the global and statement profile level, is used with one of the following attributes:

NAME
specifies the partial or fully qualified name of MQT to choose

TYPE specifies a group of MQTs by their types. Possible values are:
- NORMAL: all non-replicated MQTs
- REPLICATED: all replicated MQTs
- ALL: all MQTs

The following example illustrates an example of a guideline that enforces all replicated MQTs, as well as, the SAMP.PARTSMQT:

```
<OPTGUIDELINES>
 <MQTENFORCE NAME='SAMP.PARTSMQT'/>
 <MQTENFORCE TYPE='REPLICATED'/>
</OPTGUIDELINES>
```

Note: If you specify more than one attribute at a time, only the first one will be used. So in the following example

```
<MQTENFORCE NAME='SAMP.PARTSMQT' TYPE='REPLICATED'/>
```

Only PARTSMQT MQT will be enforced

Creating statement-level optimization guidelines:

The statement-level optimization guidelines section of the statement profile is made up of one or more access or join requests, which specify methods for accessing or joining tables in the statement.

Before you begin

Exhaust all other tuning options. For example:

1. Ensure that the data distribution statistics have been recently updated by the **RUNSTATS** utility.
2. Ensure that the data server is running with the proper optimization class setting for the workload.
3. Ensure that the optimizer has the appropriate indexes to access tables that are referenced in the query.

Procedure

To create statement-level optimization guidelines:

1. Create the optimization profile in which you want to insert the statement-level guidelines. See “Creating an optimization profile” on page 365.
2. Run the explain facility against the statement to determine whether optimization guidelines would be helpful. Proceed if that appears to be the case.
3. Obtain the *original* statement by running a query that is similar to the following:

```
select statement_text
  from explain_statement
  where explain_level = '0' and
    explain_requester = 'SIMMEN' and
    explain_time      = '2003-09-08-16.01.04.108161' and
    source_name       = 'SQLC2E03' and
    source_version    = '' and
    queryno           = 1
```

4. Edit the optimization profile and create a statement profile, inserting the statement text into the statement key. For example:

```
<STMTPROFILE ID="Guidelines for SAMP Q9">
   <STMTKEY SCHEMA="SAMP"><![CDATA[SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE,
     S.S_COMMENT
     FROM PARTS P, SUPPLIERS S, PARTSUPP PS
     WHERE P_PARTKEY = PS.PS_PARTKEY AND S.S_SUPPKEY = PS.PS_SUPPKEY
     AND P.P_SIZE  = 39 AND P.P_TYPE  = 'BRASS' AND S.S_NATION
     = 'MOROCCO' AND
     PS.PS_SUPPLYCOST = (SELECT MIN(PS1.PS_SUPPLYCOST)
     FROM PARTSUPP PS1, SUPPLIERS S1
     WHERE P.P_PARTKEY  = PS1.PS_PARTKEY AND S1.S_SUPPKEY = PS1.PS_SUPPKEY
     AND S1.S_NATION  = S.S_NATION)]]>
   </STMTKEY>
</STMTPROFILE>
```

5. Insert statement-level optimization guidelines after the statement key. Use exposed names to identify the objects that are referenced in access and join requests. The following is an example of a join request:

```
<OPTGUIDELINES>
  <HSJOIN>
    <TBSCAN TABLE='PS1'/>
    <IXSCAN TABLE='S1'
      INDEX='I1'/>
  </HSJOIN>
</OPTGUIDELINES>
```

6. Validate the file and save it.

Results

If expected results are not achieved, make changes to the guidelines or create additional guidelines, and update the optimization profile, as appropriate.

Forming table references in optimization guidelines:

The term *table reference* is used to mean any table, view, table expression, or the table which an alias references in an SQL statement or view definition. An optimization guideline can identify a table reference using either its exposed name in the original statement or the unique correlation name that is associated with the table reference in the optimized statement.

The term *table reference* is used to mean any table, view, table expression, or the table which an alias references in an SQL statement or view definition. An optimization guideline can identify a table reference using either its exposed name in the original statement or the unique correlation name that is associated with the table reference in the optimized statement.

Extended names, which are sequences of exposed names, help to uniquely identify table references that are embedded in views. An alias name cannot be used as a table reference in an optimization guideline, in such a case any guideline targeting the table reference will be ignored. Optimization guidelines that identify exposed or extended names that are not unique within the context of the entire statement

are considered ambiguous and are not applied. Moreover, if more than one optimization guideline identifies the same table reference, all optimization guidelines identifying that table reference are considered conflicting and are not applied. The potential for query transformations makes it impossible to guarantee that an exposed or extended name will still exist during optimization; therefore, any guideline that identifies such table references is ignored.

Index names are specified by using the INDEX or IXNAME attribute on some optimization guideline elements. Index names must be unqualified.

Using exposed names in the original statement to identify table references

A table reference is identified by using the exposed name of the table. The exposed name is specified in the same way that a table would be qualified in an SQL statement.

The rules for specifying SQL identifiers also apply to the TABLE attribute value of an optimization guideline. The TABLE attribute value is compared to each exposed name in the statement. Only a single match is permitted in this Db2 release. If the TABLE attribute value is schema-qualified, it matches any equivalent exposed qualified table name. If the TABLE attribute value is unqualified, it matches any equivalent correlation name or exposed table name. The TABLE attribute value is therefore considered to be implicitly qualified by the default schema that is in effect for the statement. These concepts are illustrated by the following example. Assume that the statement is optimized using the default schema Samp.

```
select s_name, s_address, s_phone, s_comment
  from parts, suppliers, partsupp ps
  where p_partkey = ps.ps_partkey and
    s.s_suppkey = ps.ps_suppkey and
    p_size = 39 and
    p_type = 'BRASS'
```

TABLE attribute values that identify a table reference in the statement include '"Samp".PARTS', 'PARTS', 'Parts' (because the identifier is not delimited, it is converted to uppercase characters). TABLE attribute values that fail to identify a table reference in the statement include '"Samp2".SUPPLIERS', 'PARTSUPP' (not an exposed name), and 'Samp.PARTS' (the identifier Samp must be delimited; otherwise, it is converted to uppercase characters).

The exposed name can be used to target any table reference in the original statement, view, SQL function, or trigger.

Using exposed names in the original statement to identify table references in views

Optimization guidelines can use extended syntax to identify table references that are embedded in views, as shown in the following example:

```
create view "Rick".v1 as
  (select * from employee a where salary > 50000)

create view "Gustavo".v2 as
  (select * from "Rick".v1
     where deptno in ('52', '53', '54')

select * from "Gustavo".v2 a
  where v2.hire_date > '01/01/2004'
```

```
<OPTGUIDELINES>
  <IXSCAN TABLE='A/"Rick".V1/A'/>
</OPTGUIDELINES>
```

The IXSCAN access request element specifies that an index scan is to be used for the EMPLOYEE table reference that is embedded in the views "Gustavo".V2 and "Rick".V1. The extended syntax for identifying table references in views is a series of exposed names separated by a slash character. The value of the TABLE attribute A/"Rick".V1/A illustrates the extended syntax. The last exposed name in the sequence (A) identifies the table reference that is a target of the optimization guideline. The first exposed name in the sequence (A) identifies the view that is directly referenced in the original statement. The exposed name or names in the middle ("Rick".V1) pertain to the view references along the path from the direct view reference to the target table reference. The rules for referring to exposed names from optimization guidelines, described in the previous section, apply to each step of the extended syntax.

Had the exposed name of the EMPLOYEE table reference in the view been unique with respect to all tables that are referenced either directly or indirectly by the statement, the extended name syntax would not be necessary.

Extended syntax can be used to target any table reference in the original statement, SQL function, or trigger.

Identifying table references using correlation names in the optimized statement

An optimization guideline can also identify a table reference using the unique correlation names that are associated with the table reference in the optimized statement. The optimized statement is a semantically equivalent version of the original statement, as determined during the query rewrite phase of optimization. The optimized statement can be retrieved from the explain tables. The TABID attribute of an optimization guideline is used to identify table references in the optimized statement.

Original statement:

```
select s.s_name, s.s_address, s.s_phone, s.s_comment
  from sm_samp.parts p, sm_samp.suppliers s, sm_samp.partsupp ps
  where p_partkey = ps.ps_partkey and
    s.s_suppkey = ps.ps_suppkey and
    p.p_size = 39 and
    p.p_type = 'BRASS' and
    s.s_nation in ('MOROCCO', 'SPAIN') and
    ps.ps_supplycost = (select min(ps1.ps_supplycost)
                          from sm_samp.partsupp ps1, sm_samp.suppliers s1
                          where p.p_partkey = ps1.ps_partkey and
                            s1.s_suppkey = ps1.ps_suppkey and
                            s1.s_nation = s.s_nation)

<OPTGUIDELINES>
  <HSJOIN>
    <TBSCAN TABLE='S1'/>
    <IXSCAN TABID='Q2'/>
  </HSJOIN>
</OPTGUIDELINES>
```

Optimized statement:

```
select q6.s_name as "S_NAME", q6.s_address as "S_ADDRESS",
    q6.s_phone as "S_PHONE", q6.s_comment as "S_COMMENT"
  from (select min(q4.$c0)
```

```
        from (select q2.ps_supplycost
                from sm_samp.suppliers as q1, sm_samp.partsupp as q2
                where q1.s_nation = 'MOROCCO' and
                  q1.s_suppkey = q2.ps_suppkey and
                  q7.p_partkey = q2.ps_partkey
                ) as q3
        ) as q4, sm_samp.partsupp as q5, sm_samp.suppliers as q6,
            sm_samp.parts as q7
  where p_size = 39 and
    q5.ps_supplycost = q4.$c0 and
    q6.s_nation in ('MOROCCO', 'SPAIN') and
    q7.p_type = 'BRASS' and
    q6.s_suppkey = q5.ps_suppkey and
    q7.p_partkey = q5.ps_partkey
```

This optimization guideline shows a hash join request, where the SUPPLIERS table in the nested subselect is the outer table, as specified by the TBSCAN access request element, and where the PARTSUPP table in the nested subselect is the inner table, as specified by the IXSCAN access request element. The TBSCAN access request element uses the TABLE attribute to identify the SUPPLIERS table reference using the corresponding exposed name in the original statement. The IXSCAN access request element, on the other hand, uses the TABID attribute to identify the PARTSUPP table reference using the unique correlation name that is associated with that table reference in the optimized statement.

If a single optimization guideline specifies both the TABLE and TABID attributes, they must identify the same table reference, or the optimization guideline is ignored.

Note: There is currently no guarantee that correlation names in the optimized statement will be stable when upgrading to a new release of the Db2 product.

Ambiguous table references

An optimization guideline is considered invalid and is not applied if it matches multiple exposed or extended names.

```
create view v1 as
  (select * from employee
     where salary > (select avg(salary) from employee)

select * from v1
  where deptno in ('M62', 'M63')

<OPTGUIDE>
  <IXSCAN TABLE='V1/EMPLOYEE'/>
</OPTGUIDE>
```

The optimizer considers the IXSCAN access request ambiguous, because the exposed name EMPLOYEE is not unique within the definition of view V1.

To eliminate the ambiguity, the view can be rewritten to use unique correlation names, or the TABID attribute can be used. Table references that are identified by the TABID attribute are never ambiguous, because all correlation names in the optimized statement are unique.

Conflicting optimization guidelines

Multiple optimization guidelines cannot identify the same table reference.

For example:

```
<OPTGUIDELINES>
  <IXSCAN TABLE='"Samp".PARTS' INDEX='I_PTYPE'/>
  <IXSCAN TABLE='"Samp".PARTS' INDEX='I_SIZE'/>
</OPTGUIDELINES>
```

Each of the IXSCAN elements references the "Samp".PARTS table in the main subselect.

When two or more guidelines refer to the same table, only the first is applied; all other guidelines are ignored, and an error is returned.

Only one INLIST2JOIN query rewrite request element at the predicate level per query can be enabled.The following example illustrates an unsupported query rewrite optimization guideline, where two IN-LIST predicates are enabled at the predicate level. Both guidelines are ignored, and a warning is returned.

```
<OPTGUIDELINES>
  <INLIST2JOIN TABLE='P' COLUMN='P_SIZE'/>
 <INLIST2JOIN TABLE='P' COLUMN='P_TYPE'/>
</OPTGUIDELINES>
```

Verifying that optimization guidelines have been used:

The optimizer makes every attempt to adhere to the optimization guidelines that are specified in an optimization profile; however, the optimizer can reject invalid or inapplicable guidelines.

Before you begin

Explain tables must exist before you can use the explain facility. The data definition language (DDL) for creating the explain tables is contained in `EXPLAIN.DDL`, which can be found in the `misc` subdirectory of the `sqllib` directory.

Procedure

To verify that a valid optimization guideline has been used:

1. Issue the EXPLAIN statement against the statement to which the guidelines apply. If an optimization guideline was in effect for the statement using an optimization profile, the optimization profile name appears as a RETURN operator argument in the EXPLAIN_ARGUMENT table. And if the optimization guideline contained an SQL embedded optimization guideline or statement profile that matched the current statement, the name of the statement profile appears as a RETURN operator argument. The types of these two new argument values are OPT_PROF and STMTPROF.
2. Examine the results of the explained statement. The following query against the explain tables can be modified to return the optimization profile name and the statement profile name for your particular combination of EXPLAIN_REQUESTER, EXPLAIN_TIME, SOURCE_NAME, SOURCE_VERSION, and QUERYNO:

```
SELECT VARCHAR(B.ARGUMENT_TYPE,   9) as TYPE,
     VARCHAR(B.ARGUMENT_VALUE, 24) as VALUE

FROM   EXPLAIN_STATEMENT A, EXPLAIN_ARGUMENT B

WHERE  A.EXPLAIN_REQUESTER = 'SIMMEN'
  AND  A.EXPLAIN_TIME      = '2003-09-08-16.01.04.108161'
  AND  A.SOURCE_NAME       = 'SQLC2E03'
  AND  A.SOURCE_VERSION    = ''
  AND  A.QUERYNO           = 1
```

```
AND  A.EXPLAIN_REQUESTER = B.EXPLAIN_REQUESTER
AND  A.EXPLAIN_TIME      = B.EXPLAIN_TIME
AND  A.SOURCE_NAME       = B.SOURCE_NAME
AND  A.SOURCE_SCHEMA     = B.SOURCE_SCHEMA
AND  A.SOURCE_VERSION    = B.SOURCE_VERSION
AND  A.EXPLAIN_LEVEL     = B.EXPLAIN_LEVEL
AND  A.STMTNO            = B.STMTNO
AND  A.SECTNO            = B.SECTNO

AND  A.EXPLAIN_LEVEL     = 'P'

AND  (B.ARGUMENT_TYPE  = 'OPT_PROF' OR ARGUMENT_TYPE = 'STMTPROF')
AND  B.OPERATOR_ID = 1
```

If the optimization guideline is active and the explained statement matches the statement that is contained in the STMTKEY element of the optimization guideline, a query that is similar to the previous example produces output that is similar to the following output. The value of the STMTPROF argument is the same as the ID attribute in the STMTPROFILE element.

```
TYPE      VALUE
--------- -------------------------
OPT_PROF  NEWTON.PROFILE1
STMTPROF  Guidelines for SAMP Q9
```

Embedded optimization guidelines:

You can also specify optimization guidelines through embedded optimization guidelines, which require no extra configuration with SYSTOOLS.OPT_PROFILE tables. These are optimization guidelines within C-style comments placed at the end of SQL statements.

To use embedded optimization guidelines, simply surround the XML <OPTGUIDELINES> start tag and the </OPTGUIDELINES> end tag of your optimization guideline with a C-style comment. Place the comment at the end of your SQL statement. For example:

```
SELECT S.S_NAME, S.S_ADDRESS, S.S_PHONE, S.S_COMMENT
FROM "Tpcd".PARTS, "Tpcd".SUPPLIERS S, "Tpcd".PARTSUPP PS
WHERE P_PARTKEY = PS.PS_PARTKEY AND S.S_SUPPKEY = PS.PS_SUPPKEY AND
   P_SIZE = 39 AND P_TYPE = 'BRASS' AND
   S.S_NATION IN ('MOROCCO', 'SPAIN') AND
   PS.PS_SUPPLYCOST = (SELECT MIN(PS1.PS_SUPPLYCOST)
   FROM "Tpcd".PARTSUPP PS1, "Tpcd".SUPPLIERS S1
   WHERE "Tpcd".PARTS.P_PARTKEY = PS1.PS_PARTKEY AND
     S1.S_SUPPKEY = PS1.PS_SUPPKEY AND
     S1.S_NATION = S.S_NATION)
ORDER BY S.S_NAME
OPTIMIZE FOR 1 ROWS
/* <OPTGUIDELINES><IXSCAN TABLE='"Tpcd".PARTS'/></OPTGUIDELINES> */
```

General rules when using embedded optimization guidelines:

- Embedded optimization guidelines can only be applied to Data Manipulation Language (DML) statements: the **SELECT**, **INSERT**, **UPDATE**, **DELETE**, and **MERGE** commands. The optimizer will ignore such comments on other types of statements. No error or warning will be provided.
- The embedded optimization guideline must be provided after the SQL portion of the statement. They cannot appear inside subselects. However, other types of comments can be provided at the end of the statement before or after the optimization guideline.

- The optimizer will look for one embedded optimization guideline comment for every DML statement. If there are multiple embedded optimization guideline comments, all of them are ignored and a warning is produced.
- The optimization guideline must be written in well-formed XML. It cannot include extraneous text.
- Embedded optimization guidelines override identical optimization guidelines specified in the global section of an optimization profile.
- Optimization guidelines provided by way of a statement profile section of an optimization profile take precedence over embedded optimization guidelines. That is, if the CURRENT OPTIMIZATION PROFILE register contains the name of an optimization profile, and the specified optimization profile contains a matching statement profile for a statement with embedded optimization guidelines, then the embedded optimization guidelines are ignored by the optimizer.

Optimization guidelines and profiles for column-organized tables:

Optimization guidelines and profiles are supported on column-organized tables. In general, any table access or join request not involving an index scan is allowed.

SQL0437W with reason code 13 is returned when a guideline that references column-organized tables cannot be satisfied. Also, EXP0035W is displayed in the `Extended Diagnostic Information` section of **db2exfmt** command output. The following table illustrates the types of optimization requests that are allowed or not allowed for column-organized tables:

Type of request	Whether allowed and messages returned
ACCESS (any table access)	Allowed
TBSCAN (table scan) [1]	Allowed
IXSCAN (index scan)	Not allowed (SQL0437W, reason code 13; EXP0035W)
LPREFETCH (list prefetch)	Not allowed (SQL0437W, reason code 13; EXP0035W)
IXAND (index ANDing)	Not allowed (SQL0437W, reason code 13; EXP0035W)
IOA (index ORing)	Not allowed (SQL0437W, reason code 13; EXP0035W)
JOIN (any join)	Allowed
HSJOIN (hash join) [2]	Allowed
NLJOIN (nested loop join)	Allowed
MSJOIN	Allowed
STARJOIN (IXA-ANDing star join) [3]	Not allowed (SQL0437W, reason code 13; EXP0035W)

Type of request	Whether allowed and messages returned
Note: 1. You cannot create an index on a column-organized table, and only table scans can be used. Any table access request that requires an index scan cannot be satisfied. 2. A column-organized table supports only HSJOIN and NLJOIN requests. Any join request that references column-organized tables can be satisfied by retrieving the data from the tables and performing the join by using row-organized data processing. If the requested join method is HSJOIN or NLJOIN, a plan with HSJOIN or NLJOIN being pushed down to column-organized data processing can also be used to satisfy the request, assuming they are eligible considering the type of join predicates being applied. HSJOIN is only eligible if there is at least one equality join predicate, and NLJOIN is only eligible if there are no equality join predicates. 3. A star join can contain only row-organized tables.	

XML schemas for access plan and query optimization profiles and guidelines:

Access plan and query optimization profiles are written as XML documents stored in the database. Each component has a specified XML format, as defined in the schema, that must be used in order for an optimization profile to be validated for use. You might find it helpful to use the samples as reference while designing profiles.

Current optimization profile schema:

The valid optimization profile contents for a given Db2 release is described by an XML schema that is known as the current optimization profile schema (COPS). An optimization profile applies only to Db2 Database for Linux, UNIX, and Windows servers.

The following listing represents the COPS for the current release of the Db2 product. The COPS can also be found in `DB2OptProfile.xsd`, which is located in the `misc` subdirectory of the `sqllib` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" version="1.0">
<!--****************************************************************************************-->
<!-- Licensed Materials - Property of IBM                                                   -->
<!-- (C) Copyright International Business Machines Corporation 2009. All rights reserved.   -->
<!-- U.S. Government Users Restricted Rights; Use, duplication or disclosure restricted by  -->
<!-- GSA ADP Schedule Contract with IBM Corp.                                               -->
<!--****************************************************************************************-->
<!--****************************************************************************************-->
<!-- Definition of the current optimization profile schema for V9.7.0.0                     -->
<!--                                                                                        -->
<!-- An optimization profile is composed of the following sections:                         -->
<!--                                                                                        -->
<!-- + A global optimization guidelines section (at most one) which defines optimization    -->
<!-- guidelines affecting any statement for which the optimization profile is in effect.    -->
<!--                                                                                        -->
<!-- + Zero or more statement profile sections, each of which defines optimization          -->
<!-- guidelines for a particular statement for which the optimization profile               -->
<!-- is in effect.                                                                          -->
<!--                                                                                        -->
<!-- The VERSION attribute indicates the version of this optimization profile               -->
<!-- schema.                                                                                -->
<!--****************************************************************************************-->
<xs:element name="OPTPROFILE">
   <xs:complexType>
      <xs:sequence>
         <!-- Global optimization guidelines section. At most one can be specified. -->
         <xs:element name="OPTGUIDELINES" type="globalOptimizationGuidelinesType" minOccurs="0"/>
         <!-- Statement profile section. Zero or more can be specified -->
         <xs:element name="STMTPROFILE" type="statementProfileType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <!-- Version attribute is currently optional -->
```

```
      <xs:attribute name="VERSION" use="optional"/>
   </xs:complexType>
</xs:element>

<!--*****************************************************************************************-->
<!-- Global optimization guidelines supported in this version:                               -->
<!-- + MQTOptimizationChoices elements influence the MQTs considered by the optimizer.       -->
<!-- + computationalPartitionGroupOptimizationsChoices elements can affect repartitioning    -->
<!-- optimizations involving nicknames.                                                      -->
<!-- + General requests affect the search space which defines the alternative query          -->
<!-- transformations, access methods, join methods, join orders, and other optimizations,    -->
<!-- considered by the compiler and optimizer.                                               -->
<!-- + MQT enforcement requests specify semantically matchable MQTs whose usage in access    -->
<!-- plans should be enforced regardless of cost estimates.                                  -->
<!-- *****************************************************************************************-->
<xs:complexType name="globalOptimizationGuidelinesType">
   <xs:sequence>
      <xs:group ref="MQTOptimizationChoices" />
      <xs:group ref="computationalPartitionGroupOptimizationChoices" />
      <xs:group ref="generalRequest"/>
      <xs:group ref="mqtEnforcementRequest" />
   </xs:sequence>
</xs:complexType>
<!-- *****************************************************************************************-->
<!-- Elements for affecting materialized query table (MQT) optimization.                     -->
<!--                                                                                         -->
<!-- + MQTOPT - can be used to disable materialized query table (MQT) optimization.          -->
<!-- If disabled, the optimizer will not consider MQTs to optimize the statement.            -->
<!--                                                                                         -->
<!-- + MQT - multiple of these can be specified. Each specifies an MQT that should be        -->
<!-- considered for optimizing the statement. Only specified MQTs will be considered.        -->
<!--                                                                                         -->
<!--*****************************************************************************************-->
<xs:group name="MQTOptimizationChoices">
   <xs:choice>
      <xs:element name="MQTOPT" minOccurs="0" maxOccurs="1">
         <xs:complexType>
            <xs:attribute name="OPTION" type="optionType" use="optional"/>
         </xs:complexType>
      </xs:element>
      <xs:element name="MQT" minOccurs="0" maxOccurs="unbounded">
         <xs:complexType>
            <xs:attribute name="NAME" type="xs:string" use="required"/>
         </xs:complexType>
      </xs:element>
   </xs:choice>
</xs:group>
<!-- *****************************************************************************************-->
<!-- Elements for affecting computational partition group (CPG) optimization.               -->
<!--                                                                                         -->
<!-- + PARTOPT - can be used disable the computational partition group (CPG) optimization    -->
<!-- which is used to dynamically redistributes inputs to join, aggregation,                 -->
<!-- and union operations when those inputs are results of remote queries.                   -->
<!--                                                                                         -->
<!-- + PART - Define the partition groups to be used in CPG optimizations.                   -->
<!--                                                                                         -->
<!-- *****************************************************************************************-->
<xs:group name="computationalPartitionGroupOptimizationChoices">
   <xs:choice>
      <xs:element name="PARTOPT" minOccurs="0" maxOccurs="1">
         <xs:complexType>
            <xs:attribute name="OPTION" type="optionType" use="optional"/>
      </xs:complexType>
      </xs:element>
      <xs:element name="PART" minOccurs="0" maxOccurs="1">
         <xs:complexType>
            <xs:attribute name="NAME" type="xs:string" use="required"/>
         </xs:complexType>
      </xs:element>
   </xs:choice>
</xs:group>
<!-- *****************************************************************************************-->
<!-- Definition of a statement profile.                                                      -->
<!-- Comprised of a statement key and optimization guidelines.                               -->
<!-- The statement key specifies semantic information used to identify the statement to      -->
<!-- which optimization guidelines apply. The optional ID attribute provides the statement   -->
<!-- profile with a name for use in EXPLAIN output.                                          -->
<!-- *****************************************************************************************-->
<xs:complexType name="statementProfileType">
```

```
   <xs:sequence>
      <!-- Statement key element -->
      <xs:element name="STMTKEY" type="statementKeyType"/>
      <xs:element name="OPTGUIDELINES" type="optGuidelinesType"/>
   </xs:sequence>
   <!-- ID attribute.Used in explain output to indicate the statement profile was used. -->
   <xs:attribute name="ID" type="xs:string" use="optional"/>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of the statement key. The statement key provides semantic information used  -->
<!-- to identify the statement to which the optimization guidelines apply.                  -->
<!-- The statement key is comprised of:                                                     -->
<!-- + statement text (as written in the application)                                       -->
<!-- + default schema (for resolving unqualified table names in the statement)              -->
<!-- + function path (for resolving unqualified types and functions in the statement)       -->
<!-- The statement text is provided as element data whereas the default schema and function -->
<!-- path are provided via the SCHEMA and FUNCPATH elements, respectively.                  -->
<!--*****************************************************************************************-->
<xs:complexType name="statementKeyType" mixed="true">
   <xs:attribute name="SCHEMA" type="xs:string" use="optional"/>
   <xs:attribute name="FUNCPATH" type="xs:string" use="optional"/>
</xs:complexType>

<!--*****************************************************************************************-->
<!--                                                                                       -->
<!-- Optimization guideline elements can be chosen from general requests, rewrite          -->
<!-- requests access requests, or join requests.                                           -->
<!--                                                                                       -->
<!-- General requests affect the search space which defines the alternative query          -->
<!-- transformations, access methods, join methods, join orders, and other optimizations,  -->
<!-- considered by the optimizer.                                                          -->
<!--                                                                                       -->
<!-- Rewrite requests affect the query transformations used in determining the optimized   -->
<!-- statement.                                                                            -->
<!--                                                                                       -->
<!-- Access requests affect the access methods considered by the cost-based optimizer,     -->
<!-- and join requests affect the join methods and join order used in the execution plan.  -->
<!--                                                                                       -->
<!-- MQT enforcement requests specify semantically matchable MQTs whose usage in access    -->
<!-- plans should be enforced regardless of cost estimates.                                -->
<!--                                                                                       -->
<!--*****************************************************************************************-->
<xs:element name="OPTGUIDELINES" type="optGuidelinesType"/>
<xs:complexType name="optGuidelinesType">
   <xs:sequence>
      <xs:group ref="generalRequest" minOccurs="0" maxOccurs="1"/>
      <xs:choice maxOccurs="unbounded">
         <xs:group ref="rewriteRequest" />
         <xs:group ref="accessRequest"/>
         <xs:group ref="joinRequest"/>
         <xs:group ref="mqtEnforcementRequest"/>
      </xs:choice>
   </xs:sequence>
</xs:complexType>
<!--***************************************************************************************** -->
<!-- Choices of general request elements.                                                  -->
<!-- REOPT can be used to override the setting of the REOPT bind option.                   -->
<!-- DPFXMLMOVEMENT can be used to affect the optimizer's plan when moving XML documents  -->
<!-- between database partitions. The value can be NONE, REFERENCE or COMBINATION. The    -->
<!-- default value is NONE.                                                                -->
<!--***************************************************************************************** -->
<xs:group name="generalRequest">
   <xs:sequence>
      <xs:element name="REOPT" type="reoptType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="DEGREE" type="degreeType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="QRYOPT" type="qryoptType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="RTS" type="rtsType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="DPFXMLMOVEMENT" type="dpfXMLMovementType" minOccurs="0" maxOccurs="1"/>
   </xs:sequence>
</xs:group>
<!--*****************************************************************************************-->
<!-- Choices of rewrite request elements.                                                -->
<!--*****************************************************************************************-->
<xs:group name="rewriteRequest">
   <xs:sequence>
      <xs:element name="INLIST2JOIN" type="inListToJoinType" minOccurs="0"/>
      <xs:element name="SUBQ2JOIN" type="subqueryToJoinType" minOccurs="0"/>
      <xs:element name="NOTEX2AJ" type="notExistsToAntiJoinType" minOccurs="0"/>
      <xs:element name="NOTIN2AJ" type="notInToAntiJoinType" minOccurs="0"/>
```

```
   </xs:sequence>
</xs:group>
<!--****************************************************************************** -->
<!-- Choices for access request elements.                                         -->
<!-- TBSCAN - table scan access request element                                   -->
<!-- IXSCAN - index scan access request element                                   -->
<!-- LPREFETCH - list prefetch access request element                             -->
<!-- IXAND - index ANDing access request element                                  -->
<!-- IXOR - index ORing access request element                                    -->
<!-- XISCAN - xml index access request element                                    -->
<!-- XANDOR - XANDOR access request element                                       -->
<!-- ACCESS - indicates the optimizer should choose the access method for the table -->
<!--****************************************************************************** -->
<xs:group name="accessRequest">
   <xs:choice>
      <xs:element name="TBSCAN" type="tableScanType"/>
      <xs:element name="IXSCAN" type="indexScanType"/>
      <xs:element name="LPREFETCH" type="listPrefetchType"/>
      <xs:element name="IXAND" type="indexAndingType"/>
      <xs:element name="IXOR" type="indexOringType"/>
      <xs:element name="XISCAN" type="indexScanType"/>
      <xs:element name="XANDOR" type="XANDORType"/>
      <xs:element name="ACCESS" type="anyAccessType"/>
   </xs:choice>
</xs:group>
<!--****************************************************************************** -->
<!-- Choices for join request elements.                                           -->
<!-- NLJOIN - nested-loops join request element                                   -->
<!-- MSJOIN - sort-merge join request element                                     -->
<!-- HSJOIN - hash join request element                                           -->
<!-- JOIN - indicates that the optimizer is to choose the join method.            -->
<!--****************************************************************************** -->
<xs:group name="joinRequest">
   <xs:choice>
      <xs:element name="NLJOIN" type="nestedLoopJoinType"/>
      <xs:element name="HSJOIN" type="hashJoinType"/>
      <xs:element name="MSJOIN" type="mergeJoinType"/>
      <xs:element name="JOIN" type="anyJoinType"/>
   </xs:choice>
</xs:group>
<!--******************************************************************************--->
<!-- MQT enforcement request element.                                            -->
<!-- MQTENFORCE - This element can be used to specify semantically matchable MQTs whose  -->
<!--   usage in access plans should be enforced regardless of Optimizer cost estimates.  -->
<!--   MQTs can be specified either directly with the NAME attribute or generally using  -->
<!--   the TYPE attribute.                                                        -->
<!--   Only the first valid attribute found is used and all subsequent ones are ignored. -->
<!--   Since this element can be specified multiple times, more than one MQT can be      -->
<!--   enforced at a time.                                                        -->
<!--   Note however, that if there is a conflict when matching two enforced MQTs to the  -->
<!--   same data source (base-table or derived) an MQT will be picked based on existing  -->
<!--   tie-breaking rules, i.e., either heuristic or cost-based.                  -->
<!--   Finally, this request overrides any other MQT optimization options specified in   -->
<!--   a profile, i.e., enforcement will take place even if MQTOPT is set to DISABLE or  -->
<!--   if the specified MQT or MQTs do not exist in the eligibility list specified by    -->
<!--   any MQT elements.                                                          -->
<!--******************************************************************************--->
<xs:group name="mqtEnforcementRequest">
   <xs:sequence>
      <xs:element name="MQTENFORCE" type="mqtEnforcementType" minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:group>
<!--****************************************************************************** -->
<!-- REOPT general request element. Can override REOPT setting at the package, db,            -->
<!-- dbm level.                                                                   -->
<!--****************************************************************************** -->
<xs:complexType name="reoptType">
   <xs:attribute name="VALUE" use="required">
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="ONCE"/>
            <xs:enumeration value="ALWAYS"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:attribute>
</xs:complexType>
<!--******************************************************************************--->
<!-- RTS general request element to enable, disable or provide a time budget for               -->
<!-- real-time statistics collection.                                             -->
```

```
<!-- OPTION attribute allows enabling or disabling real-time statistics.                    -->
<!-- TIME attribute provides a time budget in milliseconds for real-time statistics collection.-->
<!--*****************************************************************************************-->
<xs:complexType name="rtsType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
   <xs:attribute name="TIME" type="xs:nonNegativeInteger" use="optional"/>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of an "IN list to join" rewrite request                                  -->
<!-- OPTION attribute allows enabling or disabling the alternative.                       -->
<!-- TABLE attribute allows request to target IN list predicates applied to a             -->
<!-- specific table reference. COLUMN attribute allows request to target a specific IN list   -->
<!-- predicate.                                                                           -->
<!--*****************************************************************************************-->
<xs:complexType name="inListToJoinType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
   <xs:attribute name="TABLE" type="xs:string" use="optional"/>
   <xs:attribute name="COLUMN" type="xs:string" use="optional"/>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of a "subquery to join" rewrite request                                  -->
<!-- The OPTION attribute allows enabling or disabling the alternative.                   -->
<!--*****************************************************************************************-->
<xs:complexType name="subqueryToJoinType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of a "not exists to anti-join" rewrite request                           -->
<!-- The OPTION attribute allows enabling or disabling the alternative.                   -->
<!--*****************************************************************************************-->
<xs:complexType name="notExistsToAntiJoinType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of a "not IN to anti-join" rewrite request                               -->
<!-- The OPTION attribute allows enabling or disabling the alternative.                   -->
<!--*****************************************************************************************-->
<xs:complexType name="notInToAntiJoinType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Effectively the superclass from which all access request elements inherit.           -->
<!-- This type currently defines TABLE and TABID attributes, which can be used to tie an  -->
<!-- access request to a table reference in the query.                                    -->
<!-- The TABLE attribute value is used to identify a table reference using identifiers    -->
<!-- in the original SQL statement. The TABID attribute value is used to identify a table -->
<!-- referece using the unique correlation name provided via the                          -->
<!-- optimized statement. If both the TABLE and TABID attributes are specified, the TABID -->
<!-- field is ignored. The FIRST attribute indicates that the access should be the first  -->
<!-- access in the join sequence for the FROM clause.                                     -->
<!-- The SHARING attribute indicates that the access should be visible to other concurrent -->
<!-- similar accesses that may therefore share bufferpool pages. The WRAPPING attribute   -->
<!-- indicates that the access should be allowed to perform wrapping, thereby allowing it to  -->
<!-- start in the middle for better sharing with other concurrent accesses. The THROTTLE  -->
<!-- attribute indicates that the access should be allowed to be throttled if this may    -->
<!-- benefit other concurrent accesses. The SHARESPEED attribute is used to indicate whether  -->
<!-- the access should be considered fast or slow for better grouping of concurrent accesses. -->
<!--***************************************************************************************** -->
<xs:complexType name="accessType" abstract="true">
   <xs:attribute name="TABLE" type="xs:string" use="optional"/>
   <xs:attribute name="TABID" type="xs:string" use="optional"/>
   <xs:attribute name="FIRST" type="xs:string" use="optional" fixed="TRUE"/>
   <xs:attribute name="SHARING" type="optionType" use="optional" default="ENABLE"/>
   <xs:attribute name="WRAPPING" type="optionType" use="optional" default="ENABLE"/>
   <xs:attribute name="THROTTLE" type="optionType" use="optional" default="ENABLE"/>
   <xs:attribute name="SHARESPEED" type="shareSpeed" use="optional"/>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of an table scan access request method.                                  -->
<!--*****************************************************************************************-->
<xs:complexType name="tableScanType">
   <xs:complexContent>
      <xs:extension base="accessType"/>
   </xs:complexContent>
</xs:complexType>
<!-- *****************************************************************************************-->
<!-- Definition of an index scan access request element. The index name is optional.      -->
<!--***************************************************************************** -->
<xs:complexType name="indexScanType">
```

```
   <xs:complexContent>
      <xs:extension base="accessType">
         <xs:attribute name="INDEX" type="xs:string" use="optional"/>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of a list prefetch access request element. The index name is optional.    -->
<!--***************************************************************************** -->
<xs:complexType name="listPrefetchType">
   <xs:complexContent>
      <xs:extension base="accessType">
         <xs:attribute name="INDEX" type="xs:string" use="optional"/>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of an extended access element which will be used by IXAND and ACCESS      -->
<!-- requests.                                                                            -->
<!-- A single index scan be specified via the INDEX attribute. Multiple indexes           -->
<!-- can be specified via INDEX elements. The index element specification supersedes the  -->
<!-- attribute specification. If a single index is specified, the optimizer will use the  -->
<!-- index as the first index of the index ANDing access method and will choose addi-     -->
<!-- tional indexes using cost. If multiple indexes are specified the optimizer will      -->
<!-- use exactly those indexes in the specified order. If no indexes are specified        -->
<!-- via either the INDEX attribute or INDEX elements, then the optimizer will choose     -->
<!-- all indexes based upon cost.                                                         -->
<!-- Extension for XML support:                                                           -->
<!-- TYPE: Optional attribute. The allowed value is XMLINDEX. When the type is not        -->
<!-- specified, the optimizer makes a cost based decision.                                -->
<!-- ALLINDEXES: Optional attribute. The allowed value is TRUE. The default               -->
<!-- value is FALSE.                                                                      -->
<!--***************************************************************************** -->
<xs:complexType name="extendedAccessType">
   <xs:complexContent>
      <xs:extension base="accessType">
         <xs:sequence minOccurs="0">
            <xs:element name="INDEX" type="indexType" minOccurs="2" maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute name="INDEX" type="xs:string" use="optional"/>
         <xs:attribute name="TYPE" type="xs:string" use="optional" fixed="XMLINDEX"/>
         <xs:attribute name="ALLINDEXES" type="boolType" use="optional" fixed="TRUE"/>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of an index ANDing access request element.                                -->
<!-- Extension for XML support:                                                           -->
<!--   All attributes and elements in extendedAccessType are included.                    -->
<!--   Note that ALLINDEXES is a valid option only if TYPE is XMLINDEX.                   -->
<!-- STARJOIN index ANDing: Specifying STARJOIN='TRUE' or one or more NLJOIN elements     -->
<!-- identifies the index ANDing request as a star join index ANDing request. When that   -->
<!-- is the case:                                                                         -->
<!--   TYPE cannot be XMLINDEX (and therefore ALLINDEXES cannot be specified).            -->
<!--   Neither the INDEX attribute nor INDEX elements can be specified.                   -->
<!--   The TABLE or TABID attribute identifies the fact table.                            -->
<!--   Zero or more semijoins can be specified using NLJOIN elements.                    -->
<!--   If no semijoins are specified, the optimizer will choose them.                     -->
<!--   If a single semijoin is specified, the optimizer will use it as the first semijoin -->
<!--   and will choose the rest itself.                                                   -->
<!--   If multiple semijoins are specified the optimizer will use exactly those semijoins -->
<!--   in the specified order.                                                            -->
<!--***************************************************************************** -->
<xs:complexType name="indexAndingType">
   <xs:complexContent>
      <xs:extension base="extendedAccessType">
         <xs:sequence minOccurs="0">
            <xs:element name="NLJOIN" type="nestedLoopJoinType" minOccurs="1" maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute name="STARJOIN" type="boolType" use="optional"/>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of an INDEX element method. Index set is optional. If specified,          -->
<!-- at least 2 are required.                                                             -->
<!--***************************************************************************** -->
<xs:complexType name="indexType">
   <xs:attribute name="IXNAME" type="xs:string" use="optional"/>
```

```
</xs:complexType>
<!--****************************************************************************************-->
<!-- Definition of an XANDOR access request method.                                         -->
<!--****************************************************************************************-->
<xs:complexType name="XANDORType">
   <xs:complexContent>
      <xs:extension base="accessType"/>
   </xs:complexContent>
</xs:complexType>
<!--****************************************************************************************-->
<!-- Use for derived table access or other cases where the access method is not of        -->
<!-- consequence.                                                                           -->
<!-- Extension for XML support:                                                             -->
<!-- All attributes and elements in extendedAccessType are included.                        -->
<!-- Note that INDEX attribute/elements and ALLINDEXES are valid options only if TYPE      -->
<!-- is XMLINDEX.                                                                           -->
<!--****************************************************************************************-->
<xs:complexType name="anyAccessType">
   <xs:complexContent>
      <xs:extension base="extendedAccessType"/>
   </xs:complexContent>
</xs:complexType>
<!--****************************************************************************************-->
<!-- Definition of an index ORing access                                                    -->
<!-- Cannot specify more details (e.g indexes). Optimizer will choose the details based    -->
<!-- upon cost.                                                                             -->
<!--**************************************************************************************** -->
<xs:complexType name="indexOringType">
   <xs:complexContent>
      <xs:extension base="accessType"/>
   </xs:complexContent>
</xs:complexType>
<!--****************************************************************************************-->
<!-- Effectively the super class from which join request elements inherit.                  -->
<!-- This type currently defines join element inputs and also the FIRST attribute.         -->
<!-- A join request must have exactly two nested sub-elements. The sub-elements can be     -->
<!-- either an access request or another join request. The first sub-element represents    -->
<!-- outer table of the join operation while the second element represents the inner       -->
<!-- table. The FIRST attribute indicates that the join result should be the first join    -->
<!-- relative to other tables in the same FROM clause.                                      -->
<!--**************************************************************************************** -->
<xs:complexType name="joinType" abstract="true">
   <xs:choice minOccurs="2" maxOccurs="2">
      <xs:group ref="accessRequest"/>
      <xs:group ref="joinRequest"/>
   </xs:choice>
   <xs:attribute name="FIRST" type="xs:string" use="optional" fixed="TRUE"/>
</xs:complexType>
<!--**************************************************************************************** -->
<!-- Definition of nested loop join access request. Subclass of joinType.                   -->
<!-- Does not add any elements or attributes.                                               -->
<!--**************************************************************************************** -->
<xs:complexType name="nestedLoopJoinType">
   <xs:complexContent>
      <xs:extension base="joinType"/>
   </xs:complexContent>
</xs:complexType>
<!--****************************************************************************************-->
<!-- Definition of merge join access request. Subclass of joinType.                         -->
<!-- Does not add any elements or attributes.                                               -->
<!--**************************************************************************************** -->
<xs:complexType name="mergeJoinType">
   <xs:complexContent>
      <xs:extension base="joinType"/>
   </xs:complexContent>
</xs:complexType>
<!--**************************************************************************************** -->
<!-- Definition of hash join access request. Subclass of joinType.                          -->
<!-- Does not add any elements or attributes.                                               -->
<!--**************************************************************************************** -->
<xs:complexType name="hashJoinType">
   <xs:complexContent>
      <xs:extension base="joinType"/>
   </xs:complexContent>
</xs:complexType>
<!--**************************************************************************************** -->
<!-- Any join is a subclass of binary join. Does not extend it in any way.                  -->
<!-- Does not add any elements or attributes.                                               -->
<!--**************************************************************************************** -->
```

```
<xs:complexType name="anyJoinType">
   <xs:complexContent>
      <xs:extension base="joinType"/>
   </xs:complexContent>
</xs:complexType>
<!--*****************************************************************************-->
<!-- The MQTENFORCE element can be specified with one of two attributes:         -->
<!--   NAME: Specify the MQT name directly as a value to this attribute.         -->
<!--   TYPE: Specify the type of the MQTs that should be enforced with this attribute.    -->
<!--   Note that only the value of the first valid attribute found will be used. All      -->
<!--   subsequent attributes will be ignored.                                    -->
<!--*****************************************************************************-->
<xs:complexType name="mqtEnforcementType">
   <xs:attribute name="NAME" type="xs:string"/>
   <xs:attribute name="TYPE" type="mqtEnforcementTypeType"/>
</xs:complexType>
<!--*****************************************************************************-->
<!-- Allowable values for the TYPE attribute of an MQTENFORCE element:           -->
<!--   NORMAL: Enforce usage of all semantically matchable MQTs, except replicated MQTs.      -->
<!--   REPLICATED: Enforce usage of all semantically matchable replicated MQTs only.          -->
<!--   ALL: Enforce usage of all semantically matchable MQTs.                    -->
<!--***************************************************************************** -->
<xs:simpleType name="mqtEnforcementTypeType">
   <xs:restriction base="xs:string">
      <xs:enumeration value="NORMAL"/>
      <xs:enumeration value="REPLICATED"/>
      <xs:enumeration value="ALL"/>
   </xs:restriction>
</xs:simpleType>
<!--*****************************************************************************-->
<!-- Allowable values for a boolean attribute.                                   -->
<!--***************************************************************************** -->
<xs:simpleType name="boolType">
   <xs:restriction base="xs:string">
      <xs:enumeration value="TRUE"/>
      <xs:enumeration value="FALSE"/>
   </xs:restriction>
</xs:simpleType>
<!--*****************************************************************************-->
<!-- Allowable values for an OPTION attribute.                                   -->
<!--***************************************************************************** -->
<xs:simpleType name="optionType">
   <xs:restriction base="xs:string">
      <xs:enumeration value="ENABLE"/>
      <xs:enumeration value="DISABLE"/>
   </xs:restriction>
</xs:simpleType>
<!--*****************************************************************************-->
<!-- Allowable values for a SHARESPEED attribute.                                -->
<!--***************************************************************************** -->
<xs:simpleType name="shareSpeed">
   <xs:restriction base="xs:string">
      <xs:enumeration value="FAST"/>
      <xs:enumeration value="SLOW"/>
   </xs:restriction>
</xs:simpleType>
<!--*****************************************************************************-->
<!-- Definition of the qryopt type: the only values allowed are 0, 1, 2, 3, 5, 7 and 9       -->
<!--*****************************************************************************-->
<xs:complexType name="qryoptType">
   <xs:attribute name="VALUE" use="required">
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="0"/>
            <xs:enumeration value="1"/>
            <xs:enumeration value="2"/>
            <xs:enumeration value="3"/>
            <xs:enumeration value="5"/>
            <xs:enumeration value="7"/>
            <xs:enumeration value="9"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:attribute>
</xs:complexType>
<!--*****************************************************************************-->
<!-- Definition of the degree type: any number between 1 and 32767 or the strings ANY or -1  -->
<!--***************************************************************************** -->
<xs:simpleType name="intStringType">
   <xs:union>
```

```
      <xs:simpleType>
         <xs:restriction base="xs:integer">
            <xs:minInclusive value="1"></xs:minInclusive>
            <xs:maxInclusive value="32767"></xs:maxInclusive>
         </xs:restriction>
      </xs:simpleType>
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="ANY"/>
            <xs:enumeration value="-1"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:union>
</xs:simpleType>

<xs:complexType name="degreeType">
        <xs:attribute name="VALUE" type="intStringType"></xs:attribute>
</xs:complexType>
<!--*****************************************************************************************-->
<!-- Definition of DPF XML movement types                                                    -->
<!--*****************************************************************************************-->
<xs:complexType name="dpfXMLMovementType">
    <xs:attribute name="VALUE" use="required">
     <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="REFERENCE"/>
        <xs:enumeration value="COMBINATION"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
 </xs:complexType>

</xs:schema>
```

XML schema for the OPTPROFILE element:

The OPTPROFILE element is the root of an optimization profile.

This element is defined as follows:

XML Schema

```
   <xs:element name="OPTPROFILE">
      <xs:complexType>
         <xs:sequence>
            <!-- Global optimization guidelines section. -->
            <!-- At most one can be specified. -->
            <xs:element name="OPTGUIDELINES"
                   type="globalOptimizationGuidelinesType" minOccurs="0"/>
            <!-- Statement profile section. Zero or more can be specified -->
            <xs:element name="STMTPROFILE" type="statementProfileType"
                   minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
         <!-- Version attribute is currently optional -->
         <xs:attribute name="VERSION" use="optional"/>
      </xs:complexType>
   </xs:element>
```

Description

The optional OPTGUIDELINES sub-element defines the global optimization guidelines for the optimization profile. Each STMTPROFILE sub-element defines a statement profile. The VERSION attribute identifies the current optimization profile schema against which a specific optimization profile was created and validated.

XML schema for the global OPTGUIDELINES element:

The OPTGUIDELINES element defines the global optimization guidelines for the optimization profile.

It is defined by the complex type globalOptimizationGuidelinesType.

XML Schema

```
<xs:complexType name="globalOptimizationGuidelinesType">
   <xs:sequence>
      <xs:group ref="MQTOptimizationChoices"/>
      <xs:group ref="computationalPartitionGroupOptimizationChoices"/>
      <xs:group ref="generalRequest"/>
      <xs:group ref="mqtEnforcementRequest"/>
   </xs:sequence>
</xs:complexType>
```

Description

Global optimization guidelines can be defined with elements from the groups MQTOptimizationChoices, computationalPartitionGroupChoices, or generalRequest.

- MQTOptimizationChoices group elements can be used to influence MQT substitution.
- computationalPartitionGroupOptimizationChoices group elements can be used to influence computational partition group optimization, which involves the dynamic redistribution of data read from remote data sources. It applies only to partitioned federated database configurations.
- The generalRequest group elements are not specific to a particular phase of the optimization process, and can be used to change the optimizer's search space. They can be specified globally or at the statement level.
- MQT enforcement requests specify semantically matchable materialized query tables (MQTs) whose use in access plans should be enforced regardless of cost estimates.

MQT optimization choices:

The MQTOptimizationChoices group defines a set of elements that can be used to influence materialized query table (MQT) optimization. In particular, these elements can be used to enable or disable consideration of MQT substitution, or to specify the complete set of MQTs that are to be considered by the optimizer.

XML Schema

```
<xs:group name="MQTOptimizationChoices">
   <xs:choice>
      <xs:element name="MQTOPT" minOccurs="0" maxOccurs="1">
         <xs:complexType>
            <xs:attribute name="OPTION" type="optionType" use="optional"/>
         </xs:complexType>
      </xs:element>
      <xs:element name="MQT" minOccurs="0" maxOccurs="unbounded">
         <xs:complexType>
            <xs:attribute name="NAME" type="xs:string" use="required"/>
         </xs:complexType>
      </xs:element>
   </xs:choice>
</xs:group>
```

Description

The MQTOPT element is used to enable or disable consideration of MQT optimization. The OPTION attribute can take the value ENABLE (default) or DISABLE.

The NAME attribute of an MQT element identifies an MQT that is to be considered by the optimizer. The rules for forming a reference to an MQT in the NAME attribute are the same as those for forming references to exposed table names. If one or more MQT elements are specified, only those MQTs are considered by the optimizer. The decision to perform MQT substitution using one or more of the specified MQTs remains a cost-based decision.

Examples

The following example shows how to disable MQT optimization.

```
<OPTGUIDELINES>
   <MQTOPT OPTION='DISABLE'/>
</OPTGUIDELINES>
```

The following example shows how to limit MQT optimization to the Samp.PARTSMQT table and the COLLEGE.STUDENTS table.

```
<OPTGUIDELINES>
   <MQT NAME='Samp.PARTSMQT'/>
   <MQT NAME='COLLEGE.STUDENTS'/>
<OPTGUIDELINES>
```

Computational partition group optimization choices:

The computationalPartitionGroupOptimizationChoices group defines a set of elements that can be used to influence computational partition group optimization. In particular, these elements can be used to enable or disable computational group optimization, or to specify the partition group that is to be used for computational partition group optimization.

XML Schema

```
<xs:group name="computationalPartitionGroupOptimizationChoices">
   <xs:choice>
      <xs:element name="PARTOPT" minOccurs="0" maxOccurs="1">
         <xs:complexType>
            <xs:attribute name="OPTION" type="optionType" use="optional"/>
         </xs:complexType>
      </xs:element>
      <xs:element name="PART" minOccurs="0" maxOccurs="1">
         <xs:complexType>
            <xs:attribute name="NAME" type="xs:string" use="required"/>
         </xs:complexType>
      </xs:element>
   </xs:choice>
</xs:group>
```

Description

The PARTOPT element is used to enable or disable consideration of computational partition group optimization. The OPTION attribute can take the value ENABLE (default) or DISABLE.

The PART element can be used to specify the partition group that is to be used for computational partition group optimization. The NAME attribute must identify an existing partition group. The decision to perform dynamic redistribution using the specified partition group remains a cost-based decision.

Examples

The following example shows how to disable computational partition group optimization.

```
<OPTGUIDELINES>
   <PARTOPT OPTION='DISABLE'/>
</OPTGUIDELINES>
```

The following example shows how to specify that the WORKPART partition group is to be used for computational partition group optimization.

```
<OPTGUIDELINES>
   <MQT NAME='Samp.PARTSMQT'/>
   <PART NAME='WORKPART'/>
<OPTGUIDELINES>
```

General optimization guidelines as global requests:

The generalRequest group defines guidelines that are not specific to a particular phase of the optimization process, and can be used to change the optimizer's search space.

General optimization guidelines can be specified at both the global and statement levels. The description and syntax of general optimization guideline elements is the same for both global optimization guidelines and statement-level optimization guidelines. For more information, see "XML schema for general optimization guidelines".

XML schema for the STMTPROFILE element:

The STMTPROFILE element defines a statement profile within an optimization profile.

It is defined by the complex type statementProfileType.

XML Schema

```
<xs:complexType name="statementProfileType">
   <xs:sequence>
      <xs:element name="STMTMATCH" type="stmtMatchType" minOccurs="0"/>
      <xs:element name="STMTKEY" type="statementKeyType"/>
      <xs:element name="OPTGUIDELINES" type="optGuidelinesType"/>
   </xs:sequence>
   <xs:attribute name="ID" type="xs:string" use="optional"/>
</xs:complexType>
```

Description

A statement profile specifies optimization guidelines for a particular statement, and includes the following parts:

- Statement matching

  The statements in an optimization profile are either exactly or inexactly matched to the statement or statements that are compiled. The value of the STMTMATCH attribute represents which matching method is applied.
- Statement key

  An optimization profile can be in effect for more than one statement in an application. Using the statement key, the optimizer automatically matches each statement profile to a corresponding statement in the application. This matching provides optimization guidelines for a statement without editing the application.

The statement key includes the text of the statement (as written in the application), as well as other information that is needed to unambiguously identify the correct statement. The STMTKEY subelement represents the statement key.

- Statement-level optimization guidelines

  This part of the statement profile specifies the optimization guidelines in effect for the statement that is identified by the statement key. For information, see "XML schema for the statement-level OPTGUIDELINES element".

- Statement profile name

  A user-specified name that appears in diagnostic output to identify a particular statement profile.

XML schema for the STMTMATCH element:

The STMTMATCH element enables the matching method used for the optimization profile or for specific statements in the optimization profile.

It is defined by the complex type stmtMatchType.

```
XML Schema

   <xs:complexType name="stmtMatchType">
      <xs:attribute name="EXACT" type="boolType" use="optional" default="TRUE"/>
   </xs:complexType>
```

Description

The optional EXACT attribute specifies the matching method. If the value is set to TRUE, exact matching is applied. If the value is set to FALSE, inexact matching is applied. Exact matching is the default setting.

Example

The following example shows an STMTMATCH element definition at the statement level which enables inexact matching for the statement in the STMTKEY element.

```
<STMTPROFILE ID='S1'>
      <STMTMATCH  EXACT='FALSE'/>
      <STMTKEY>
          <![CDATA[select t1.c1, count(*) from t1,t2 where t1.c1 = t2.c1 and t1.c1 > 0]]>
      </STMTKEY>
      ...
</STMTPROFILE>
```

XML schema for the STMTKEY element:

The STMTKEY element enables the optimizer to match a statement profile to a corresponding statement in an application.

It is defined by the complex type statementKeyType.

```
XML Schema

   <xs:complexType name="statementKeyType" mixed="true">
      <xs:attribute name="SCHEMA" type="xs:string" use="optional"/>
      <xs:attribute name="FUNCPATH" type="xs:string" use="optional"/>
   </xs:complexType>
   </xs:schema>
```

Description

The optional SCHEMA attribute can be used to specify the default schema part of the statement key.

The optional FUNCPATH attribute can be used to specify the function path part of the statement key. Multiple paths must be separated by commas, and the specified function paths must match exactly the function paths that are specified in the compilation key.

Example

The following example shows a statement key definition that associates a particular statement with a default schema of 'COLLEGE' and a function path of 'SYSIBM,SYSFUN,SYSPROC,DAVE'.

```
<STMTKEY SCHEMA='COLLEGE' FUNCPATH='SYSIBM,SYSFUN,SYSPROC,DAVE'>
   <![CDATA[select * from orders" where foo(orderkey) > 20]]>
</STMTKEY>
```

CDATA tagging (starting with <![CDATA[and ending with]]>) is necessary because the statement text contains the special XML character '>'.

Statement key and compilation key matching:

The statement key identifies the application statement to which statement-level optimization guidelines apply. The matching method can be specified using the STMTMATCH element in the optimization profile.

When an SQL statement is compiled, various factors influence how the statement is interpreted semantically by the compiler. The SQL statement and the settings of SQL compiler parameters together form the compilation key. Each part of a statement key corresponds to some part of a compilation key.

A statement key is made of the following parts:

- Statement text, which is the text of the statement as written in the application
- Default schema, which is the schema name that is used as the implicit qualifier for unqualified table names. This part is optional but should be provided if there are unqualified table names in the statement.
- Function path, which is the function path that is used when resolving unqualified function and data-type references. This part is optional but should be provided if there are unqualified user-defined functions or user-defined types in the statement.

When the data server compiles an SQL statement and finds an active optimization profile, it attempts to match each statement key in the optimization profile with the current compilation key. The type of matching depends if exact or inexact matching is specified in the optimization profile. You can specify which type of matching to use by specifying the STMTMATCH element in the optimization profile. By setting the EXACT attribute to TRUE or FALSE, you can enable either exact or inexact matching. If you do not specify the STMTMATCH element, exact matching is automatically enabled.

For exact matching, a statement key and compilation key match if each specified part of the statement key matches the corresponding part of the compilation key. If a part of the statement key is not specified, the omitted part is considered matched

by default. Each unspecified part of the statement key is treated as a wild card that matches the corresponding part of any compilation key.

With inexact matching, literals, host variables, and parameter markers are ignored when the statement text from the statement key and compilation key is being matched.

Exact and inexact matching operates as follows:

- Matching is case insensitive for keywords. For example, `select` can match `SELECT`.
- Matching is case insensitive for nondelimited identifiers. For example, `T1` can match `t1`.
- Delimited and nondelimited identifiers can match except for one case. For example, `T1` and `"T1"` will match, and so will `t1` and `"T1"`. However, `t1` will not match with `"t1"`.

There is no match between `t1` and `"t1"` because `t1` and `"t1"` represent different tables. The nondelimited identifier `t1` is changed to uppercase and represents the table `T1` but in the case of the delimited identifier `"t1"` is not changed and represents the table `t1`. In this scenario, the case sensitivity matters because the two identifiers represent different tables.

After the data server finds a statement key that matches the current compilation key, it stops searching. If there are multiple statement profiles whose statement keys match the current compilation key, only the first such statement profile (based on document order) is used.

Inexact matching:

During compilation, if there is an active optimization profile, the compiling statements are matched either exactly or inexactly with the statements in the optimization profile.

Inexact matching is used for flexible matching between the compiling statements and the statements within the optimization profile. Inexact matching ignores literals, host variables, and parameter markers when matching the compiling statement to the optimization profile statements. Therefore, you can compile many different statements with different literal values in the predicate and the statements still match. For example, the following statements match inexactly but they do not match exactly:

```
select c1 into :hv1 from t1 where c1 > 10

select c1 into :hv2 from t1 where c1 > 20
```

Inexact matching is applied to both SQL and XQuery statements. However, string literals that are passed as function parameters representing SQL or XQuery statements or statement fragments, including individual column names are not inexactly matched. XML functions such as XMLQUERY, XMLTABLE, and XMLEXISTS that are used in an SQL statement are exactly matched. String literals could contain the following items:

- A whole statement with SQL embedded inside XQuery, or XQuery embedded inside an SQL statement
- An identifier, such as a column name
- An XML expression that contains a search path

For XQuery, inexact matching ignores only the literals. The following literals are ignored in inexact matching with some restrictions on the string literals:

- decimal literals
- double literals
- integer literals
- string literals that are not input parameters for functions: db2-fn:sqlquery, db2-fn:xmlcolumn, db2-fn:xmlcolumn-contains

The following XQuery statements match if inexact matching is enabled:

```
xquery let $i:= db2-fn:sqlquery("select c1 from tab1")/a/b[c=1] return $i

xquery let $i:= db2-fn:sqlquery("select c1 from tab1")/a/b[c=2] return $i
```

For inexact matching, the special register is not supported. The following examples show some of the type of statements that do not match in inexact matching:

- `c1 between 5 and :hv`
  `5 between c1 and c2`
- `c1 in (select c1 from t1)`
  `c1 in (1,2,3)`
- `c1 in (c1, 1, 2)`
  `c1 in (c2, 1, 2)`
- `A = 5`
  `A = 5 + :hv`
- `with RR`
  `with RS`
- `c2 < CURRENT TIME`
  `c2 < '11:12:40'`
- `c3 > CURRENT TIMESTAMP`
  `c3 > '07/29/2010'`

Syntax for specifying matching

Within the optimization profile, you can set either exact or inexact matching at either the global level or at the statement-level. The XML element STMTMATCH can be used to set the matching method.

The STMTMATCH element has an EXACT attribute which can be set to either TRUE or FALSE. If you specify the value TRUE, exact matching is enabled. If you specify the value FALSE, inexact matching is enabled. If you do not specify this element or if you specify only the STMTMATCH element without the EXACT attribute, exact matching is enabled by default.

To have a matching method apply to all statements within the optimization profile, place the STMTMATCH element at the global level just after the top OPTPROFILE element.

To have a matching method apply to a specific statement within the optimization profile, place the STMTMATCH element just after the STMTPROFILE element.

The following example shows an optimization profile with a STMTMATCH element at both the global level and statement level:

```
<?xml version="1.0" encoding="UTF-8"?>
<OPTPROFILE>

<!--Global section -->
   <STMTMATCH  EXACT='FALSE'/>

<!-- Statement level profile -->
```

```
   <STMTPROFILE ID='S1'>
      <STMTMATCH  EXACT='TRUE'/>
      <STMTKEY>
          <![CDATA[select t1.c1, count(*) from t1,t2 where t1.c1 = t2.c1 and t1.c1 > 0]]>
      </STMTKEY>
      <OPTGUIDELINES>
         <NLJOIN>
            <TBSCAN TABLE='T1'/>
            <TBSCAN TABLE='T2'/>
         </NLJOIN>
      </OPTGUIDELINES>
   </STMTPROFILE>

   <STMTPROFILE ID='S2'>
      <STMTKEY><![CDATA[select * from T1 where c1 in( 10,20)]]>
      </STMTKEY>
      <OPTGUIDELINES>
         <REGISTRY>
            <OPTION  NAME='DB2_REDUCED_OPTIMIZATION'  VALUE='YES'/>
         </REGISTRY>
      </OPTGUIDELINES>
   </STMTPROFILE>

</OPTPROFILE>
```

Order of precedence

In the example the STMTMATCH element has been set at both the global and statement level. Therefore, to determine which matching method gets executed depends on the order of precedence. The following is the order of precedence from highest to lowest:

1. Statement level profile settings
2. Global level profile settings

This order means that in the example, inexact matching is set at the global level because EXACT is set to FALSE. Therefore, inexact matching is applied to all the statements in the profile unless the compiling statement matches the first statement. Then, exact matching is enabled for that statement because the STMTMATCH element for that statement has EXACT set to TRUE.

The last statement in the example optimization profile file does not have an STMTMATCH element. The global setting takes precedence and inexact matching is applied to this statement.

Inexact matching examples for SQL statements in optimization profiles:

Inexact matching in optimization profiles occurs if you set the EXACT attribute to false in the STMTMATCH tag. The compiling statement is then matched to the statements in an active optimization profile. The compiler matches these statements based on different default matching rules and rules that are specific to inexact matching.

The following examples show where inexact matching is successful for matching SQL statements.

Example 1: Inexact matching in the predicate clause

Each of the following pair of statement fragments have different literal values for the predicate, but still match:

```
between '07/29/2010' and '08/29/2010'
between '09/29/2010' and '10/29/2010'
```

```
'ab'     like :hv1
'AYYANG' like :hv2

(A=1 AND B=1)       OR (A=2 AND B=2)
(A=:hv1 AND B=:hv2) OR (A=3 AND B=3)

c1 > 0 selectivity 0.1
c1 > 0 selectivity 0.9

c1 = ?
c1 = :hv1
```

Example 2: Inexact matching in the IN list predicate

All of the following statement fragments have different values in the IN list predicate, but still match:

```
c1 in (:hv1, :hv2, :hv3);

c1 in (:hv2, :hv3);

c1 in ( ?, ?, ?, ?);

c1 in (c1, c2 );

c1 in (:hv1, :hv2, c1, 1, 2, 3, c2, ?, ?);
```

Example 3: Inexact matching in the select list

The following statement fragment has different host variables in the select list, but still matches:

```
select c1 into :hv1 from t1
select c1 into :hv2 from t1
```

The following statement fragment has different literals in the select list, but still matches:

```
select 1, c1 from t1
select 2, c1 from t1
```

The following statement fragment has a different subquery in the select list, but still matches:

```
select c1, (select c1 from t1 where c2 = 0) from t1
select c1, (select c1 from t1 where c2 = 5) from t1
```

The following statement fragment has a different expression in the select list, but still matches:

```
select c1 + 1
select c1 + 2
```

Example 4: Inexact matching for different clauses

The following statement fragment has different rows for the optimize clause, but still matches:

```
optimize for 1 row
optimize for 10 row
```

The following statement fragment has different rows for the fetch clause, but still matches:

```
fetch first 10 rows only
fetch first 50 rows only
```

The following statement fragment has different literal value for the having clause, but still matches:

```
having c1 > 0
having c1 > 10
```

Each of the following pairs of statement fragments either have different column positioning for the order by clause or have different literal values for the expression in the order by clause, but still match:

```
order by c1+1, c2 + 2, 4
order by c1+2, c2 + 3, 4
```

Each of the following pairs of statement fragments either have different literal values or host variables for the set clause, but still match:

```
set c1 = 1
set c1 = 2

set queryno = 2
set queryno = 3

set querytag = 'query1'
set querytag = 'query2'

SET :HV00001 :HI00001 = <subquery>
SET :HV00009 :HI00009 = <subquery>
```

Each of the following pairs of statement fragments have different literal values for the group by clause, but still match:

```
group by c1 + 1
group by c1 + 2

group by 1,2,3
group by 3,2,1
```

Each of the following pairs of statement fragments have different literal values for the values clause, but still match:

```
values 1,2,3
values 3,4,5

values ( 10, 'xml', 'a' )
values ( 20, 'xml', 'ab' )
```

Example 5: Inexact matching for non-XML functions

Each of the following pairs of statement fragments have different literal values in the function call, but have the same number of literals and still match:

```
decimal(c1, 5, 2)
decimal(c1, 9, 3)

Blob('%abc%')
Blob('cde%')

max(1, 100, 200)
max(INFINITY, NAN, SNAN)
```

Example 6: Inexact matching for special expressions

Each of the following pairs of statement fragments have different literal values in either the case when expression or the mod expression, but still match:

```
order by mod(c1, 2)
order by mod(c1, 4)

case when b1 < 10 then 'A' else 'B' end
case when b1 < 20 then 'C' else 'D' end
```

Example 7: Inexact matching for the CAST function

In Db2 Version 10.5 Fix Pack 3 and later fix packs, the **CAST** of a literal, host variable or parameter marker matches statements with only the literal, host variable or parameter marker without the CAST function.

```
CAST( 'a' AS VARCHAR(5) )
CAST( 'b' AS VARCHAR(10) )

CAST( 'a' AS VARCHAR(5) )
'a'

CAST( 'a' AS VARCHAR(5) )
'b'

CAST( :myHostVar AS VARCHAR(5) )
:myHostVar

CAST( C1+1 AS VARCHAR(5) )
C1+2
```

Example 8: Inexact matching for the ORDER BY clause

In Db2 Version 10.5 Fix Pack 3 and later fix packs sequential column numbers are preserved under an ORDER BY clause.

```
ORDER BY C1, UPPER(C2), 3, 1+4
ORDER BY C1, UPPER(C2), 3, 5+6
```

Example 9: Inexact matching for duplicate rows in a VALUES list

In Db2 Version 10.5 Fix Pack 3 and later fix packs, duplicate rows in VALUES list where all columns consist of only one literal, host variable or parameter marker are removed before the statement text is compared.

```
VALUES (1,1,1), (2,2,2), ..., (9,9,9)
VALUES (1,2,3)

VALUES (1+1,1,1), (2,2,2), (3,3+3,3), (4,4,4)
VALUES (5+5,5,5), (6,6+6,6)

VALUES (1,2), (C1,1), (2,2), (3,3)
VALUES (C1,4)

VALUES (C1+1,1), (C2+2,2), (3,3)
VALUES (C1+4,4), (C2+5,5)
```

XML schema for the statement-level OPTGUIDELINES element:

The OPTGUIDELINES element of a statement profile defines the optimization guidelines in effect for the statement that is identified by the associated statement key. It is defined by the type optGuidelinesType.

XML Schema

```
<xs:element name="OPTGUIDELINES" type="optGuidelinesType"/>
<xs:complexType name="optGuidelinesType">
   <xs:sequence>
      <xs:group ref="general request" minOccurs="0" maxOccurs="1"/>
      <xs:choice maxOccurs="unbounded">
         <xs:group ref="rewriteRequest"/>
         <xs:group ref="accessRequest"/>
         <xs:group ref="joinRequest"/>
         <xs:group ref="mqtEnforcementRequest"/>
      </xs:choice>
  </xs:sequence>
</xs:complexType>
```

Description

The optGuidelinesType group defines the set of valid sub-elements of the OPTGUIDELINES element. Each sub-element is interpreted as an optimization

guideline by the Db2 optimizer. Sub-elements can be categorized as either general request elements, rewrite request elements, access request elements, or join request elements.

- *General request elements* are used to specify general optimization guidelines, which can be used to change the optimizer's search space.
- *Rewrite request elements* are used to specify query rewrite optimization guidelines, which can be used to affect the query transformations that are applied when the optimized statement is being determined.
- *Access request elements* and *join request elements* are plan optimization guidelines, which can be used to affect access methods, join methods, and join orders that are used in the execution plan for the optimized statement.
- *MQT enforcement request elements* specify semantically matchable materialized query tables (MQTs) whose use in access plans should be enforced regardless of cost estimates.

Note: Optimization guidelines that are specified in a statement profile take precedence over those that are specified in the global section of an optimization profile.

XML schema for general optimization guidelines:

The generalRequest group defines guidelines that are not specific to a particular phase of the optimization process. The guidelines can be used to change the optimizer search space.

```
<!--****************************************************************************** --> \
<!-- Choices of general request elements.                                         --> \
<!-- REOPT can be used to override the setting of the REOPT bind option.          --> \
<!-- DPFXMLMOVEMENT can be used to affect the optimizer's plan when moving XML documents  --> \
<!-- between database partitions. The allowed value can be REFERENCE or COMBINATION. The  --> \
<!-- default value is NONE.                                                       --> \
<!--****************************************************************************** --> \
<xs:group name="generalRequest">
   <xs:sequence>
      <xs:element name="REOPT" type="reoptType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="DEGREE" type="degreeType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="QRYOPT" type="qryoptType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="RTS" type="rtsType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="DPFXMLMOVEMENT" type="dpfXMLMovementType" minOccurs="0" maxOccurs="1"/>
      <xs:element name="REGISTRY" type="registryType" minOccurs="0" maxOccurs="1"/>
   </xs:sequence>
</xs:group>
```

General optimization guidelines can be specified at both the global and statement levels. The description and syntax of general optimization guideline elements is the same for both global optimization guidelines and statement-level optimization guidelines.

Description

General request elements define general optimization guidelines, which affect the optimization search space. Affecting the optimization search space can affect the applicability of rewrite and cost-based optimization guidelines.

DEGREE requests:

The DEGREE general request element can be used to override the setting of the **DEGREE** bind parameter, the value of the **dft_degree** database configuration parameter, or the result of a previous SET CURRENT DEGREE statement.

The DEGREE general request element is only considered if the instance is configured for intrapartition parallelism; otherwise, a warning is returned. It is defined by the complex type degreeType.

XML Schema

```
<xs:simpleType name="intStringType">
   <xs:union>
      <xs:simpleType>
         <xs:restriction base="xs:integer">
            <xs:minInclusive value="1"></xs:minInclusive>
            <xs:maxInclusive value="32767"></xs:maxInclusive>
         </xs:restriction>
      </xs:simpleType>
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="ANY"/>
            <xs:enumeration value="-1"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:union>
</xs:simpleType>

<xs:complexType name="degreeType">
   <xs:attribute name="VALUE"
      type="intStringType"></xs:attribute>
</xs:complexType>
```

Description

The DEGREE general request element has a required VALUE attribute that specifies the setting of the DEGREE option. The attribute can take an integer value from 1 to 32 767 or the string value -1 or ANY. The value -1 (or ANY) specifies that the degree of parallelism is to be determined by the data server. A value of 1 specifies that the query should not use intrapartition parallelism.

DPFXMLMOVEMENT requests:

The DPFXMLMOVEMENT general request element can be used in partitioned database environments to override the optimizer's decision to choose a plan in which either a column of type XML is moved or only a reference to that column is moved between database partitions. It is defined by the complex type dpfXMLMovementType.

```
<xs:complexType name="dpfXMLMovementType">
   <xs:attribute name="VALUE" use="required">
      <xs:simpleType>
         <xs:restriction base="xs:string"
            <xs:enumeration value="REFERENCE"/>
            <xs:enumeration value="COMBINATION"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:attribute>
</xs:complexType>
```

Description

In partitioned database environments, data must sometimes be moved between database partitions during statement execution. In the case of XML columns, the optimizer can choose to move the actual documents that are contained in those columns or merely a reference to the source documents on the original database partitions.

The DPFXMLMOVEMENT general request element has a required VALUE attribute with the following possible values: REFERENCE or COMBINATION. If a row that contains an XML column needs to be moved from one database partition to another:

- REFERENCE specifies that references to the XML documents are to be moved through the table queue (TQ) operator. The documents themselves remain on the source database partition.
- COMBINATION specifies that some XML documents are moved, and that only references to the remaining XML documents are moved through the TQ operator.

The decision of whether the documents or merely references to those documents are moved depends on the conditions that prevail when the query runs. If the DPFXMLMOVEMENT general request element has not been specified, the optimizer makes cost-based decisions that are intended to maximize performance.

QRYOPT requests:

The QRYOPT general request element can be used to override the setting of the **QUERYOPT** bind parameter, the value of the **dft_queryopt** database configuration parameter, or the result of a previous SET CURRENT QUERY OPTIMIZATION statement. It is defined by the complex type qryoptType.

XML Schema

```
<xs:complexType name="qryoptType">
   <xs:attribute name="VALUE" use="required">
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="0"/>
            <xs:enumeration value="1"/>
            <xs:enumeration value="2"/>
            <xs:enumeration value="3"/>
            <xs:enumeration value="5"/>
            <xs:enumeration value="7"/>
            <xs:enumeration value="9"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:attribute>
</xs:complexType>
```

Description

The QRYOPT general request element has a required VALUE attribute that specifies the setting of the QUERYOPT option. The attribute can take any of the following values: 0, 1, 2, 3, 5, 7, or 9. For detailed information about what these values represent, see “Optimization classes”.

REOPT requests:

The REOPT general request element can be used to override the setting of the **REOPT** bind parameter, which affects the optimization of statements that contain parameter markers or host variables. It is defined by the complex type reoptType.

XML Schema

```
<xs:complexType name="reoptType">
   <xs:attribute name="VALUE" use="required">
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="ONCE"/>
```

```
            <xs:enumeration value="ALWAYS"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
   </xs:complexType>
```

Description

The REOPT general request element has a required VALUE attribute that specifies the setting of the REOPT option. The attribute can take the value ONCE or ALWAYS. ONCE specifies that the statement should be optimized for the first set of host variable or parameter marker values. ALWAYS specifies that the statement should be optimized for each set of host variable or parameter marker values.

REGISTRY requests:

The REGISTRY element is used to set registry variables in the optimization profile. Embedded in the REGISTRY element is the OPTION element where the registry variable is set.

The REGISTRY element is defined by the complex type registryType and the OPTION element is defined by the complex type genericOptionType.

XML Schema

```
   <xs:complexType name="registryType">
      <xs:sequence>
         <xs:element name="OPTION" type="genericOptionType"
             minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="genericOptionType">
      <xs:attribute name="NAME" type="xs:string" use="required"/>
      <xs:attribute name="VALUE" type="xs:string" use="required"/>
   </xs:complexType>
```

Description

The REGISTRY element sets registry variables at the global level, statement level, or both. The OPTION element which is embedded in the REGISTRY element has a NAME and VALUE attribute. These attributes specify the registry variable name and value that are applied to the profile or to specific statements in the profile.

RTS requests:

The RTS general request element can be used to enable or disable real-time statistics collection. It can also be used to limit the amount of time taken by real-time statistics collection.

For certain queries or workloads, it might be good practice to limit real-time statistics collection so that extra overhead at statement compilation time can be avoided. The RTS general request element is defined by the complex type rtsType.

```
<!--*****************************************************************************************--> \
<!-- RTS general request element to enable, disable or provide a time budget for             --> \
<!-- real-time statistics collection.                                                        --> \
<!-- OPTION attribute allows enabling or disabling real-time statistics.                     --> \
<!-- TIME attribute provides a time budget in milliseconds for real-time statistics collection.--> \
<!--***************************************************************************************** --> \
<xs:complexType name="rtsType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
   <xs:attribute name="TIME" type="xs:nonNegativeInteger" use="optional"/>
</xs:complexType>
```

Description

The RTS general request element has two optional attributes.

- The OPTION attribute is used to enable or disable real-time statistics collection. It can take the value ENABLE (default) or DISABLE.
- The TIME attribute specifies the maximum amount of time (in milliseconds) that can be spent (per statement) on real-time statistics collection at statement compilation time.

If ENABLE is specified for the OPTION attribute, automatic statistics collection and real-time statistics must be enabled through their corresponding configuration parameters. Otherwise, the optimization guideline will not be applied, and SQL0437W with reason code 13 is returned.

XML schema for query rewrite optimization guidelines:

The rewriteRequest group defines guidelines that impact the query rewrite phase of the optimization process.

```
XML Schema

   <xs:group name="rewriteRequest">
      <xs:sequence>
         <xs:element name="INLIST2JOIN" type="inListToJoinType" minOccurs="0"/>
         <xs:element name="SUBQ2JOIN"   type="subqueryToJoinType" minOccurs="0"/>
         <xs:element name="NOTEX2AJ"    type="notExistsToAntiJoinType" minOccurs="0"/>
         <xs:element name="NOTIN2AJ"    type="notInToAntiJoinType" minOccurs="0"/>
      </xs:sequence>
   </xs:group>
```

Description

If the INLIST2JOIN element is used to specify both statement-level and predicate-level optimization guidelines, the predicate-level guidelines override the statement-level guidelines.

IN-LIST-to-join query rewrite requests:

A INLIST2JOIN query rewrite request element can be used to enable or disable the IN-LIST predicate-to-join rewrite transformation. It can be specified as a statement-level optimization guideline or a predicate-level optimization guideline. In the latter case, only one guideline per query can be enabled. The INLIST2JOIN request element is defined by the complex type inListToJoinType.

```
XML Schema

   <xs:complexType name="inListToJoinType">
      <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
      <xs:attribute name="TABLE"  type="xs:string"  use="optional"/>
      <xs:attribute name="COLUMN" type="xs:string"  use="optional"/>
   </xs:complexType>
```

Description

The INLIST2JOIN query rewrite request element has three optional attributes and no sub-elements. The OPTION attribute can take the value ENABLE (default) or DISABLE. The TABLE and COLUMN attributes are used to specify an IN-LIST predicate. If these attributes are not specified, or are specified with an empty string (“”) value, the guideline is handled as a statement-level guideline. If one or both of these attributes are specified, it is handled as a predicate-level guideline. If the TABLE attribute is not specified, or is specified with an empty string value, but the COLUMN attribute is specified, the optimization guideline is ignored and SQL0437W with reason code 13 is returned.

NOT-EXISTS-to-anti-join query rewrite requests:

The NOTEX2AJ query rewrite request element can be used to enable or disable the NOT-EXISTS predicate-to-anti-join rewrite transformation. It can be specified as a statement-level optimization guideline only. The NOTEX2AJ request element is defined by the complex type notExistsToAntiJoinType.

XML Schema

```
<xs:complexType name="notExistsToAntiJoinType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
</xs:complexType>
```

Description

The NOTEX2AJ query rewrite request element has one optional attribute and no sub-elements. The OPTION attribute can take the value ENABLE (default) or DISABLE.

NOT-IN-to-anti-join query rewrite requests:

The NOTIN2AJ query rewrite request element can be used to enable or disable the NOT-IN predicate-to-anti-join rewrite transformation. It can be specified as a statement-level optimization guideline only. The NOTIN2AJ request element is defined by the complex type notInToAntiJoinType.

XML Schema

```
<xs:complexType name="notInToAntiJoinType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
</xs:complexType>
```

Description

The NOTIN2AJ query rewrite request element has one optional attribute and no sub-elements. The OPTION attribute can take the value ENABLE (default) or DISABLE.

Subquery-to-join query rewrite requests:

The SUBQ2JOIN query rewrite request element can be used to enable or disable the subquery-to-join rewrite transformation. It can be specified as a statement-level optimization guideline only. The SUBQ2JOIN request element is defined by the complex type subqueryToJoinType.

XML Schema

```
<xs:complexType name="subqueryToJoinType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
</xs:complexType>
```

Description

The SUBQ2JOIN query rewrite request element has one optional attribute and no sub-elements. The OPTION attribute can take the value ENABLE (default) or DISABLE.

XML schema for plan optimization guidelines:

Plan optimization guidelines can consist of access requests or join requests.

- An *access request* specifies an access method for a table reference.

- A *join request* specifies a method and sequence for performing a join operation. Join requests are composed of other access or join requests.

Most of the available access requests correspond to the optimizer's data access methods, such as table scan, index scan, and list prefetch, for example. Most of the available join requests correspond to the optimizer's join methods, such as nested-loop join, hash join, and merge join. Each access request or join request element can be used to influence plan optimization.

Access requests:

The accessRequest group defines the set of valid access request elements. An access request specifies an access method for a table reference.

XML Schema

```
<xs:group name="accessRequest">
   <xs:choice>
      <xs:element name="TBSCAN" type="tableScanType"/>
      <xs:element name="IXSCAN" type="indexScanType"/>
      <xs:element name="LPREFETCH" type="listPrefetchType"/>
      <xs:element name="IXAND" type="indexAndingType"/>
      <xs:element name="IXOR" type="indexOringType"/>
      <xs:element name="XISCAN" type="indexScanType"/>
      <xs:element name="XANDOR" type="XANDORType"/>
      <xs:element name="ACCESS" type="anyAccessType"/>
   </xs:choice>
</xs:group>
```

Description

- TBSCAN, IXSCAN, LPREFETCH, IXAND, IXOR, XISCAN, and XANDOR

  These elements correspond to Db2 data access methods, and can only be applied to local tables that are referenced in a statement. They cannot refer to nicknames (remote tables) or derived tables (the result of a subselect).
- ACCESS

  This element, which causes the optimizer to choose the access method, can be used when the join order (not the access method) is of primary concern. The ACCESS element must be used when the target table reference is a derived table. For XML queries, this element can also be used with attribute TYPE = XMLINDEX to specify that the optimizer is to choose XML index access plans.

Access types:

Common aspects of the TBSCAN, IXSCAN, LPREFETCH, IXAND, IXOR, XISCAN, XANDOR, and ACCESS elements are defined by the abstract type accessType.

XML Schema

```
<xs:complexType name="accessType" abstract="true">
   <xs:attribute name="TABLE" type="xs:string" use="optional"/>
   <xs:attribute name="TABID" type="xs:string" use="optional"/>
   <xs:attribute name="FIRST" type="xs:string" use="optional" fixed="TRUE"/>
   <xs:attribute name="SHARING" type="optionType" use="optional"
     default="ENABLE"/>
   <xs:attribute name="WRAPPING" type="optionType" use="optional"
     default="ENABLE"/>
   <xs:attribute name="THROTTLE" type="optionType" use="optional"/>
   <xs:attribute name="SHARESPEED" type="shareSpeed" use="optional"/>
</xs:complexType>

<xs:complexType name="extendedAccessType">
```

```
    <xs:complexContent>
       <xs:extension base="accessType">
          <xs:sequence minOccurs="0">
             <xs:element name="INDEX" type="indexType" minOccurs="2"
               maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="INDEX" type="xs:string" use="optional"/>
          <xs:attribute name="TYPE" type="xs:string" use="optional"
            fixed="XMLINDEX"/>
          <xs:attribute name="ALLINDEXES" type="boolType" use="optional"
            fixed="TRUE"/>
       </xs:extension>
    </xs:complexContent>
 </xs:complexType>
```

Description

All access request elements extend the complex type accessType. Each such element must specify the target table reference using either the TABLE or TABID attribute. For information on how to form proper table references from an access request element, see “Forming table references in optimization guidelines”.

Access request elements can also specify an optional FIRST attribute. If the FIRST attribute is specified, it must have the value TRUE. Adding the FIRST attribute to an access request element indicates that the execution plan should include the specified table as the first table in the join sequence of the corresponding FROM clause. Only one access or join request per FROM clause can specify the FIRST attribute. If multiple access or join requests targeting tables of the same FROM clause specify the FIRST attribute, all but the first such request is ignored and a warning (SQL0437W with reason code 13) is returned.

New optimizer guidelines enable you to influence the compiler's scan sharing decisions. In cases where the compiler would have allowed sharing scans, wrapping scans, or throttling, specifying the appropriate guideline will prevent sharing scans, wrapping scans, or throttling. A sharing scan can be seen by other scans that are participating in scan sharing, and those scans can base certain decisions on that information. A wrapping scan is able to start at an arbitrary point in the table to take advantage of pages that are already in the buffer pool. A throttled scan has been delayed to increase the overall level of sharing.

Valid optionType values (for the SHARING, WRAPPING, and THROTTLE attributes) are DISABLE and ENABLE (the default). SHARING and WRAPPING cannot be enabled when the compiler chooses to disable them. Using ENABLE will have no effect in those cases. THROTTLE can be either enabled or disabled. Valid SHARESPEED values (to override the compiler's estimate of scan speed) are FAST and SLOW. The default is to allow the compiler to determine values, based on its estimate.

The only supported value for the TYPE attribute is XMLINDEX, which indicates to the optimizer that the table must be accessed using one of the XML index access methods, such as IXAND, IXOR, XANDOR, or XISCAN. If this attribute is not specified, the optimizer makes a cost-based decision when selecting an access plan for the specified table.

The optional INDEX attribute can be used to specify an index name.

The optional INDEX element can be used to specify two or more names of indexes as index elements. If the INDEX attribute and the INDEX element are both specified, the INDEX attribute is ignored.

The optional ALLINDEXES attribute, whose only supported value is TRUE, can only be specified if the TYPE attribute has a value of XMLINDEX. If the ALLINDEXES attribute is specified, the optimizer must use all applicable relational indexes and indexes over XML data to access the specified table, regardless of cost.

Any access requests:

The ACCESS access request element can be used to specify that the optimizer is to choose an appropriate method for accessing a table, based on cost, and must be used when referencing a derived table. A derived table is the result of another subselect. This access request element is defined by the complex type anyAccessType.

```
XML Schema

   <xs:complexType name="anyAccessType">
      <xs:complexContent>
         <xs:extension base="extendedAccessType"/>
      </xs:complexContent>
   </xs:complexType>
```

Description

The complex type anyAccessType is a simple extension of the abstract type extendedAccessType. No new elements or attributes are added.

The TYPE attribute, whose only supported value is XMLINDEX, indicates to the optimizer that the table must be accessed using one of the XML index access methods, such as IXAND, IXOR, XANDOR, or XISCAN. If this attribute is not specified, the optimizer makes a cost-based decision when selecting an access plan for the specified table.

The optional INDEX attribute can be used to specify an index name only if the TYPE attribute has a value of XMLINDEX. If this attribute is specified, the optimizer might choose one of the following plans:

- An XISCAN plan using the specified index over XML data
- An XANDOR plan, such that the specified index over XML data is one of the indexes under XANDOR; the optimizer will use all applicable indexes over XML data in the XANDOR plan
- An IXAND plan, such that the specified index is the leading index of IXAND; the optimizer will add more indexes to the IXAND plan in a cost-based fashion
- A cost-based IXOR plan

The optional INDEX element can be used to specify two or more names of indexes as index elements only if the TYPE attribute has a value of XMLINDEX. If this element is specified, the optimizer might choose one of the following plans:

- An XANDOR plan, such that the specified indexes over XML data appear under XANDOR; the optimizer will use all applicable indexes over XML data in the XANDOR plan
- An IXAND plan, such that the specified indexes are the indexes of IXAND, in the specified order
- A cost-based IXOR plan

If the INDEX attribute and the INDEX element are both specified, the INDEX attribute is ignored.

The optional ALLINDEXES attribute, whose only supported value is TRUE, can only be specified if the TYPE attribute has a value of XMLINDEX. If this attribute is specified, the optimizer must use all applicable relational indexes and indexes over XML data to access the specified table, regardless of cost. The optimizer chooses one of the following plans:

- An XANDOR plan with all applicable indexes over XML data appearing under the XANDOR operator
- An IXAND plan with all applicable relational indexes and indexes over XML data appearing under the IXAND operator
- An IXOR plan
- An XISCAN plan if only a single index is defined on the table and that index is of type XML

Examples

The following guideline is an example of an any access request:

```
<OPTGUIDELINES>
  <HSJOIN>
    <ACCESS TABLE='S1'/>
    <IXSCAN TABLE='PS1'/>
  </HSJOIN>
</OPTGUIDELINES>
```

The following example shows an ACCESS guideline specifying that some XML index access to the SECURITY table should be used. The optimizer might pick any XML index plan, such as an XISCAN, IXAND, XANDOR, or IXOR plan.

```
SELECT * FROM security
  WHERE XMLEXISTS('$SDOC/Security/SecurityInformation/
    StockInformation[Industry= "OfficeSupplies"]')

<OPTGUIDELINES>
  <ACCESS TABLE='SECURITY' TYPE='XMLINDEX'/>
</OPTGUIDELINES>
```

The following example shows an ACCESS guideline specifying that all possible index access to the SECURITY table should be used. The choice of method is left to the optimizer. Assume that two XML indexes, SEC_INDUSTRY and SEC_SYMBOL, match the two XML predicates. The optimizer chooses either the XANDOR or the IXAND access method using a cost-based decision.

```
SELECT * FROM security
  WHERE XMLEXISTS('$SDOC/Security/SecurityInformation/
      StockInformation[Industry= "Software"]') AND
    XMLEXISTS('$SDOC/Security/Symbol[.="IBM"]')

<OPTGUIDELINES>
  <ACCESS TABLE='SECURITY' TYPE='XMLINDEX' ALLINDEXES='TRUE'/>
</OPTGUIDELINES>
```

The following example shows an ACCESS guideline specifying that the SECURITY table should be accessed using at least the SEC_INDUSTRY XML index. The optimizer chooses one of the following access plans in a cost-based fashion:

- An XISCAN plan using the SEC_INDUSTRY XML index
- An IXAND plan with the SEC_INDUSTRY index as the first leg of the IXAND. The optimizer is free to use more relational or XML indexes in the IXAND plan

following cost-based analysis. If a relational index were available on the TRANS_DATE column, for example, that index might appear as an additional leg of the IXAND if that were deemed to be beneficial by the optimizer.

- A XANDOR plan using the SEC_INDUSTRY index and other applicable XML indexes

```
SELECT * FROM security
  WHERE trans_date = CURRENT DATE AND
    XMLEXISTS('$SDOC/Security/SecurityInformation/
      StockInformation[Industry= "Software"]') AND
    XMLEXISTS('$SDOC/Security/Symbol[.="IBM"]')

<OPTGUIDELINES>
  <ACCESS TABLE='SECURITY' TYPE='XMLINDEX' INDEX='SEC_INDUSTRY'/>
</OPTGUIDELINES>
```

Index ANDing access requests:

The IXAND access request element can be used to specify that the optimizer is to use the index ANDing data access method to access a local table. It is defined by the complex type indexAndingType.

The IXAND access request element can be used to specify that the optimizer is to use the index ANDing data access method to access a local table. It is defined by the complex type indexAndingType.

XML Schema

```
<xs:complexType name="indexAndingType">
   <xs:complexContent>
      <xs:extension base="extendedAccessType">
         <xs:sequence minOccurs="0">
            <xs:element name="NLJOIN" type="nestedLoopJoinType" minOccurs="1"
              maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute name="STARJOIN" type="boolType" use="optional"/>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

Description

The complex type indexAndingType is an extension of extendedAccessType. When the STARJOIN attribute and NLJOIN elements are not specified, indexAndingType becomes a simple extension of extendedAccessType. The extendedAccessType type extends the abstract type accessType by adding an optional INDEX attribute, optional INDEX sub-elements, an optional TYPE attribute, and an optional ALLINDEXES attribute. The INDEX attribute can be used to specify the first index that is to be used in the index ANDing operation. If the INDEX attribute is used, the optimizer chooses additional indexes and the access sequence in a cost-based fashion. The INDEX sub-elements can be used to specify the exact set of indexes and access sequence. The order in which the INDEX sub-elements appear indicates the order in which the individual index scans should be performed. The specification of INDEX sub-elements supersedes the specification of the INDEX attribute.

- If no indexes are specified, the optimizer chooses both the indexes and the access sequence in a cost-based fashion.
- If indexes are specified using either the attribute or sub-elements, these indexes must be defined on the table that is identified by the TABLE or TABID attribute.
- If there are no indexes defined on the table, the access request is ignored and an error is returned.

The TYPE attribute, whose only supported value is XMLINDEX, indicates to the optimizer that the table must be accessed using one or more indexes over XML data.

The optional INDEX attribute can be used to specify an XML index name only if the TYPE attribute has a value of XMLINDEX. A relational index can be specified in the optional INDEX attribute regardless of the TYPE attribute specification. The specified index is used by the optimizer as the leading index of an IXAND plan. The optimizer will add more indexes to the IXAND plan in a cost-based fashion.

The optional INDEX element can be used to specify two or more names of indexes over XML data as index elements only if the TYPE attribute has a value of XMLINDEX. Relational indexes can be specified in the optional INDEX elements regardless of the TYPE attribute specification. The specified indexes are used by the optimizer as the indexes of an IXAND plan in the specified order.

If the TYPE attribute is not present, INDEX attributes and INDEX elements are still valid for relational indexes.

If the INDEX attribute and the INDEX element are both specified, the INDEX attribute is ignored.

The optional ALLINDEXES attribute, whose only supported value is TRUE, can only be specified if the TYPE attribute has a value of XMLINDEX. If this attribute is specified, the optimizer must use all applicable relational indexes and indexes over XML data in an IXAND plan to access the specified table, regardless of cost.

If the TYPE attribute is specified, but neither INDEX attribute, INDEX element, nor ALLINDEXES attribute is specified, the optimizer will choose an IXAND plan with at least one index over XML data. Other indexes in the plan can be either relational indexes or indexes over XML data. The order and choice of indexes is determined by the optimizer in a cost-based fashion.

Block indexes must appear before record indexes in an index ANDing access request. If this requirement is not met, an error is returned. The index ANDing access method requires that at least one predicate is able to be indexed for each index. If index ANDing is not eligible because the required predicate does not exist, the access request is ignored and an error is returned. If the index ANDing data access method is not in the search space that is in effect for the statement, the access request is ignored and an error is returned.

You can use the IXAND access request element to request a star join index ANDing plan. The optional STARJOIN attribute on the IXAND element specifies that the IXAND is for a star join index ANDing plan. NLJOINs can be sub-elements of the IXAND, and must be properly constructed star join semi-joins. `STARJOIN="FALSE"` specifies a request for a regular base access index ANDing plan. `STARJOIN="TRUE"` specifies a request for a star join index ANDing plan. The default value is determined by context: If the IXAND has one or more semi-join child elements, the default is TRUE; otherwise, the default is FALSE. If `STARJOIN="TRUE"` is specified:

- The INDEX, TYPE, and ALLINDEXES attributes cannot be specified
- INDEX elements cannot be specified

If NLJOIN elements are specified:

- The INDEX, TYPE, and ALLINDEXES attributes cannot be specified

- INDEX elements cannot be specified
- The only supported value for the STARJOIN attribute is TRUE

The following example illustrates an index ANDing access request:

```
SQL statement:

   select s.s_name, s.s_address, s.s_phone, s.s_comment
     from "Samp".parts, "Samp".suppliers s, "Samp".partsupp ps
     where p_partkey = ps.ps_partkey and
       s.s_suppkey = ps.ps_suppkey and
       p_size = 39 and
       p_type = 'BRASS' and
       s.s_nation in ('MOROCCO', 'SPAIN') and
       ps.ps_supplycost = (select min(ps1.ps_supplycost)
                             from "Samp".partsupp ps1, "Samp".suppliers s1
                             where "Samp".parts.p_partkey = ps1.ps_partkey and
                               s1.s_suppkey = ps1.ps_suppkey and
                               s1.s_nation = s.s_nation)
     order by s.s_name
     optimize for 1 row

Optimization guideline:

   <OPTGUIDELINES>
     <IXAND TABLE='"Samp".PARTS' FIRST='TRUE'>
       <INDEX IXNAME='ISIZE'/>
       <INDEX IXNAME='ITYPE'/>
     </IXAND>
   </OPTGUIDELINES>
```

The index ANDing request specifies that the PARTS table in the main subselect is to be satisfied using an index ANDing data access method. The first index scan will use the ISIZE index, and the second index scan will use the ITYPE index. The indexes are specified by the IXNAME attribute of the INDEX element. The FIRST attribute setting specifies that the PARTS table is to be the first table in the join sequence with the SUPPLIERS, PARTSUPP, and derived tables in the same FROM clause.

The following example illustrates a star join index ANDing guideline that specifies the first semi-join but lets the optimizer choose the remaining ones. It also lets the optimizer choose the specific access method for the outer table and the index for the inner table in the specified semi-join.

```
<IXAND TABLE="F">
  <NLJOIN>
    <ACCESS TABLE="D1"/>
    <IXSCAN TABLE="F"/>
  </NLJOIN>
</IXAND>
```

The following guideline specifies all of the semi-joins, including details, leaving the optimizer with no choices for the plan at and after the IXAND.

```
<IXAND TABLE="F" STARJOIN="TRUE">
  <NLJOIN>
    <TBSCAN TABLE="D1"/>
    <IXSCAN TABLE="F" INDEX="FX1"/>
  </NLJOIN>
  <NLJOIN>
    <TBSCAN TABLE="D4"/>
    <IXSCAN TABLE="F" INDEX="FX4"/>
  </NLJOIN>
  <NLJOIN>
```

```
      <TBSCAN TABLE="D3"/>
      <IXSCAN TABLE="F" INDEX="FX3"/>
    </NLJOIN>
  </IXAND>
```

Index ORing access requests:

The IXOR access request element can be used to specify that the optimizer is to use the index ORing data access method to access a local table. It is defined by the complex type indexOringType.

The IXOR access request element can be used to specify that the optimizer is to use the index ORing data access method to access a local table. It is defined by the complex type indexOringType.

XML Schema

```
  <xs:complexType name="indexOringType">
     <xs:complexContent>
        <xs:extension base="accessType"/>
     </xs:complexContent>
  </xs:complexType>
```

Description

The complex type indexOringType is a simple extension of the abstract type accessType. No new elements or attributes are added. If the index ORing access method is not in the search space that is in effect for the statement, the access request is ignored and SQL0437W with reason code 13 is returned. The optimizer chooses the predicates and indexes that are used in the index ORing operation in a cost-based fashion. The index ORing access method requires that at least one IN predicate is able to be indexed or that a predicate with terms is able to be indexed and connected by a logical OR operation. If index ORing is not eligible because the required predicate or indexes do not exist, the request is ignored and SQL0437W with reason code 13 is returned.

The following example illustrates an index ORing access request:

SQL statement:

```
  select s.s_name, s.s_address, s.s_phone, s.s_comment
    from "Samp".parts, "Samp".suppliers s, "Samp".partsupp ps
    where p_partkey = ps.ps_partkey and
      s.s_suppkey = ps.ps_suppkey and
      p_size = 39 and
      p_type = 'BRASS' and
      s.s_nation in ('MOROCCO', 'SPAIN') and
      ps.ps_supplycost = (select min(ps1.ps_supplycost)
                            from "Samp".partsupp ps1, "Samp".suppliers s1
                            where "Samp".parts.p_partkey = ps1.ps_partkey and
                              s1.s_suppkey = ps1.ps_suppkey and
                              s1.s_nation = s.s_nation)
    order by s.s_name
    optimize for 1 row

Optimization guideline:

  <OPTGUIDELINES>
    <IXOR TABLE='S'/>
  </OPTGUIDELINES>
```

This index ORing access request specifies that an index ORing data access method is to be used to access the SUPPLIERS table that is referenced in the main

subselect. The optimizer will choose the appropriate predicates and indexes for the index ORing operation in a cost-based fashion.

Index scan access requests:

The IXSCAN access request element can be used to specify that the optimizer is to use an index scan to access a local table. It is defined by the complex type indexScanType.

XML Schema

```
<xs:complexType name="indexScanType">
   <xs:complexContent>
      <xs:extension base="accessType">
         <xs:attribute name="INDEX" type="xs:string" use="optional"/>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

Description

The complex type indexScanType extends the abstract accessType by adding an optional INDEX attribute. The INDEX attribute specifies the unqualified name of the index that is to be used to access the table.

- If the index scan access method is not in the search space that is in effect for the statement, the access request is ignored and SQL0437W with reason code 13 is returned.
- If the INDEX attribute is specified, it must identify an index defined on the table that is identified by the TABLE or TABID attribute. If the index does not exist, the access request is ignored and SQL0437W with reason code 13 is returned.
- If the INDEX attribute is not specified, the optimizer chooses an index in a cost-based fashion. If no indexes are defined on the target table, the access request is ignored and SQL0437W with reason code 13 is returned.

The following guideline is an example of an index scan access request:

```
<OPTGUIDELINES>
  <IXSCAN TABLE='S' INDEX='I_SUPPKEY'/>
</OPTGUIDELINES>
```

List prefetch access requests:

The LPREFETCH access request element can be used to specify that the optimizer is to use a list prefetch index scan to access a local table. It is defined by the complex type listPrefetchType.

XML Schema

```
<xs:complexType name="listPrefetchType">
   <xs:complexContent>
      <xs:extension base="accessType">
         <xs:attribute name="INDEX" type="xs:string" use="optional"/>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

Description

The complex type listPrefetchType extends the abstract type accessType by adding an optional INDEX attribute. The INDEX attribute specifies the name of the index that is to be used to access the table.

- If the list prefetch access method is not in the search space that is in effect for the statement, the access request is ignored and SQL0437W with reason code 13 is returned.
- The list prefetch access method requires that at least one predicate is able to be indexed. If the list prefetch access method is not eligible because the required predicate does not exist, the access request is ignored and SQL0437W with reason code 13 is returned.
- If the INDEX attribute is specified, it must identify an index defined on the table that is specified by the TABLE or TABID attribute. If the index does not exist, the access request is ignored and SQL0437W with reason code 13 is returned.
- If the INDEX attribute is not specified, the optimizer chooses an index in a cost-based fashion. If no indexes are defined on the target table, the access request is ignored and SQL0437W with reason code 13 is returned.

The following guideline is an example of a list prefetch access request:

```
<OPTGUIDELINES>
  <LPREFETCH TABLE='S1' INDEX='I_SNATION'/>
</OPTGUIDELINES>
```

Table scan access requests:

The TBSCAN access request element can be used to specify that the optimizer is to use a sequential table scan to access a local table. It is defined by the complex type tableScanType.

XML Schema

```
<xs:complexType name="tableScanType">
   <xs:complexContent>
      <xs:extension base="accessType"/>
   </xs:complexContent>
</xs:complexType>
```

Description

The complex type tableScanType is a simple extension of the abstract type accessType. No new elements or attributes are added. If the table scan access method is not in the search space that is in effect for the statement, the access request is ignored and SQL0437W with reason code 13 is returned.

The following guideline is an example of a table scan access request:

```
<OPTGUIDELINES>
  <TBSCAN TABLE='S1'/>
</OPTGUIDELINES>
```

XML index ANDing and ORing access requests:

The XANDOR access request element can be used to specify that the optimizer is to use multiple XANDORed index over XML data scans to access a local table. It is defined by the complex type XANDORType.

XML Schema

```
<xs:complexType name="XANDORType">
   <xs:complexContent>
      <xs:extension base="accessType"/>
   </xs:complexContent>
</xs:complexType>
```

Description

The complex type XANDORType is a simple extension of the abstract type accessType. No new elements or attributes are added.

Example

Given the following query:

```
SELECT * FROM security
  WHERE trans_date = CURRENT DATE AND
    XMLEXISTS('$SDOC/Security/SecurityInformation/
      StockInformation[Industry = "Software"]') AND
    XMLEXISTS('$SDOC/Security/Symbol[.="IBM"]')
```

The following XANDOR guideline specifies that the SECURITY table should be accessed using a XANDOR operation against all applicable XML indexes. Any relational indexes on the SECURITY table will not be considered, because a relational index cannot be used with a XANDOR operator.

```
<OPTGUIDELINES>
  <XANDOR TABLE='SECURITY'/>
</OPTGUIDELINES>
```

XML index scan access requests:

The XISCAN access request element can be used to specify that the optimizer is to use an index over XML data scan to access a local table. It is defined by the complex type indexScanType.

```
XML Schema

   <xs:complexType name="indexScanType">
      <xs:complexContent>
         <xs:extension base="accessType"/>
            <xs:attribute name="INDEX" type="xs:string" use="optional"/>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
```

Description

The complex type indexScanType extends the abstract accessType by adding an optional INDEX attribute. The INDEX attribute specifies the name of the index over XML data that is to be used to access the table.

- If the index over XML data scan access method is not in the search space that is in effect for the statement, the access request is ignored and SQL0437W with reason code 13 is returned.
- If the INDEX attribute is specified, it must identify an index over XML data defined on the table that is identified by the TABLE or TABID attribute. If the index does not exist, the access request is ignored and SQL0437W with reason code 13 is returned.
- If the INDEX attribute is not specified, the optimizer chooses an index over XML data in a cost-based fashion. If no indexes over XML data are defined on the target table, the access request is ignored and SQL0437W with reason code 13 is returned.

Example

Given the following query:

```
SELECT * FROM security
  WHERE XMLEXISTS('$SDOC/Security/SecurityInformation/
    StockInformation[Industry = "OfficeSupplies"]')
```

The following XISCAN guideline specifies that the SECURITY table should be accessed using an XML index named SEC_INDUSTRY.

```
<OPTGUIDELINES>
  <XISCAN TABLE='SECURITY' INDEX='SEC_INDUSTRY'/>
</OPTGUIDELINES>
```

Join requests:

The joinRequest group defines the set of valid join request elements. A join request specifies a method for joining two tables.

XML Schema

```
<xs:group name="joinRequest">
   <xs:choice>
      <xs:element name="NLJOIN" type="nestedLoopJoinType"/>
      <xs:element name="HSJOIN" type="hashJoinType"/>
      <xs:element name="MSJOIN" type="mergeJoinType"/>
      <xs:element name="JOIN" type="anyJoinType"/>
   </xs:choice>
</xs:group>
```

Description

- NLJOIN, MSJOIN, and HSJOIN

  These elements correspond to the nested-loop, merge, and hash join methods, respectively.
- JOIN

  This element, which causes the optimizer to choose the join method, can be used when the join order is not of primary concern.

All join request elements contain two sub-elements that represent the input tables of the join operation. Join requests can also specify an optional FIRST attribute.

The following guideline is an example of a join request:

```
<OPTGUIDELINES>
  <HSJOIN>
    <ACCESS TABLE='S1'/>
    <IXSCAN TABLE='PS1'/>
  </HSJOIN>
</OPTGUIDELINES>
```

The nesting order ultimately determines the join order. The following example illustrates how larger join requests can be constructed from smaller join requests:

```
<OPTGUIDELINES>
  <MSJOIN>
    <NLJOIN>
      <IXSCAN TABLE='"Samp".Parts'/>
      <IXSCAN TABLE="PS"/>
    </NLJOIN>
    <IXSCAN TABLE='S'/>
  </MSJOIN>
</OPTGUIDELINES>
```

Join types:

Common aspects of all join request elements are defined by the abstract type joinType.

XML Schema

```
<xs:complexType name="joinType" abstract="true">
   <xs:choice minOccurs="2" maxOccurs="2">
      <xs:group ref="accessRequest"/>
      <xs:group ref="joinRequest"/>
   </xs:choice>
   <xs:attribute name="FIRST" type="xs:string" use="optional" fixed="TRUE"/>
</xs:complexType>
```

Description

Join request elements that extend the complex type joinType must have exactly two sub-elements. Either sub-element can be an access request element chosen from the accessRequest group, or another join request element chosen from the joinRequest group. The first sub-element appearing in the join request specifies the outer table of the join operation, and the second element specifies the inner table.

If the FIRST attribute is specified, it must have the value TRUE. Adding the FIRST attribute to a join request element indicates that you want an execution plan in which the tables that are targeted by the join request are the outermost tables in the join sequence for the corresponding FROM clause. Only one access or join request per FROM clause can specify the FIRST attribute. If multiple access or join requests that target tables of the same FROM clause specify the FIRST attribute, all but the initial request are ignored and SQL0437W with reason code 13 is returned.

Any join requests:

The JOIN join request element can be used to specify that the optimizer is to choose an appropriate method for joining two tables in a particular order.

Either table can be local or derived, as specified by an access request sub-element, or it can be the result of a join operation, as specified by a join request sub-element. A derived table is the result of another subselect. This join request element is defined by the complex type anyJoinType.

XML Schema

```
<xs:complexType name="anyJoinType">
   <xs:complexContent>
      <xs:extension base="joinType"/>
   </xs:complexContent>
</xs:complexType>
```

Description

The complex type anyJoinType is a simple extension of the abstract type joinType. No new elements or attributes are added.

The following example illustrates the use of the JOIN join request element to force a particular join order for a set of tables:

SQL statement:

```
select s.s_name, s.s_address, s.s_phone, s.s_comment
   from "Samp".parts, "Samp".suppliers s, "Samp".partsupp ps
   where p_partkey = ps.ps_partkey and
```

```
      s.s_suppkey = ps.ps_suppkey and
      p_size = 39 and
      p_type = 'BRASS' and
      s.s_nation in ('MOROCCO', 'SPAIN') and
      ps.ps_supplycost = (select min(ps1.ps_supplycost)
                          from "Samp".partsupp ps1, "Samp".suppliers s1
                          where "Samp".parts.p_partkey = ps1.ps_partkey and
                            s1.s_suppkey = ps1.ps_suppkey and
                            s1.s_nation = s.s_nation)
    order by s.s_name

Optimization guideline:

  <OPTGUIDELINES>
    <JOIN>
      <JOIN>
        <ACCESS TABLE='Samp".PARTS'/>
        <ACCESS TABLE='S'/>
      </JOIN>
      <ACCESS TABLE='PS'>
    </JOIN>
  </OPTGUIDELINES>
```

The JOIN join request elements specify that the PARTS table in the main subselect is to be joined with the SUPPLIERS table, and that this result is to be joined to the PARTSUPP table. The optimizer will choose the join methods for this particular sequence of joins in a cost-based fashion.

Hash join requests:

The HSJOIN join request element can be used to specify that the optimizer is to join two tables using a hash join method.

Either table can be local or derived, as specified by an access request sub-element, or it can be the result of a join operation, as specified by a join request sub-element. A derived table is the result of another subselect. This join request element is defined by the complex type hashJoinType.

XML Schema

```
  <xs:complexType name="hashJoinType">
     <xs:complexContent>
        <xs:extension base="joinType"/>
     </xs:complexContent>
  </xs:complexType>
```

Description

The complex type hashJoinType is a simple extension of the abstract type joinType. No new elements or attributes are added. If the hash join method is not in the search space that is in effect for the statement, the join request is ignored and SQL0437W with reason code 13 is returned.

The following guideline is an example of a hash join request:

```
  <OPTGUIDELINES>
    <HSJOIN>
      <ACCESS TABLE='S1'/>
      <IXSCAN TABLE='PS1'/>
    </HSJOIN>
  </OPTGUIDELINES>
```

Merge join requests:

The MSJOIN join request element can be used to specify that the optimizer is to join two tables using a merge join method.

Either table can be local or derived, as specified by an access request sub-element, or it can be the result of a join operation, as specified by a join request sub-element. A derived table is the result of another subselect. This join request element is defined by the complex type mergeJoinType.

XML Schema

```
<xs:complexType name="mergeJoinType">
   <xs:complexContent>
      <xs:extension base="joinType"/>
   </xs:complexContent>
</xs:complexType>
```

Description

The complex type mergeJoinType is a simple extension of the abstract type joinType. No new elements or attributes are added. If the merge join method is not in the search space that is in effect for the statement, the join request is ignored and SQL0437W with reason code 13 is returned.

The following guideline is an example of a merge join request:

```
<OPTGUIDELINES>
  <MSJOIN>
    <NLJOIN>
      <IXSCAN TABLE='"Samp".Parts'/>
      <IXSCAN TABLE="PS"/>
    </NLJOIN>
    <IXSCAN TABLE='S'/>
  </MSJOIN>
</OPTGUIDELINES>
```

Nested-loop join requests:

The NLJOIN join request element can be used to specify that the optimizer is to join two tables using a nested-loop join method.

Either table can be local or derived, as specified by an access request sub-element, or it can be the result of a join operation, as specified by a join request sub-element. A derived table is the result of another subselect. This join request element is defined by the complex type nestedLoopJoinType.

XML Schema

```
<xs:complexType name="nestedLoopJoinType">
   <xs:complexContent>
      <xs:extension base="joinType"/>
   </xs:complexContent>
</xs:complexType>
```

Description

The complex type nestedLoopJoinType is a simple extension of the abstract type joinType. No new elements or attributes are added. If the nested-loop join method is not in the search space that is in effect for the statement, the join request is ignored and SQL0437W with reason code 13 is returned.

The following guideline is an example of a nested-loop join request:

```
<OPTGUIDELINES>
  <NLJOIN>
    <IXSCAN TABLE='"Samp".Parts'/>
    <IXSCAN TABLE="PS"/>
  </NLJOIN>
</OPTGUIDELINES>
```

SYSTOOLS.OPT_PROFILE table:

The SYSTOOLS.OPT_PROFILE table contains all of the optimization profiles.

There are two methods to create this table:

- Call the SYSINSTALLOBJECTS procedure:

  ```
  db2 "call sysinstallobjects('opt_profiles', 'c', '', '')"
  ```

- Issue the CREATE TABLE statement:

  ```
  create table systools.opt_profile (
    schema  varchar(128) not null,
    name    varchar(128) not null,
    profile blob (2m)    not null,
    primary key (schema, name)
  )
  ```

The columns in the SYSTOOLS.OPT_PROFILE table are defined as follows:

SCHEMA
Specifies the schema name for an optimization profile. The name can include up to 30 alphanumeric or underscore characters, but define it as VARCHAR(128), as shown.

NAME
Specifies the base name for an optimization profile. The name can include up to 128 alphanumeric or underscore characters.

PROFILE
Specifies an XML document that defines the optimization profile.

Triggers to flush the optimization profile cache:

The optimization profile cache is automatically flushed whenever an entry in the SYSTOOLS.OPT_PROFILE table is updated or deleted.

The following SQL procedure and triggers must be created before automatic flushing of the profile cache can occur.

```
CREATE PROCEDURE SYSTOOLS.OPT_FLUSH_CACHE( IN SCHEMA VARCHAR(128),
                                           IN NAME VARCHAR(128) )
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN ATOMIC
 -- FLUSH stmt (33) + quoted schema (130) + dot (1) + quoted name (130) = 294
 DECLARE FSTMT VARCHAR(294) DEFAULT 'FLUSH OPTIMIZATION PROFILE CACHE '; --

 IF NAME IS NOT NULL THEN
   IF SCHEMA IS NOT NULL THEN
     SET FSTMT = FSTMT || '"' || SCHEMA || '".'; --
   END IF; --

   SET FSTMT = FSTMT || '"' || NAME || '"'; --

   EXECUTE IMMEDIATE FSTMT; --
 END IF; --
```

```
END;

CREATE TRIGGER SYSTOOLS.OPT_PROFILE_UTRIG AFTER UPDATE ON SYSTOOLS.OPT_PROFILE
REFERENCING OLD AS O
FOR EACH ROW
  CALL SYSTOOLS.OPT_FLUSH_CACHE( O.SCHEMA, O.NAME );

CREATE TRIGGER SYSTOOLS.OPT_PROFILE_DTRIG AFTER DELETE ON SYSTOOLS.OPT_PROFILE
REFERENCING OLD AS O
FOR EACH ROW
  CALL SYSTOOLS.OPT_FLUSH_CACHE( O.SCHEMA, O.NAME );
```

Managing the SYSTOOLS.OPT_PROFILE table:

Optimization profiles must be associated with a unique schema-qualified name and stored in the SYSTOOLS.OPT_PROFILE table. You can use the **LOAD**, **IMPORT**, and **EXPORT** commands to manage the files in that table.

For example, the **IMPORT** command can be used from any Db2 client to insert or update data in the SYSTOOLS.OPT_PROFILE table. The **EXPORT** command can be used to copy a profile from the SYSTOOLS.OPT_PROFILE table into a file.

The following example shows how to insert three new profiles into the SYSTOOLS.OPT_PROFILE table. Assume that the files are in the current directory.

1. Create an input file (for example, `profiledata`) with the schema, name, and file name for each profile on a separate line:

   ```
   "ROBERT","PROF1","ROBERT.PROF1.xml"
   "ROBERT","PROF2","ROBERT.PROF2.xml"
   "DAVID", "PROF1","DAVID.PROF1.xml"
   ```

2. Execute the **IMPORT** command:

   ```
   import from profiledata of del
     modified by lobsinfile
     insert into systools.opt_profile
   ```

To update existing rows, use the INSERT_UPDATE option on the **IMPORT** command:

```
import from profiledata of del
  modified by lobsinfile
  insert_update into systools.opt_profile
```

To copy the ROBERT.PROF1 profile into `ROBERT.PROF1.xml`, assuming that the profile is less than 32 700 bytes long, use the **EXPORT** command:

```
export to robert.prof1.xml of del
  select profile from systools.opt_profile
    where schema='ROBERT' and name='PROF1'
```

For more information, including how to export more than 32 700 bytes of data, see "EXPORT command".

Database partition group impact on query optimization:

In partitioned database environments, the optimizer recognizes and uses the collocation of tables when it determines the best access plan for a query.

If tables are frequently involved in join queries, they should be divided among database partitions in such a way that the rows from each table being joined are located on the same database partition. During the join operation, the collocation

of data from both joined tables prevents the movement of data from one database partition to another. Place both tables in the same database partition group to ensure that the data is collocated.

Depending on the size of the table, spreading data over more database partitions reduces the estimated time to execute a query. The number of tables, the size of the tables, the location of the data in those tables, and the type of query (such as whether a join is required) all affect the cost of the query.

Collecting accurate catalog statistics, including advanced statistics features:

Accurate database statistics are critical for query optimization. Perform **RUNSTATS** command operations regularly on any tables that are critical to query performance.

You might also want to collect statistics on system catalog tables, if an application queries these tables directly and if there is significant catalog update activity, such as that resulting from the execution of data definition language (DDL) statements. Automatic statistics collection can be enabled to allow the Db2 data server to automatically perform a **RUNSTATS** command operation. Real-time statistics collection can be enabled to allow the Db2 data server to provide even more timely statistics by collecting them immediately before queries are optimized.

If you are collecting statistics manually by using the **RUNSTATS** command, use the following options at a minimum:

```
RUNSTATS ON TABLE DB2USER.DAILY_SALES
  WITH DISTRIBUTION AND SAMPLED DETAILED INDEXES ALL
```

Distribution statistics make the optimizer aware of data skew. Detailed index statistics provide more details about the I/O required to fetch data pages when the table is accessed by using a particular index. Collecting detailed index statistics uses considerable processing time and memory for large tables. The **SAMPLED** option provides detailed index statistics with nearly the same accuracy but requires a fraction of the CPU and memory. These options are used by automatic statistics collection when a statistical profile is not provided for a table.

To improve query performance, consider collecting more advanced statistics, such as column group statistics or LIKE statistics, or creating statistical views.

Statistical views are helpful when gathering statistics for complex relationships. Gathering statistics for statistical views can be automated through the automatic statistics collection feature in Db2. Enabling or disabling the automatic statistic collection of statistical views is done by using the **auto_stats_views** database configuration parameter. To enable this function, issue the following command:

```
update db cfg for dbname using auto_stats_views on
```

To disable this feature, issue the following command:

```
update db cfg for dbname using auto_stats_views off
```

This database configuration parameter is off by default. The command that is issued to automatically collect statistics on statistical views is equivalent to the following command:

```
runstats on view view_name with distribution
```

Collecting statistics for a large table or statistical view can be time consuming. Statistics of the same quality can often be collected by considering just a small sample of the overall data. Consider enabling automatic sampling for all

background statistic collections; this may reduce the statistic collection time. To enable this function, issue the following command:

```
update db cfg for dbname using auto_sampling on
```

Collected statistics are not always exact. In addition to providing more efficient data access, an index can help provide more accurate statistics for columns which are often used in query predicates. When statistics are collected for a table and its indexes, index objects can provide accurate statistics for the leading index columns.

Column group statistics:

If your query has more than one join predicate joining two tables, the Db2 optimizer calculates how selective each of the predicates is before choosing a plan for executing the query.

For example, consider a manufacturer who makes products from raw material of various colors, elasticities, and qualities. The finished product has the same color and elasticity as the raw material from which it is made. The manufacturer issues the query:

```
SELECT PRODUCT.NAME, RAWMATERIAL.QUALITY
  FROM PRODUCT, RAWMATERIAL
  WHERE
    PRODUCT.COLOR = RAWMATERIAL.COLOR AND
    PRODUCT.ELASTICITY = RAWMATERIAL.ELASTICITY
```

This query returns the names and raw material quality of all products. There are two join predicates:

```
PRODUCT.COLOR = RAWMATERIAL.COLOR
PRODUCT.ELASTICITY = RAWMATERIAL.ELASTICITY
```

The optimizer assumes that the two predicates are independent, which means that all variations of elasticity occur for each color. It then estimates the overall selectivity of the pair of predicates by using catalog statistics information for each table based on the number of levels of elasticity and the number of different colors. Based on this estimate, it might choose, for example, a nested loop join in preference to a merge join, or the reverse.

However, these two predicates might not be independent. For example, highly elastic materials might be available in only a few colors, and the very inelastic materials might be available in a few other colors that are different from the elastic ones. In that case, the combined selectivity of the predicates eliminates fewer rows and the query returns more rows. Without this information, the optimizer might no longer choose the best plan.

To collect the column group statistics on PRODUCT.COLOR and PRODUCT.ELASTICITY, issue the following **RUNSTATS** command:

```
RUNSTATS ON TABLE PRODUCT ON COLUMNS ((COLOR, ELASTICITY))
```

The optimizer uses these statistics to detect cases of correlation and to dynamically adjust the combined selectivities of correlated predicates, thus obtaining a more accurate estimate of the join size and cost.

When a query groups data by using keywords such as GROUP BY or DISTINCT, column group statistics also enable the optimizer to compute the number of distinct groupings.

Consider the following query:

```
SELECT DEPTNO, YEARS, AVG(SALARY)
  FROM EMPLOYEE
  GROUP BY DEPTNO, MGR, YEAR_HIRED
```

Without any index or column group statistics, the optimizer estimates the number of groupings (and, in this case, the number of rows returned) as the product of the number of distinct values in DEPTNO, MGR, and YEAR_HIRED. This estimate assumes that the grouping key columns are independent. However, this assumption could be incorrect if each manager manages exactly one department. Moreover, it is unlikely that each department hires employees every year. Thus, the product of distinct values of DEPTNO, MGR, and YEAR_HIRED could be an overestimate of the actual number of distinct groups.

Column group statistics collected on DEPTNO, MGR, and YEAR_HIRED provide the optimizer with the exact number of distinct groupings for the previous query:

```
RUNSTATS ON TABLE EMPLOYEE ON COLUMNS ((DEPTNO, MGR, YEAR_HIRED))
```

In addition to JOIN predicate correlation, the optimizer manages correlation with simple equality predicates, such as:

```
DEPTNO = 'Sales' AND  MGR = 'John'
```

In this example, predicates on the DEPTNO column in the EMPLOYEE table are likely to be independent of predicates on the YEAR column. However, the predicates on DEPTNO and MGR are not independent, because each department would usually be managed by one manager at a time. The optimizer uses statistical information about columns to determine the combined number of distinct values and then adjusts the cardinality estimate to account for correlation between columns.

Column group statistics can also be used on statistical views. The column group statistics help adjust the skewed statistics in the statistical view when there is more than one strong correlation in the queries. The optimizer can use these statistics to obtain better cardinality estimates which might result in better access plans.

Correlation of simple equality predicates:

In addition to join predicate correlation, the optimizer manages correlation with simple equality predicates of the type `COL =`.

For example, consider a table of different types of cars, each having a MAKE (that is, a manufacturer), MODEL, YEAR, COLOR, and STYLE, such as sedan, station wagon, or sports-utility vehicle. Because almost every manufacturer makes the same standard colors available for each of their models and styles, year after year, predicates on COLOR are likely to be independent of those on MAKE, MODEL, STYLE, or YEAR. However, predicates based on MAKE and MODEL are not independent, because only a single car maker would make a model with a particular name. Identical model names used by two or more car makers is very unlikely.

If an index on the two columns MAKE and MODEL exists, or column group statistics are collected, the optimizer uses statistical information about the index or columns to determine the combined number of distinct values and to adjust the selectivity or cardinality estimates for the correlation between these two columns. If the predicates are local equality predicates, the optimizer does not need a unique index to make an adjustment.

Statistical views

The Db2 cost-based optimizer uses an estimate of the number of rows processed by an access plan operator to accurately cost that operator. This cardinality estimate is the single most important input to the optimizer cost model, and its accuracy largely depends upon the statistics that the **RUNSTATS** command collects from the database.

More sophisticated statistics are required to represent more complex relationships, such as the following relationships:

- Comparisons involving expressions. For example, `price > MSRP + Dealer_markup`.
- Relationships spanning multiple tables. For example, `product.name = 'Alloy wheels' and product.key = sales.product_key`.
- Any relationships other than predicates involving independent attributes and simple comparison operations.

Statistical views are able to represent these types of complex relationships, because statistics are collected on the result set returned by the view, rather than the base tables referenced by the view.

When a query is compiled, the optimizer matches the query to the available statistical views. When the optimizer computes cardinality estimates for intermediate result sets, it uses the statistics from the view to compute a better estimate.

Queries do not need to reference the statistical view directly in order for the optimizer to use the statistical view. The optimizer uses the same matching mechanism that is used for materialized query tables (MQTs) to match queries to statistical views. In this respect, statistical views are similar to MQTs, except that they are not stored permanently, do not consume disk space, and do not have to be maintained.

A statistical view is created by first creating a view and then enabling it for optimization by using the ALTER VIEW statement. The **RUNSTATS** command is then run against the statistical view, populating the system catalog tables with statistics for the view. For example, to create a statistical view that represents the join between the TIME dimension table and the fact table in a star schema, run the following statements and commands:

```
CREATE VIEW SV_TIME_FACT AS (
  SELECT T.* FROM TIME T, SALES S
    WHERE T.TIME_KEY = S.TIME_KEY)

ALTER VIEW SV_TIME_FACT ENABLE QUERY OPTIMIZATION

RUNSTATS ON TABLE DB2DBA.SV_TIME_FACT WITH DISTRIBUTION
```

Once the view is enabled for optimization, it is identified as a statistical view in the SYSCAT.TABLES catalog view with a 'Y' in position 13 of the PROPERTY column.

This statistical view can be used to improve the cardinality estimate and, consequently, the access plan and query performance for queries such as the following query:

```
SELECT SUM(S.PRICE)
  FROM SALES S, TIME T, PRODUCT P
  WHERE
```

```
T.TIME_KEY = S.TIME_KEY AND
T.YEAR_MON = 200712 AND
P.PROD_KEY = S.PROD_KEY AND
P.PROD_DESC = 'Power drill'
```

Without a statistical view, the optimizer assumes that all fact table TIME_KEY values corresponding to a particular TIME dimension YEAR_MON value occur uniformly within the fact table. However, sales might have been strong in December, resulting in many more sales transactions than during other months.

Statistics that are gathered on queries that have complex expressions in the predicate can be used by the optimizer to calculate accurate cardinality estimates which results in better access plans.

For many star join queries many statistical views might need to be created, however, if you have referential integrity constraints you can narrow down these many statistical views. The statistics for the other views can be inferred from the reduced number of statistical views by using referential integrity constraints.

Another way to obtain better access plans is to apply column group statistics on statistical views. These group statistics help to adjust filter factors which help to gather more accurate statistics which the optimizer can use to obtain accurate cardinality estimates.

Statistics can also be gathered automatically from statistical views through the automatic statistics collection feature in Db2. This new feature can be enabled or disabled by using the **auto_stats_views** database configuration parameter. This database configuration parameter is off by default and can be enabled by using the **UPDATE DB CFG** command. The statistics collected by the automatic statistics collection is equivalent to issuing the command `runstats on view` *view_name* `with distribution`.

Utility throttling can be used for statistical views to restrict the performance impact on the workload. For example, the command `runstats on view` *view_name* `util_impact_priority 10` contains the impact on the workload within a limit specified by the **util_impact_lim** database manager configuration parameter while statistics are collected on statistical views.

A statistical view cannot directly or indirectly reference a catalog table.

Using statistical views:

A view must be enabled for optimization before its statistics can be used to optimize a query. A view that is enabled for optimization is known as a *statistical view*.

About this task

A view that is not a statistical view is said to be disabled for optimization and is known as a *regular view*. A view is disabled for optimization when it is first created. Use the ALTER VIEW statement to enable a view for optimization. For privileges and authorities that are required to perform this task, see the description of the ALTER VIEW statement. For privileges and authorities that are required to use the **RUNSTATS** utility against a view, see the description of the **RUNSTATS** command.

A view cannot be enabled for optimization if any one of the following conditions is true:

- The view directly or indirectly references a materialized query table (MQT). (An MQT or statistical view can reference a statistical view.)
- The view directly or indirectly references a catalog table.
- The view is inoperative.
- The view is a typed view.
- There is another view alteration request in the same ALTER VIEW statement.

If the definition of a view that is being altered to enable optimization contains any of the following items, a warning is returned, and the optimizer will not exploit the view's statistics:

- Aggregation or distinct operations
- Union, except, or intersect operations
- OLAP specification

Procedure

1. Enable the view for optimization.

   A view can be enabled for optimization using the ENABLE OPTIMIZATION clause on the ALTER VIEW statement. A view that has been enabled for optimization can subsequently be disabled for optimization using the DISABLE OPTIMIZATION clause. For example, to enable MYVIEW for optimization, enter the following:

   ```
   alter view myview enable query optimization
   ```

2. Invoke the **RUNSTATS** command. For example, to collect statistics on MYVIEW, enter the following:

   ```
   runstats on table db2dba.myview
   ```

   To use row-level sampling of 10 percent of the rows while collecting view statistics, including distribution statistics, enter the following:

   ```
   runstats on table db2dba.myview with distribution tablesample bernoulli (10)
   ```

   To use page-level sampling of 10 percent of the pages while collecting view statistics, including distribution statistics, enter the following:

   ```
   runstats on table db2dba.myview with distribution tablesample system (10)
   ```

3. Optional: If queries that are impacted by the view definition are part of static SQL packages, rebind those packages to take advantage of changes to access plans resulting from the new statistics.

View statistics that are relevant to optimization:

Only statistics that characterize the data distribution of the query that defines a statistical view, such as CARD and COLCARD, are considered during query optimization.

The following statistics that are associated with view records can be collected for use by the optimizer.

- Table statistics (SYSCAT.TABLES, SYSSTAT.TABLES)
  - CARD - Number of rows in the view result
- Column statistics (SYSCAT.COLUMNS, SYSSTAT.COLUMNS)
  - COLCARD - Number of distinct values of a column in the view result

  - AVGCOLLEN - Average length of a column in the view result
  - HIGH2KEY - Second highest value of a column in the view result
  - LOW2KEY - Second lowest value of a column in the view result
  - NUMNULLS - Number of null values in a column in the view result
  - SUB_COUNT - Average number of sub-elements in a column in the view result
  - SUB_DELIM_LENGTH - Average length of each delimiter separating sub-elements
- Column distribution statistics (SYSCAT.COLDIST, SYSSTAT.COLDIST)
  - DISTCOUNT - Number of distinct quantile values that are less than or equal to COLVALUE statistics
  - SEQNO - Frequency ranking of a sequence number to help uniquely identify a row in the table
  - COLVALUE - Data value for which frequency or quantile statistics are collected
  - VALCOUNT - Frequency with which a data value occurs in a view column; or for quantiles, the number of values that are less than or equal to the data value (COLVALUE)

Statistics that do not describe data distribution (such as NPAGES and FPAGES) can be collected, but are ignored by the optimizer.

Scenario: Improving cardinality estimates using statistical views:

In a data warehouse, fact table information often changes quite dynamically, whereas dimension table data is static. This means that dimension attribute data might be positively or negatively correlated with fact table attribute data.

Traditional base table statistics currently available to the optimizer do not allow it to discern relationships across tables. Column and table distribution statistics on statistical views (and MQTs) can be used to give the optimizer the necessary information to correct these types of cardinality estimation errors.

Consider the following query that computes annual sales revenue for golf clubs sold during July of each year:

```
select  sum(f.sales_price), d2.year
  from  product d1, period d2, daily_sales f
  where  d1.prodkey = f.prodkey
    and  d2.perkey = f.perkey
    and  d1.item_desc = 'golf club'
    and  d2.month = 'JUL'
  group by  d2.year
```

A star join query execution plan can be an excellent choice for this query, provided that the optimizer can determine whether the semi-join involving PRODUCT and DAILY_SALES, or the semi-join involving PERIOD and DAILY_SALES, is the most selective. To generate an efficient star join plan, the optimizer must be able to choose the most selective semi-join for the outer leg of the index ANDing operation.

Data warehouses often contain records for products that are no longer on store shelves. This can cause the distribution of PRODUCT columns after the join to appear dramatically different than their distribution before the join. Because the optimizer, for lack of better information, will determine the selectivity of local

predicates based solely on base table statistics, the optimizer might become overly optimistic regarding the selectivity of the predicate `item_desc = 'golf club'`

For example, if golf clubs historically represent 1% of the products manufactured, but now account for 20% of sales, the optimizer would likely overestimate the selectivity of `item_desc = 'golf club'`, because there are no statistics describing the distribution of *item_desc* after the join. And if sales in all twelve months are equally likely, the selectivity of the predicate `month = 'JUL'` would be around 8%, and thus the error in estimating the selectivity of the predicate `item_desc = 'golf club'` would mistakenly cause the optimizer to perform the seemingly more selective semi-join between PRODUCT and DAILY_SALES as the outer leg of the star join plan's index ANDing operation.

The following example provides a step-by-step illustration of how to set up statistical views to solve this type of problem.

Consider a database from a typical data warehouse, where STORE, CUSTOMER, PRODUCT, PROMOTION, and PERIOD are the dimension tables, and DAILY_SALES is the fact table. The following tables provide the definitions for these tables.

Table 62. STORE (63 rows)

Column	storekey	store_number	city	state	district	...
Attribute	integer not null primary key	char(2)	char(20)	char(5)	char(14)	...

Table 63. CUSTOMER (1 000 000 rows)

Column	custkey	name	address	age	gender	...
Attribute	integer not null primary key	char(30)	char(40)	smallint	char(1)	...

Table 64. PRODUCT (19 450 rows)

Column	prodkey	category	item_desc	price	cost	...
Attribute	integer not null primary key	integer	char(30)	decimal(11)	decimal(11)	...

Table 65. PROMOTION (35 rows)

Column	promokey	promotype	promodesc	promovalue	...
Attribute	integer not null primary key	integer	char(30)	decimal(5)	...

Table 66. PERIOD (2922 rows)

Column	perkey	calendar_date	month	period	year	...
Attribute	integer not null primary key	date	char(3)	smallint	smallint	...

Table 67. DAILY_SALES (754 069 426 rows)

Column	storekey	custkey	prodkey	promokey	perkey	sales_price	...
Attribute	integer	integer	integer	integer	integer	decimal(11)	...

Suppose the company managers want to determine whether or not consumers will buy a product again if they are offered a discount on a return visit. Moreover, suppose this study is done only for store '01', which has 18 locations nationwide. Table 68 shows information about the different categories of promotion that are available.

Table 68. PROMOTION (35 rows)

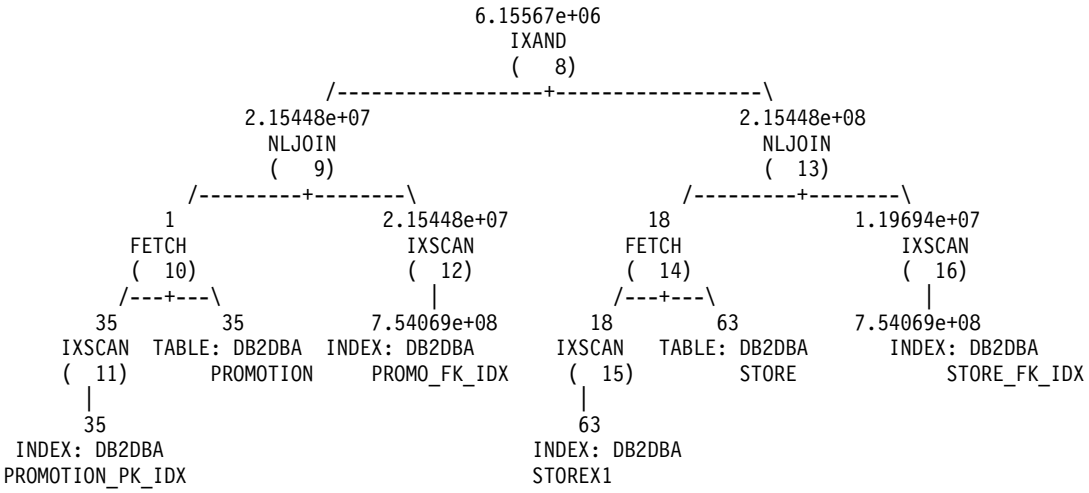
promotype	promodesc	COUNT (promotype)	percentage of total
1	Return customers	1	2.86%
2	Coupon	15	42.86%
3	Advertisement	5	14.29%
4	Manager's special	3	8.57%
5	Overstocked items	4	11.43%
6	End aisle display	7	20.00%

The table indicates that discounts for return customers represents only 2.86% of the 35 kinds of promotions that were offered.

The following query returns a count of 12 889 514:

```
select  count(*)
  from  store d1, promotion d2, daily_sales f
  where  d1.storekey = f.storekey
    and  d2.promokey = f.promokey
    and  d1.store_number = '01'
    and  d2.promotype = 1
```

This query executes according to the following plan that is generated by the optimizer. In each node of this diagram, the first row is the cardinality estimate, the second row is the operator type, and the third row (the number in parentheses) is the operator ID.



At the nested loop join (number 9), the optimizer estimates that around 2.86% of the product sold resulted from customers coming back to buy the same products at a discounted price (2.15448e+07 ÷ 7.54069e+08 ≈ 0.0286). Note that this is the same value before and after joining the PROMOTION table with the DAILY_SALES

table. Table 69 summarizes the cardinality estimates and their percentage (the filtering effect) before and after the join.

Table 69. Cardinality estimates before and after joining with DAILY_SALES.

	Before Join		After Join	
Predicate	**count**	**percentage of rows qualified**	**count**	**percentage of rows qualified**
store_number = '01'	18	28.57%	2.15448e+08	28.57%
promotype = 1	1	2.86%	2.15448e+07	2.86%

Because the probability of `promotype = 1` is less than that of `store_number = '01'`, the optimizer chooses the semi-join between PROMOTION and DAILY_SALES as the outer leg of the star join plan's index ANDing operation. This leads to an estimated count of approximately 6 155 670 products sold using promotion type 1 - an incorrect cardinality estimate that is off by a factor of 2.09 (12 889 514 ÷ 6 155 670 ≈ 2.09).

What causes the optimizer to only estimate half of the actual number of records satisfying the two predicates? Store '01' represents about 28.57% of all the stores. What if other stores had more sales than store '01' (less than 28.57%)? Or what if store '01' actually sold most of the product (more than 28.57%)? Likewise, the 2.86% of products sold using promotion type 1 shown in Table 69 can be misleading. The actual percentage in DAILY_SALES could very well be a different figure than the projected one.

You can use statistical views to help the optimizer correct its estimates. First, you need to create two statistical views representing each semi-join in the previous query. The first statistical view provides the distribution of stores for all daily sales. The second statistical view represents the distribution of promotion types for all daily sales. Note that each statistical view can provide the distribution information for any particular store number or promotion type. In this example, you use a 10% sample rate to retrieve the records in DAILY_SALES for the respective views and save them in global temporary tables. You can then query those tables to collect the necessary statistics to update the two statistical views.

1. Create a view representing the join of STORE with DAILY_SALES.

```
create view sv_store_dailysales as
  (select s.*
    from store s, daily_sales ds
    where s.storekey = ds.storekey)
```

2. Create a view representing the join of PROMOTION with DAILY_SALES.

```
create view sv_promotion_dailysales as
  (select p.*
    from promotion.p, daily_sales ds
    where p.promokey = ds.promokey)
```

3. Make the views statistical views by enabling them for query optimization:

```
alter view sv_store_dailysales enable query optimization
alter view sv_promotion_dailysales enable query optimization
```

4. Execute the **RUNSTATS** command to collect statistics on the views:

```
runstats on table db2dba.sv_store_dailysales with distribution
runstats on table db2dba.sv_promotion_dailysales with distribution
```

5. Run the query again so that it can be re-optimized. Upon reoptimization, the optimizer will match SV_STORE_DAILYSALES and SV_PROMOTION_DAILYSALES with the query, and will use the view statistics

to adjust the cardinality estimate of the semi-joins between the fact and dimension tables, causing a reversal of the original order of the semi-joins chosen without these statistics. The new plan is as follows:

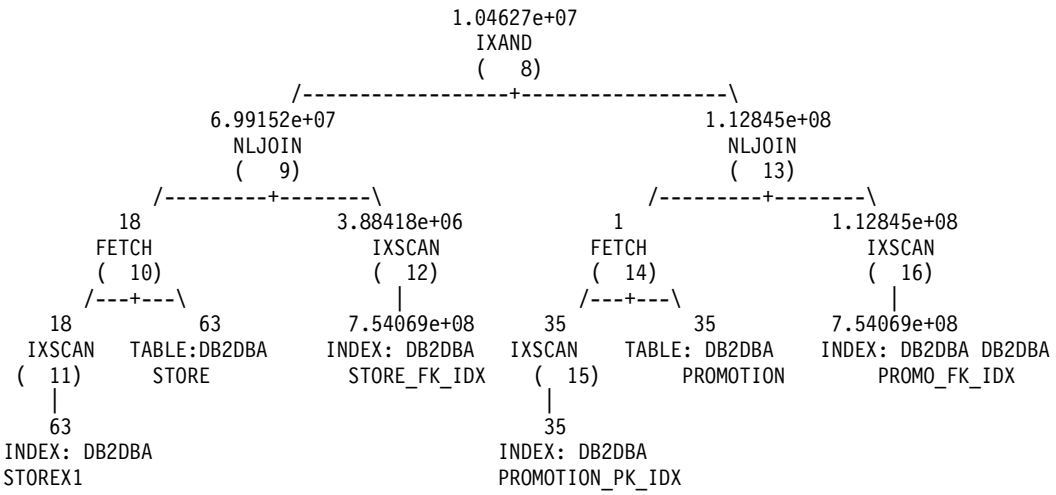


Table 70 summarizes the cardinality estimates and their percentage (the filtering effect) before and after the join for each semi-join.

Table 70. Cardinality estimates before and after joining with DAILY_SALES.

	Before Join		**After Join (***no statistical views***)**		**After Join (***with statistical views***)**	
Predicate	**count**	**percentage of rows qualified**	**count**	**percentage of rows qualified**	**count**	**percentage of rows qualified**
store_number = '01'	18	28.57%	2.15448e+08	28.57%	6.99152e+07	9.27%
promotype = 1	1	2.86%	2.15448e+07	2.86%	1.12845e+08	14.96%

Note that this time, the semi-join between STORE and DAILY_SALES is performed on the outer leg of the index ANDing plan. This is because the two statistical views essentially tell the optimizer that the predicate `store_number = '01'` will filter more rows than `promotype = 1`. This time, the optimizer estimates that there are approximately 10 462 700 products sold. This estimate is off by a factor of 1.23 (12 889 514 ÷ 10 462 700 ≈ 1.23), which is a significant improvement over the estimate without statistical views (in Table 69 on page 440).

Statistics used from expression columns in statistical views:

The query optimizer can use statistics from the expression columns in statistical views to obtain accurate cardinality estimates which results in better access plans.

For example, consider a query that uses the UCASE scalar function:

```
SELECT * FROM t1, t2 WHERE t2.c2 = UCASE(t1.c1)
```

The query optimizer can use the statistics from a statistical view for these types of queries to obtain better access plans.

The earlier example query would benefit from a statistical view that includes the UCASE scalar function on column c1. The following example creates a view that includes the UCASE scalar function on column c1.

```
CREATE VIEW sv AS (SELECT UCASE(c1) AS c1 FROM t1)
ALTER VIEW sv ENABLE QUERY OPTIMIZATION
RUNSTATS ON TABLE dba.sv WITH DISTRIBUTION
```

To obtain statistics for these types of queries, one side of the predicate must be an expression that matches an expression in the statistical view column definition exactly.

Here are some examples where the query optimizer does not use the statistics from a statistical view:

- One side of the predicate in the query is an expression that is matched with more than one expression column in a statistical view:

  ```
  create view SV14(c1, c2) as (select c1+c2, c1*c2 from t1 where c1 > 3);
  alter view SV14 enable query optimization;
  runstats on table schema.sv1 with distribution;
  select * from t1 where (c1+c2) + (c1*c2) > 5 and c1 > 3;
  ```

  Here the expression `(c1+c2) + (c1*c2)` matched to two columns in view SV14. The statistics of view SV14 for this expression are not used.
- One side of the predicate in the query is an expression that is partially matched with an expression column in a statistical view:

  ```
  create view SV15(c1, c2) as (select c1+c2, c1*c2 from t1 where c1 > 3);
  alter view SV15 enable query optimization;
  runstats on table schema.SV15 with distribution;
  select * from t1 where (c1+c2) + 10 > 5 and c1 > 3;
  ```

  Here the expression `(c1+c2) + 10` is partially matched to `c1+c2` in view SV15. The statistics of view SV15 for this expression are not used.
- One side of the predicate in the query is indirectly matched to an expression column in a statistical view:

  ```
  create view SV16(c1, c2) as (select c1+c2, c1*c2 from t1 where c1 > 3);
  alter view SV16 enable query optimization;
  runstats on table schema.SV16 with distribution;
  select * from t3 left join table (select ta.c1 from t2 left join table
    (select c1+c2,c3 from t1 where c1 > 3) as ta(c1,c3) on t2.c1 = ta.c3) as
     tb(c1) on t3.c1= TB.C1;
  ```

  Here the column TB.C1 indirectly matches the expression `c1+c2` in view SV16. The statistics of view SV16 for this expression are not used.

Referential integrity constraints help reduce the number of statistical views:

Statistical views can be used to obtain good cardinality estimates for the Db2 query optimizer, but they also require resources to maintain and process. However, by using referential integrity constraints, you can cut down on the number of statistical views used to gather statistics.

If you want to collect accurate statistics for a star join query that joins different sets of tables, you can create multiple statistical views or one statistical view by using referential integrity constraints.

For example, consider a schema which has tables D1, D2, D3, D4, D5, and F:

```
create table D1 (D1_PK int not null primary key,
                                D1_C1 int,
                                D1_C2 int,
                                D1_C3 int,
                                D1_FK int not null);
create table D2 (D2_PK int not null primary key,
```

```
                                  D2_C1 int,
                                  D2_C2 int,
                                  D2_C3 int,
                                  D2_FK int not null);
create table D3 (D3_PK int not null primary key,
                                  D3_C1 int,
                                  D3_C2 int,
                                  D3_C3 int,
                                  D3_FK int not null);
create table D4 (D4_PK int not null primary key,
                                  D4_C1 int,
                                  D4_C2 int,
                                  D4_C3 int,
                                  D4_FK int not null);
create table D5 (D5_PK int not null primary key,
                                  D5_C1 int,
                                  D5_C2 int,
                                  D5_C3 int,
                                  D5_FK int not null);
create table F  (F_FK1 int not null,
                                 F_FK2 int not null,
                                 F_FK3 int not null,
                                 F_FK4 int not null,
                                 F_FK5 int not null,
                                 F_C1 int, F_C2 int);
```

Consider that table F is altered in the following way:

```
alter table F add foreign key (F_FK1)
 references D1 (D1_PK) on delete cascade;
alter table F add foreign key (F_FK2)
 references D2 (D2_PK) on delete cascade;
alter table F add foreign key (F_FK3)
 references D3 (D3_PK) on delete cascade;
alter table F add foreign key (F_FK4)
 references D4 (D4_PK) on delete cascade;
alter table F add foreign key (F_FK5)
 references D5 (D5_PK) on delete cascade;
```

Also, consider that you want to provide statistics for the following query:

```
select distinct * from F, D1, D2, D3 where F_FK1 = D1_PK and F_FK2
   = D2_PK and F_FK3 = D3_PK and D1_C1='ON' and D2_C2>='2009-01-01';
```

To gather accurate statistics you can create the complete set of views, as follows:

```
create view SV1 as (select D1.* from F, D1 where   F_FK1 = D1_PK);
alter view SV1 enable query optimization;

create view SV2 as(select D2.* from F, D2 where  F_FK2 = D2_PK);
alter view SV2 enable query optimization;

create view SV3 as(select D3.* from F, D3 where  F_FK3 = D3_PK);
alter view SV3 enable query optimization;

create view SV4 as(select D1.*, D2.*, D3.* from F, D1, D2, D3 where
  F_FK1 = D1_PK and F_FK2 = D2_PK and F_FK3 = D3_PK);
alter view SV4 enable query optimization;
```

You can reduce the number of statistical views created to obtain accurate statistics if referential integrity constraints exist between join columns. This reduction in the number of statistical views needed, saves you time in creating, updating, and maintaining statistical views. For this example, the following single statistical view would be sufficient to obtain the same statistics as the complete set of statistical views created earlier:

```
create view SV5 as (select D1.*, D2.*, D3.*, D4.*, D5.* from F, D1, D2, D3, D4, D5
   where
      F_FK1 = D1_PK and F_FK2 = D2_PK and F_FK3 = D3_PK
      and F_FK4 = D4_PK and F_FK5 = D5_PK
);
alter view SV5 enable query optimization;
```

The statistics for SV4, SV3, SV2, and SV1 are inferred from SV5 based on referential integrity constraints. The referential integrity constraints between F, D1, D2, D3, D4, and D5 ensure that the joins among them are lossless. These lossless joins let us infer that the following cardinalities are the same:

- The cardinality of the join result between F and D1
- The cardinality of SV1
- The cardinality of the statistical view SV5

The same reasoning applies to the joins between F and D2, F and D3, and F and D4. This reasoning shows that just one statistical view SV5 is required by the Db2 optimizer to obtain improved cardinality estimates for the earlier query.

For referential integrity constraints that permit NULL values, the percentage of NULL values that are in foreign key values can affect statistics inferred from a statistical view.

If the NULL values are a significant percentage of the foreign key values, the statistics and cardinality estimates inferred by the Db2 optimizer by using the constraints can be inaccurate. Prevent these inaccuracies by preventing the referential integrity constraints from being used to infer statistics. To prevent the constraints from being used to infer statistics, disable query optimization by using the DISABLE QUERY OPTIMIZATION option of the ALTER TABLE command.

You can detect this situation by using the explain facility. The explain facility shows the proportion of NULL values in the foreign key by producing an explain diagnostic message if the number of NULL values is possibly too high and if statistics are being inferred from a statistical view.

Statistics used from column group statistics on statistical views:

In complex scenarios where statistics need to be gathered for a query, creating a statistical view or gathering column group statistics on a table is not enough. You might need to combine the two to collect column group statistics on a statistical view. This combination of statistical view and column group statistics can help the Db2 optimizer to generate better access plans.

The use of column groups statistics on statistical views usually occurs when additional correlation exists on top of a typical situation that statistical views would normally handle.

For example, consider the following query and statistical view:

```
select * from T1,T2 where T1.C1=T2.D1 and T1.C2=5 and T2.D3=10;
create view SV2 as (select * from T1,T2 where T1.C1=T2.D1);
alter view SV2 enable query optimization;
runstats on table db2.SV2 on all columns with distribution;
```

This query might run slowly and the cardinality estimate can be inaccurate. If you check the access plan, you might find that there is a strong correlation between T1.C2 and T2.D3 although the cardinality estimate has been adjusted by the statistical view. Therefore, the cardinality estimate is inaccurate.

To resolve this situation, you can collect column group statistics of the view SV2 by issuing the following command:

```
runstats on table db2.SV2 on all columns and columns((C2,D3)) with distribution;
```

These additional statistics help improve the cardinality estimates which might result in better access plans.

Note: Collecting distribution statistics for a list of columns in a statistical view is not supported.

Collecting column group statistics on statistical views can also be used to compute the number of distinct groupings, or the grouping key cardinality, for queries that require data to be grouped in a certain way. A grouping requirement can result from operations such as the GROUP BY or DISTINCT operations.

For example, consider the following query and statistical view:

```
select T1.C1, T1.C2 from T1,T2 where T1.C3=T2.C3 group by T1.C1, T1.C2;
create view SV2 as (select T1.C1, T1.C2 from T1,T2 where T1.C3=T2.C3);
alter view SV2 enable query optimization;
```

Collecting column group statistics on the statistical view covering the join predicate helps the optimizer estimate the grouping key cardinality more accurately. Issue the following command to collect the column group statistics:

```
runstats on table db2.SV2 on columns((C1,C2));
```

Catalog statistics

When the query compiler optimizes query plans, its decisions are heavily influenced by statistical information about the size of the database tables, indexes, and statistical views. This information is stored in system catalog tables.

The optimizer also uses information about the distribution of data in specific columns of tables, indexes, and statistical views if these columns are used to select rows or to join tables. The optimizer uses this information to estimate the costs of alternative access plans for each query.

Statistical information about the cluster ratio of indexes, the number of leaf pages in indexes, the number of table rows that overflow their original pages, and the number of filled and empty pages in a table can also be collected. You can use this information to decide when to reorganize tables or indexes.

Table statistics in a partitioned database environment are collected only for that portion of the table that resides on the database partition on which the utility is running, or for the first database partition in the database partition group that contains the table. Information about statistical views is collected for all database partitions.

Statistics that are updated by the runstats utility

Catalog statistics are collected by the runstats utility, which can be started by issuing the **RUNSTATS** command, calling the ADMIN_CMD procedure, or calling the db2Runstats API. Updates can be initiated either manually or automatically.

In IBM Data Studio Version 3.1 or later, you can use the task assistant for collecting catalog statistics. Task assistants can guide you through the process of setting

options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

Statistics about declared temporary tables are not stored in the system catalog, but are stored in memory structures that represent the catalog information for declared temporary tables. It is possible (and in some cases, it might be useful) to perform runstats on a declared temporary table.

The runstats utility collects the following information about tables and indexes:

- The number of pages that contain rows
- The number of pages that are in use
- The number of rows in the table (the *cardinality*)
- The number of rows that overflow
- For multidimensional clustering (MDC) and insert time clustering (ITC) tables, the number of blocks that contain data
- For partitioned tables, the degree of data clustering within a single data partition
- Data distribution statistics, which are used by the optimizer to estimate efficient access plans for tables and statistical views whose data is not evenly distributed and whose columns have a significant number of duplicate values
- Detailed index statistics, which are used by the optimizer to determine how efficient it is to access table data through an index
- Subelement statistics for LIKE predicates, especially those that search for patterns within strings (for example, `LIKE %disk%`), are also used by the optimizer

The runstats utility collects the following statistics for each data partition in a table. These statistics are only used for determining whether a partition needs to be reorganized:

- The number of pages that contain rows
- The number of pages that are in use
- The number of rows in the table (the cardinality)
- The number of rows that overflow
- For MDC and ITC tables, the number of blocks that contain data

Distribution statistics are not collected:

- When the **num_freqvalues** and **num_quantiles** database configuration parameters are set to 0
- When the distribution of data is known, such as when each data value is unique
- When the column contains a LONG, LOB, or structured data type
- For row types in sub-tables (the table-level statistics NPAGES, FPAGES, and OVERFLOW are not collected)
- If quantile distributions are requested, but there is only one non-null value in the column
- For extended indexes or declared temporary tables

The runstats utility collects the following information about each column in a table or statistical view, and the first column in an index key:

- The cardinality of the column

- The average length of the column (the average space, in bytes, that is required when the column is stored in database memory or in a temporary table)
- The second highest value in the column
- The second lowest value in the column
- The number of null values in the column

For columns that contain large object (LOB) or LONG data types, the runstats utility collects only the average length of the column and the number of null values in the column. The average length of the column represents the length of the data descriptor, except when LOB data is located inline on the data page. The average amount of space that is required to store the column on disk might be different than the value of this statistic.

The runstats utility collects the following information about each XML column:

- The number of NULL XML documents
- The number of non-NULL XML documents
- The number of distinct paths
- The sum of the node count for each distinct path
- The sum of the document count for each distinct path
- The *k* pairs of (path, node count) with the largest node count
- The *k* pairs of (path, document count) with the largest document count
- The *k* triples of (path, value, node count) with the largest node count
- The *k* triples of (path, value, document count) with the largest document count
- For each distinct path that leads to a text or attribute value:
  - The number of distinct values that this path can take
  - The highest value
  - The lowest value
  - The number of text or attribute nodes
  - The number of documents that contain the text or attribute nodes

Each row in an XML column stores an XML document. The node count for a path or path-value pair refers to the number of nodes that are reachable by that path or path-value pair. The document count for a path or path-value pair refers to the number of documents that contain that path or path-value pair.

For Db2 V9.7 Fix Pack 1 and later releases, the following apply to the collection of distribution statistics on an XML column:

- Distribution statistics are collected for each index over XML data specified on an XML column.
- The runstats utility must collect both distribution statistics and table statistics to collect distribution statistics for an index over XML data. Table statistics must be gathered in order for distribution statistics to be collected since XML distribution statistics are stored with table statistics.

  Collecting only index statistics, or collecting index statistics during index creation, will not collect distribution statistics for an index over XML data.

  As the default, the runstats utility collects a maximum of 250 quantiles for distribution statistics for each index over XML data. The maximum number of quantiles for a column can be specified when executing the runstats utility.
- Distribution statistics are collected for indexes over XML data of type VARCHAR, DOUBLE, TIMESTAMP, and DATE. XML distribution statistics are not collected for indexes over XML data of type VARCHAR HASHED.

- XML distribution statistics are collected when automatic table runstats operations are performed.
- XML distribution statistics are not created when loading data with the STATISTICS option.
- XML distribution statistics are not collected for partitioned indexes over XML data defined on a partitioned table.

The runstats utility collects the following information about column groups:

- A timestamp-based name for the column group
- The cardinality of the column group

The runstats utility collects the following information about indexes:

- The number of index entries (the *index cardinality*)
- The number of leaf pages
- The number of index levels
- The degree of clustering of the table data to the index
- The degree of clustering of the index keys with regard to data partitions
- The ratio of leaf pages located on disk in index key order to the number of pages in the range of pages occupied by the index
- The number of distinct values in the first column of the index
- The number of distinct values in the first two, three, and four columns of the index
- The number of distinct values in all columns of the index
- The number of leaf pages located on disk in index key order, with few or no large gaps between them
- The average leaf key size, without include columns
- The average leaf key size, with include columns
- The number of pages on which all record identifiers (RIDs) are marked deleted
- The number of RIDs that are marked deleted on pages where not all RIDs are marked deleted

If you request detailed index statistics, additional information about the degree of clustering of the table data to the index, and the page fetch estimates for different buffer sizes, is collected.

For a partitioned index, these statistics are representative of a single index partition, with the exception of the distinct values in the first column of the index; the first two, three, and four columns of the index; and in all columns of the index. Per-index partition statistics are also collected for the purpose of determining whether an index partition needs to be reorganized.

Statistics collection invalidates cached dynamic statements that reference tables for which statistics have been collected. This is done so that cached dynamic statements can be re-optimized with the latest statistics.

Catalog statistics tables:

Statistical information about the size of database tables, indexes, and statistical views is stored in system catalog tables.

The following tables provide a brief description of this statistical information and show where it is stored.

- The "Table" column indicates whether a particular statistic is collected if the **FOR INDEXES** or **AND INDEXES** parameter on the **RUNSTATS** command is not specified.
- The "Indexes" column indicates whether a particular statistic is collected if the **FOR INDEXES** or **AND INDEXES** parameter is specified.

Some statistics can be provided only by the table, some can be provided only by the indexes, and some can be provided by both.

- Table 1. Table Statistics (SYSCAT.TABLES and SYSSTAT.TABLES)
- Table 2. Column Statistics (SYSCAT.COLUMNS and SYSSTAT.COLUMNS)
- Table 3. Multi-column Statistics (SYSCAT.COLGROUPS and SYSSTAT.COLGROUPS)
- Table 4. Multi-column Distribution Statistics (SYSCAT.COLGROUPDIST and SYSSTAT.COLGROUPDIST)
- Table 5. Multi-column Distribution Statistics (SYSCAT.COLGROUPDISTCOUNTS and SYSSTAT.COLGROUPDISTCOUNTS)
- Table 6. Index Statistics (SYSCAT.INDEXES and SYSSTAT.INDEXES)
- Table 7. Column Distribution Statistics (SYSCAT.COLDIST and SYSSTAT.COLDIST)

The multi-column distribution statistics listed in Table 4. Multi-column Distribution Statistics (SYSCAT.COLGROUPDIST and SYSSTAT.COLGROUPDIST) and Table 5. Multi-column Distribution Statistics (SYSCAT.COLGROUPDISTCOUNTS and SYSSTAT.COLGROUPDISTCOUNTS) are not collected by the **RUNSTATS** utility. You cannot update them manually.

Table 71. Table Statistics (SYSCAT.TABLES and SYSSTAT.TABLES)

Statistic	Description	RUNSTATS Option	
		Table	**Indexes**
FPAGES	Number of pages being used by a table	Yes	Yes
NPAGES	Number of pages containing rows	Yes	Yes
OVERFLOW	Number of rows that overflow	Yes	No
CARD	Number of rows in a table (cardinality)	Yes	Yes (Note 1)
ACTIVE_BLOCKS	For MDC tables, the total number of occupied blocks	Yes	No

Note:

1. If the table has no indexes defined and you request statistics for indexes, no CARD statistics are updated. The previous CARD statistics are retained.

Table 72. Column Statistics (SYSCAT.COLUMNS and SYSSTAT.COLUMNS)

Statistic	Description	RUNSTATS Option	
		Table	**Indexes**
COLCARD	Column cardinality	Yes	Yes (Note 1)
AVGCOLLEN	Average length of a column	Yes	Yes (Note 1)
HIGH2KEY	Second highest value in a column	Yes	Yes (Note 1)

Table 72. Column Statistics (SYSCAT.COLUMNS and SYSSTAT.COLUMNS) (continued)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
LOW2KEY	Second lowest value in a column	Yes	Yes (Note 1)
NUMNULLS	The number of null values in a column	Yes	Yes (Note 1)
SUB_COUNT	The average number of sub-elements	Yes	No (Note 2)
SUB_DELIM_LENGTH	The average length of each delimiter separating sub-elements	Yes	No (Note 2)

Note:

1. Column statistics are collected for the first column in the index key.
2. These statistics provide information about data in columns that contain a series of sub-fields or sub-elements that are delimited by blanks. The SUB_COUNT and SUB_DELIM_LENGTH statistics are collected only for columns of type CHAR and VARCHAR with a code page attribute of single-byte character set (SBCS), FOR BIT DATA, or UTF-8.

Table 73. Multi-column Statistics (SYSCAT.COLGROUPS and SYSSTAT.COLGROUPS)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
COLGROUPCARD	Cardinality of the column group	Yes	No

Table 74. Multi-column Distribution Statistics (SYSCAT.COLGROUPDIST and SYSSTAT.COLGROUPDIST)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
TYPE	F = Frequency value Q = Quantile value	Yes	No
ORDINAL	Ordinal number of the column in the group	Yes	No
SEQNO	Sequence number *n* that represents the *n*th TYPE value	Yes	No
COLVALUE	The data value as a character literal or a null value	Yes	No

Table 75. Multi-column Distribution Statistics (SYSCAT.COLGROUPDISTCOUNTS and SYSSTAT.COLGROUPDISTCOUNTS)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
TYPE	F = Frequency value Q = Quantile value	Yes	No

Table 75. Multi-column Distribution Statistics (SYSCAT.COLGROUPDISTCOUNTS and SYSSTAT.COLGROUPDISTCOUNTS) (continued)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
SEQNO	Sequence number *n* that represents the *n*th TYPE value	Yes	No
VALCOUNT	If TYPE = F, VALCOUNT is the number of occurrences of COLVALUE for the column group with this SEQNO. If TYPE = Q, VALCOUNT is the number of rows whose value is less than or equal to COLVALUE for the column group with this SEQNO.	Yes	No
DISTCOUNT	If TYPE = Q, this column contains the number of distinct values that are less than or equal to COLVALUE for the column group with this SEQNO. Null if unavailable.	Yes	No

Table 76. Index Statistics (SYSCAT.INDEXES and SYSSTAT.INDEXES)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
NLEAF	Number of index leaf pages	No	Yes
NLEVELS	Number of index levels	No	Yes
CLUSTERRATIO	Degree of clustering of table data	No	Yes (Note 2)
CLUSTERFACTOR	Finer degree of clustering	No	Detailed (Notes 1,2)
DENSITY	Ratio (percentage) of SEQUENTIAL_PAGES to number of pages in the range of pages that is occupied by the index (Note 3)	No	Yes
FIRSTKEYCARD	Number of distinct values in the first column of the index	No	Yes

Table 76. Index Statistics (SYSCAT.INDEXES and SYSSTAT.INDEXES) (continued)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
FIRST2KEYCARD	Number of distinct values in the first two columns of the index	No	Yes
FIRST3KEYCARD	Number of distinct values in the first three columns of the index	No	Yes
FIRST4KEYCARD	Number of distinct values in the first four columns of the index	No	Yes
FULLKEYCARD	Number of distinct values in all columns of the index, excluding any key value in an index for which all record identifiers (RIDs) are marked deleted	No	Yes
PAGE_FETCH_PAIRS	Page fetch estimates for different buffer sizes	No	Detailed (Notes 1,2)
AVGPARTITION_CLUSTERRATIO	Degree of data clustering within a single data partition	No	Yes (Note 2)
AVGPARTITION_CLUSTERFACTOR	Finer measurement of degree of clustering within a single data partition	No	Detailed (Notes 1,2)
AVGPARTITION_PAGE_FETCH_PAIRS	Page fetch estimates for different buffer sizes, generated on the basis of a single data partition	No	Detailed (Notes 1,2)
DATAPARTITION_CLUSTERFACTOR	Number of data partition references during an index scan	No (Note 6)	Yes (Note 6)
SEQUENTIAL_PAGES	Number of leaf pages located on disk in index key order, with few, or no large gaps between them	No	Yes
AVERAGE_SEQUENCE_PAGES	Average number of index pages that are accessible in sequence; this is the number of index pages that the prefetchers can detect as being in sequence	No	Yes

Table 76. Index Statistics (SYSCAT.INDEXES and SYSSTAT.INDEXES) (continued)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
AVERAGE_RANDOM_PAGES	Average number of random index pages between sequential page accesses	No	Yes
AVERAGE_SEQUENCE_GAP	Gap between sequences	No	Yes
AVERAGE_SEQUENCE_FETCH_PAGES	Average number of table pages that are accessible in sequence; this is the number of table pages that the prefetchers can detect as being in sequence when they fetch table rows using the index	No	Yes (Note 4)
AVERAGE_RANDOM_FETCH_PAGES	Average number of random table pages between sequential page accesses when fetching table rows using the index	No	Yes (Note 4)
AVERAGE_SEQUENCE_FETCH_GAP	Gap between sequences when fetching table rows using the index	No	Yes (Note 4)
NUMRIDS	The number of RIDs in the index, including deleted RIDs	No	Yes
NUMRIDS_DELETED	The total number of RIDs in the index that are marked deleted, except RIDs on those leaf pages where all RIDs are marked deleted	No	Yes
NUM_EMPTY_LEAFS	The total number of leaf pages on which all RIDs are marked deleted	No	Yes
INDCARD	Number of index entries (index cardinality)	No	Yes

Table 76. Index Statistics (SYSCAT.INDEXES and SYSSTAT.INDEXES) (continued)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
Note: 1. Detailed index statistics are collected by specifying the DETAILED clause on the **RUNSTATS** command. 2. CLUSTERFACTOR and PAGE_FETCH_PAIRS are not collected with the **DETAILED** clause unless the table is of sufficient size (greater than about 25 pages). In this case, CLUSTERRATIO is -1 (not collected). If the table is relatively small, only CLUSTERRATIO is collected by the **RUNSTATS** utility; CLUSTERFACTOR and PAGE_FETCH_PAIRS are not collected. If the **DETAILED** clause is not specified, only CLUSTERRATIO is collected. 3. This statistic measures the percentage of pages in the file containing the index that belongs to that table. For a table with only one index defined on it, DENSITY should be 100. DENSITY is used by the optimizer to estimate how many irrelevant pages from other indexes might be read, on average, if the index pages were prefetched. 4. This statistic cannot be computed when the table is in a DMS table space. 5. Prefetch statistics are not collected during a load or create index operation, even if statistics collection is specified when the command is invoked. Prefetch statistics are also not collected if the **seqdetect** database configuration parameter is set to NO. 6. When **RUNSTATS** options for table is "No", statistics are not collected when table statistics are collected; when **RUNSTATS** options for indexes is "Yes", statistics are collected when the **RUNSTATS** command is used with the **INDEXES** options.			

Table 77. Column Distribution Statistics (SYSCAT.COLDIST and SYSSTAT.COLDIST)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
DISTCOUNT	If TYPE = Q, DISTCOUNT is the number of distinct values that are less than or equal to COLVALUE statistics	Distribution (Note 2)	No
TYPE	Indicator of whether the row provides frequent-value or quantile statistics	Distribution	No
SEQNO	Frequency ranking of a sequence number to help uniquely identify the row in the table	Distribution	No
COLVALUE	Data value for which a frequency or quantile statistic is collected	Distribution	No
VALCOUNT	Frequency with which the data value occurs in a column; for quantiles, the number of values that are less than or equal to the data value (COLVALUE)	Distribution	No

Table 77. Column Distribution Statistics (SYSCAT.COLDIST and SYSSTAT.COLDIST) (continued)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
Note: 1. Column distribution statistics are collected by specifying the **WITH DISTRIBUTION** clause on the **RUNSTATS** command. Distribution statistics cannot be collected unless there is a sufficient lack of uniformity in the column values. 2. DISTCOUNT is collected only for columns that are the first key column in an index.			

Automatic statistics collection:

The Db2 optimizer uses catalog statistics to determine the most efficient access plan for a query. With automatic statistics collection, part of the Db2 automated table maintenance feature, you can let the database manager determine whether statistics must be updated.

Instead of using automatic statistic collection, you can collect statistics manually. However, deciding which statistics to collect for a workload is complex, and keeping these statistics up-to-date is time consuming. Out-of-date or incomplete table or index statistics might lead the optimizer to select a suboptimal plan, which slows down query execution.

Automatic statistics collection can occur in two ways:

- For synchronous collection at statement compilation time, you can use the real-time statistics (RTS) feature. The **auto_stmt_stats** database configuration parameter is used for RTS statistics collection.
- For asynchronous collection, you can enable the **RUNSTATS** command to run in the background. The **auto_runstats** database configuration parameter is used for automatic background statistics collection.

Both of these parameters are enabled by default when you create a database. Although background statistics collection can be enabled while real-time statistics collection is disabled, background statistics collection is always enabled when real-time statistics collection occurs.

You can use the Configuration Advisor to determine the initial configuration for databases, including the appropriate settings for various database configuration parameter.

In IBM Data Studio Version 3.1 or later, you can use the task assistant for configuring automatic statistics collection. Task assistants can guide you through the process of setting options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.In IBM Data Studio Version 3.1 or later, you can use the task assistant to configure automatic statistics collection. Task assistants can guide you through the process of setting options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

Understanding asynchronous and real-time statistics collection

The query optimizer determines how synchronous or asynchronous statistics are collected, based on the needs of the query and the amount of table update activity (the number of update, insert, or delete operations).

You can enable real-time statistics collection, so that statistics can be fabricated by using certain metadata. *Fabrication* means deriving or creating statistics, rather than collecting them as part of normal **RUNSTATS** command activity. For example, the number of rows in a table can be derived from knowing the number of pages in the table, the page size, and the average row width. In some cases, statistics are not derived but are maintained by the index and data manager and can be stored in the catalog. For example, the index manager maintains a count of the number of leaf pages and levels in each index.

Real-time statistics collection provides more timely and more accurate statistics than asynchronous statistics collection. Accurate statistics can result in better query execution plans and improved performance. Regardless of whether you enable real-time statistics collection, asynchronous statistics collection occurs at 2-hour intervals. This interval might not be frequent enough to provide accurate statistics for some applications. Real-time or synchronous statistics collection also initiates asynchronous collection requests in the following cases:

- Synchronous statistics collection is used for sampling because the table is large.
- Synchronous statistics were fabricated.
- Synchronous statistics collection failed because the collection time is exceeded.

In addition, table activity might be high enough to require asynchronous collection, but might not be high enough to require synchronous statistics collection.

At most, two asynchronous requests can be processed at the same time, and only for different tables. One request must have been initiated by real-time statistics collection, and the other must have been initiated by asynchronous statistics collection.

The performance impact of automatic statistics collection is minimized in several ways:

- Asynchronous statistics collection is performed by using a throttled **RUNSTATS** utility. Throttling controls the amount of resource that is consumed by the **RUNSTATS** utility, based on current database activity. As database activity increases, the utility runs more slowly, reducing its resource demands.
- Synchronous statistics collection is limited to 5 seconds per query. The RTS optimization guideline determines the amount of time. If synchronous collection exceeds the time limit, an asynchronous collection request is submitted.
- Synchronous statistics collection does not store the statistics in the system catalog. Instead, the statistics are stored in a statistics cache and are later stored in the system catalog by an asynchronous operation. This storage sequence avoids the memory usage and possible lock contention that are involved in updating the system catalog. Statistics in the statistics cache are available for subsequent SQL compilation requests.
- Only one synchronous statistics collection operation occurs per table. Other requests requiring synchronous statistics collection fabricate statistics, if possible, and continue with statement compilation. This behavior is also enforced in a partitioned database environment, where operations on different database partitions might require synchronous statistics.

-
- Only tables with missing statistics or high levels of activity (as measured by the number of update, insert, or delete operations) are considered for statistics collection. Even if a table meets the statistics collection criteria, statistics are not collected synchronously unless query optimization requires them. In some cases, the query optimizer can choose an access plan without statistics. To check if asynchronous statistics collection is required, tables with more than 4000 pages are sampled to determine whether high table activity changed the statistics. Statistics for such large tables are collected only if warranted.
- Statistics collection during an online maintenance window depends on whether the statistics are asynchronous or synchronous:
  - For asynchronous statistics collection, the **RUNSTATS** utility is automatically scheduled to run during the online maintenance window that you specify in your maintenance policy. This policy also specifies the set of tables that are within the scope of automatic statistics collection, minimizing unnecessary resource consumption.
  - Synchronous statistics collection and fabrication do not use the online maintenance window that you specify in your maintenance policy, because synchronous requests must occur immediately and have limited collection time. Instead, synchronous statistics collection and fabrication uses to the policy that specifies the set of tables that are within the scope of automatic statistics collection.
- While automatic statistics collection is being performed, the affected tables are still available for regular database activity (update, insert, or delete operations).
- Synchronous are not collected for the following objects:
  - Real-time statistics are not collected for statistical views.
  - Real-time statistics are not collected for nicknames. To refresh nickname statistics in the system catalog for synchronous statistics collection, call the SYSPROC.NNSTAT procedure. For asynchronous statistics collection, Db2 software automatically calls the SYSPROC.NNSAT procedure to refresh the nickname statistics in the system catalog.
- Declared temporary tables (DGTTs) can have only real-time statistics collected.

Although real-time statistics collection is designed to minimize statistics collection memory usage, try it in a test environment first to ensure that there is no negative performance impact. There might be a negative performance impact in some online transaction processing (OLTP) scenarios, especially if there is a limit on how long a query can run.

Real-time synchronous statistics collection is performed for regular tables, materialized query tables (MQTs), and global temporary tables. Asynchronous statistics are not collected for global temporary tables. Global temporary tables cannot be excluded from real-time statistics via the automatic maintenance policy facility.

Automatic statistics collection (synchronous or asynchronous) does not occur for the following objects:

- Tables that are marked VOLATILE (tables that have the VOLATILE field set in the SYSCAT.TABLES catalog view).
- Created temporary tables (CGTTs).
- Tables for which you manually updated statistics by issuing UPDATE statements against SYSSTAT catalog views, including manual updates of expression-based

key column statistics for any expression-based indexes that the table has, even though those statistics are in a separate statistical view.

When you modify table statistics manually, the database manager assumes that you are now responsible for maintaining the statistics. To induce the database manager to maintain statistics for a table with manually updated statistics, collect the statistics by using the **RUNSTATS** command, or specify statistics collection when using the **LOAD** command. Tables that you created before Version 9.5 and for which you manually updated statistics before upgrading are not affected by this limitation. Their statistics are automatically maintained by the database manager until you manually update them.

Statistics fabrication does not occur for the following objects:

- Statistical views
- Tables for which you manually updated statistics by issuing UPDATE statements against SYSSTAT catalog views. If real-time statistics collection is not enabled, some statistics fabrication still occurs for tables for which you manually updated statistics.

In a partitioned database environment, statistics are collected on a single database partition and then extrapolated. The database manager always collects statistics (both RTS and automatic background statistics) on the first database partition of the database partition group.

No real-time statistics collection occurs until at least 5 minutes after database activation.

Real-time statistics processing occurs for both static and dynamic SQL.

A table that you truncated, either by using the TRUNCATE statement or by using the **IMPORT** command, is automatically recognized as having out-of-date statistics.

Automatic statistics collection, both synchronous and asynchronous, invalidates cached dynamic statements that reference tables for which statistics were collected. This invalidation is done so that cached dynamic statements can be re-optimized with the latest statistics.

Asynchronous automatic statistics collection operations might be interrupted when the database is deactivated. If you did not explicitly activate the database by using the **ACTIVATE DATABASE** command or the **sqle_activate_db** API, the database is deactivated when the last user disconnects from the database. If operations are interrupted, error messages might be recorded in the Db2 diagnostic log file. To avoid interrupting asynchronous automatic statistics collection operations, explicitly activate the database.

If a table has expression-based indexes, statistics for the expression-based key columns are collected and cached as part of statistics collection for the table. Statistics are not fabricated for expression-based key columns

Real-time statistics and explain processing

There is no real-time processing for a query that is only explained (not executed) by the **EXPLAIN** facility. The following table summarizes the behavior under different values of the **CURRENT EXPLAIN MODE** special register.

Table 78. Real-time statistics collection as a function of the value of the CURRENT EXPLAIN MODE special register

CURRENT EXPLAIN MODE special register value	Real-time statistics collection considered
YES	Yes
EXPLAIN	No
NO	Yes
REOPT	Yes
RECOMMEND INDEXES	No
EVALUATE INDEXES	No

Automatic statistics collection and the statistics cache

You can use the statistics cached to make synchronously collected statistics available to all queries. This cache is part of the catalog cache. In a partitioned database environment, the statistics cache is on only the catalog database partition even though each database partition has a catalog cache. If you enable real-time statistics collection, catalog cache requirements are higher. Consider tuning the value of the **catalogcache_sz** database configuration parameter if you enable real-time statistics collection.

Automatic statistics collection and statistical profiles

You can customize the type of statistics that are collected by creating your own statistics profile for a particular table. For details, see collecting statistics using a statistics profile.

RTS and automatic background statistics are collected according to a statistical profile that is in effect for a table, with the following exceptions:

- To minimize the memory usage of synchronous statistics collection, the database manager might collect statistics by using sampling. In this case, the sampling rate and method might be different from those rates and methods that you specified in the statistical profile.
- RTS collection might fabricate statistics, but it might not be possible to fabricate all types of statistics that you specified in the statistical profile. For example, column statistics such as COLCARD, HIGH2KEY, and LOW2KEY cannot be fabricated unless the column is the primary column in some index.

If RTS statistics collection cannot gather all the statistics that you specified in the statistical profile, an asynchronous collection request is submitted.

The following sections explain different operating characteristics of automatic statistics collection.

Enabling automatic statistics collection:

Having accurate and complete database statistics is critical to efficient data access and optimal workload performance. Use the automatic statistics collection feature of the automated table maintenance functionality to update and maintain relevant database statistics.

About this task

To enable automatic statistics collection, you must first configure your database by setting the **auto_maint** and the **auto_tbl_maint** database configuration parameters to ON.

Procedure

After setting the **auto_maint** and the **auto_tbl_maint** database configuration parameters to ON, you have the following options:

- To enable background statistics collection, set the **auto_runstats** database configuration parameter to ON.
- To enable background statistics collection for statistical views, set both the **auto_stats_views** and **auto_runstats** database configuration parameters to ON.
- To enable background statistics collection to use sampling automatically for large tables and statistical views, also set the **auto_sampling** database configuration parameter to ON. Use this setting in addition to **auto_runstats** (tables only) or to **auto_runstats** and **auto_stats_views** (tables and statistical views).
- To enable real-time statistics collection, set both **auto_stmt_stats** and **auto_runstats** database configuration parameters to ON.

Collecting statistics using a statistics profile:

The **RUNSTATS** utility provides the option to register and use a statistics profile, which specifies the type of statistics that are to be collected for a particular table; for example, table statistics, index statistics, or distribution statistics. This feature simplifies statistics collection by enabling you to store **RUNSTATS** options for convenient future use.

To register a profile and collect statistics at the same time, issue the **RUNSTATS** command with the **SET PROFILE** parameter. To register a profile only, issue the **RUNSTATS** command with the **SET PROFILE ONLY** parameter. To collect statistics using a profile that was already registered, issue the **RUNSTATS** command with the **USE PROFILE** parameter.

To see what options are currently specified in the statistics profile for a particular table, query the SYSCAT.TABLES catalog view. For example:

```
SELECT STATISTICS_PROFILE FROM SYSCAT.TABLES WHERE TABNAME = 'EMPLOYEE'
```

Storage used by automatic statistics collection and reorganization:

The automatic statistics collection and reorganization features store working data in tables that are part of your database. These tables are created in the SYSTOOLSPACE table space.

SYSTOOLSPACE is created automatically with default options when the database is activated. Storage requirements for these tables are proportional to the number of tables in the database and can be estimated at approximately 1 KB per table. If this is a significant size for your database, you might want to drop and then recreate the table space yourself, allocating storage appropriately. Although the automatic maintenance and health monitoring tables in the table space are automatically recreated, any history that was captured in those tables is lost when you drop the table space.

Automatic statistics collection activity logging:

The statistics log is a record of all of the statistics collection activities (both manual and automatic) that have occurred against a specific database.

The default name of the statistics log is `db2optstats.`*`number`*`.log`. It resides in the `$`**`diagpath`**`/events` directory. The statistics log is a rotating log. Log behavior is controlled by the **DB2_OPTSTATS_LOG** registry variable.

The statistics log can be viewed directly or it can be queried using the SYSPROC.PD_GET_DIAG_HIST table function. This table function returns a number of columns containing standard information about any logged event, such as the timestamp, Db2 instance name, database name, process ID, process name, and thread ID. The log also contains generic columns for use by different logging facilities. The following table describes how these generic columns are used by the statistics log.

Table 79. Generic columns in the statistics log file

Column name	Data type	Description
OBJTYPE	VARCHAR(64)	The type of object to which the event applies. For statistics logging, this is the type of statistics to be collected. OBJTYPE can refer to a statistics collection background process when the process starts or stops. It can also refer to activities that are performed by automatic statistics collection, such as a sampling test, initial sampling, and table evaluation. Possible values for statistics collection activities are: **TABLE STATS** Table statistics are to be collected. **INDEX STATS** Index statistics are to be collected. **TABLE AND INDEX STATS** Both table and index statistics are to be collected. Possible values for automatic statistics collection are: **EVALUATION** The automatic statistics background collection process has begun an evaluation phase. During this phase, tables will be checked to determine if they need updated statistics, and statistics will be collected, if necessary. **INITIAL SAMPLING** Statistics are being collected for a table using sampling. The sampled statistics are stored in the system catalog. This allows automatic statistics collection to proceed quickly for a table with no statistics. Subsequent operations will collect statistics without sampling. Initial sampling is performed during the evaluation phase of automatic statistics collection. **SAMPLING TEST** Statistics are being collected for a table using sampling. The sampled statistics are not stored in the system catalog. The sampled statistics will be compared to the current catalog statistics to determine if and when full statistics should be collected for this table. The sampling is performed during the evaluation phase of automatic statistics collection. **STATS DAEMON** The statistics daemon is a background process used to handle requests that are submitted by real-time statistics processing. This object type is logged when the background process starts or stops.
OBJNAME	VARCHAR(255)	The name of the object to which the event applies, if available. For statistics logging, this is the table or index name. If OBJTYPE is STATS DAEMON or EVALUATION, OBJNAME is the database name and OBJNAME_QUALIFIER is NULL.
OBJNAME_QUALIFIER	VARCHAR(255)	For statistics logging, this is the schema of the table or index.

Table 79. Generic columns in the statistics log file (continued)

Column name	Data type	Description
EVENTTYPE	VARCHAR(24)	The event type is the action that is associated with this event. Possible values for statistics logging are: **COLLECT** This action is logged for a statistics collection operation. **START** This action is logged when the real-time statistics background process (OBJTYPE = STATS DAEMON) or an automatic statistics collection evaluation phase (OBJTYPE = EVALUATION) starts. **STOP** This action is logged when the real-time statistics background process (OBJTYPE = STATS DAEMON) or an automatic statistics collection evaluation phase (OBJTYPE = EVALUATION stops. **ACCESS** This action is logged when an attempt has been made to access a table for statistics collection purposes. This event type is used to log an unsuccessful access attempt when the object is unavailable. **WRITE** This action is logged when previously collected statistics that are stored in the statistics cache are written to the system catalog.
FIRST_EVENTQUALIFIERTYPE	VARCHAR(64)	The type of the first event qualifier. Event qualifiers are used to describe what was affected by the event. For statistics logging, the first event qualifier is the timestamp for when the event occurred. For the first event qualifier type, the value is AT.
FIRST_EVENTQUALIFIER	CLOB(16k)	The first qualifier for the event. For statistics logging, the first event qualifier is the timestamp for when the statistics event occurred. The timestamp of the statistics event might be different than the timestamp of the log record, as represented by the TIMESTAMP column.
SECOND_EVENTQUALIFIERTYPE	VARCHAR(64)	The type of the second event qualifier. For statistics logging, the value can be BY or NULL. This field is not used for other event types.

Table 79. Generic columns in the statistics log file (continued)

Column name	Data type	Description
SECOND_EVENTQUALIFIER	CLOB(16k)	The second qualifier for the event. For statistics logging, this represents how statistics were collected for COLLECT event types. Possible values are: **User** Statistics collection was performed by a Db2 user invoking the **LOAD**, **REDISTRIBUTE**, or **RUNSTATS** command, or issuing the CREATE INDEX statement. **Synchronous** Statistics collection was performed at SQL statement compilation time by the Db2 server. The statistics are stored in the statistics cache but not the system catalog. **Synchronous sampled** Statistics collection was performed using sampling at SQL statement compilation time by the Db2 server. The statistics are stored in the statistics cache but not the system catalog. **Fabricate** Statistics were fabricated at SQL statement compilation time using information that is maintained by the data and index manager. The statistics are stored in the statistics cache but not the system catalog. **Fabricate partial** Only some statistics were fabricated at SQL statement compilation time using information that is maintained by the data and index manager. In particular, only the HIGH2KEY and LOW2KEY values for certain columns were fabricated. The statistics are stored in the statistics cache but not the system catalog. **Asynchronous** Statistics were collected by a Db2 background process and are stored in the system catalog. This field is not used for other event types.
THIRD_EVENTQUALIFIERTYPE	VARCHAR(64)	The type of the third event qualifier. For statistics logging, the value can be DUE TO or NULL.

Table 79. Generic columns in the statistics log file (continued)

Column name	Data type	Description
THIRD_EVENTQUALIFIER	CLOB(16k)	The third qualifier for the event. For statistics logging, this represents the reason why a statistics activity could not be completed. Possible values are: **Timeout** Synchronous statistics collection exceeded the time budget. **Error** The statistics activity failed due to an error. **RUNSTATS error** Synchronous statistics collection failed due to a **RUNSTATS** error. For some errors, SQL statement compilation might have completed successfully, even though statistics could not be collected. For example, if there was insufficient memory to collect statistics, SQL statement compilation will continue. **Object unavailable** Statistics could not be collected for the database object because it could not be accessed. Some possible reasons include: • The object is locked in super exclusive (Z) mode • The table space in which the object resides is unavailable • The table indexes need to be recreated **Conflict** Synchronous statistics collection was not performed because another application was already collecting synchronous statistics. Check the FULLREC column or the **db2diag** log files for the error details.
EVENTSTATE	VARCHAR(255)	State of the object or action as a result of the event. For statistics logging, this indicates the state of the statistics operation. Possible values are: • Start • Success • Failure

Example

In this example, the query returns statistics log records for events up to one year prior to the current timestamp by invoking PD_GET_DIAG_HIST.

```
select pid, tid,
    substr(eventtype, 1, 10),
    substr(objtype, 1, 30) as objtype,
    substr(objname_qualifier, 1, 20) as objschema,
    substr(objname, 1, 10) as objname,
    substr(first_eventqualifier, 1, 26) as event1,
    substr(second_eventqualifiertype, 1, 2) as event2_type,
    substr(second_eventqualifier, 1, 20) as event2,
    substr(third_eventqualifiertype, 1, 6) as event3_type,
    substr(third_eventqualifier, 1, 15) as event3,
    substr(eventstate, 1, 20) as eventstate
  from table(sysproc.pd_get_diag_hist
    ('optstats', 'EX', 'NONE',
      current_timestamp - 1 year, cast(null as timestamp))) as sl
  order by timestamp(varchar(substr(first_eventqualifier, 1, 26), 26));
```

The results are ordered by the timestamp stored in the FIRST_EVENTQUALIFIER column, which represents the time of the statistics event.

```
PID   TID          EVENTTYPE OBJTYPE              OBJSCHEMA OBJNAME    EVENT1                     EVENT2_ EVENT2              EVENT3_ EVENT3   EVENTSTATE
                                                                                                   TYPE                        TYPE
----- ------------ --------- -------------------- --------- -------- -------------------------- ------- ------------------- ------- -------- ----------
28399    1082145120 START     STATS DAEMON         -         PROD_DB  2007-07-09-18.37.40.398905 -       -                   -       -        success
28389  183182027104 COLLECT   TABLE AND INDEX STATS DB2USER   DISTRICT 2007-07-09-18.37.43.261222 BY      Synchronous         -       -        start
28389  183182027104 COLLECT   TABLE AND INDEX STATS DB2USER   DISTRICT 2007-07-09-18.37.43.407447 BY      Synchronous         -       -        success
28399    1082145120 COLLECT   TABLE AND INDEX STATS DB2USER   CUSTOMER 2007-07-09-18.37.43.471614 BY      Asynchronous        -       -        start
28399    1082145120 COLLECT   TABLE AND INDEX STATS DB2USER   CUSTOMER 2007-07-09-18.37.43.524496 BY      Asynchronous        -       -        success
28399    1082145120 STOP      STATS DAEMON         -         PROD_DB  2007-07-09-18.37.43.526212 -       -                   -       -        success
28389  183278496096 COLLECT   TABLE STATS          DB2USER   ORDER_LINE 2007-07-09-18.37.48.676524 BY      Synchronous sampled -       -        start
28389  183278496096 COLLECT   TABLE STATS          DB2USER   ORDER_LINE 2007-07-09-18.37.53.677546 BY      Synchronous sampled DUE TO  Timeout  failure
28389    1772561034 START     EVALUATION           -         PROD_DB  2007-07-10-12.36.11.092739 -       -                   -       -        success
28389    8231991291 COLLECT   TABLE AND INDEX STATS DB2USER   DISTRICT 2007-07-10-12.36.30.737603 BY      Asynchronous        -       -        start
28389    8231991291 COLLECT   TABLE AND INDEX STATS DB2USER   DISTRICT 2007-07-10-12.36.34.029756 BY      Asynchronous        -       -        success
28389    1772561034 STOP      EVALUATION           -         PROD_DB  2007-07-10-12.36.39.685188 -       -                   -       -        success
28399    1504428165 START     STATS DAEMON         -         PROD_DB  2007-07-10-12.37.43.319291 -       -                   -       -        success
28399    1504428165 COLLECT   TABLE AND INDEX STATS DB2USER   CUSTOMER 2007-07-10-12.37.43.471614 BY      Asynchronous        -       -        start
28399    1504428165 COLLECT   TABLE AND INDEX STATS DB2USER   CUSTOMER 2007-07-10-12.37.44.524496 BY      Asynchronous        -       -        failure
28399    1504428165 STOP      STATS DAEMON         -         PROD_DB  2007-07-10-12.37.45.905975 -       -                   -       -        success
28399    4769515044 START     STATS DAEMON         -         PROD_DB  2007-07-10-12.48.33.319291 -       -                   -       -        success
28389    4769515044 WRITE     TABLE AND INDEX STATS DB2USER   CUSTOMER 2007-07-10-12.48.33.969888 BY      Asynchronous        -       -        start
28389    4769515044 WRITE     TABLE AND INDEX STATS DB2USER   CUSTOMER 2007-07-10-12.48.34.215230 BY      Asynchronous        -       -        success
```

Improving query performance for large statistics logs:

If the statistics log files are large, you can improve query performance by copying the log records into a table, creating indexes, and then gathering statistics.

Procedure

1. Create a table with appropriate columns for the log records.

```
create table db2user.stats_log (
  pid         bigint,
  tid         bigint,
  timestamp   timestamp,
  dbname      varchar(128),
  retcode     integer,
  eventtype   varchar(24),
  objtype     varchar(30),
  objschema   varchar(20),
  objname     varchar(30),
  event1_type varchar(20),
  event1      timestamp,
  event2_type varchar(20),
  event2      varchar(40),
  event3_type varchar(20),
  event3      varchar(40),
  eventstate  varchar(20))
```

2. Declare a cursor for a query against SYSPROC.PD_GET_DIAG_HIST.

```
declare c1 cursor for
  select pid, tid, timestamp, dbname, retcode, eventtype,
      substr(objtype, 1, 30) as objtype,
      substr(objname_qualifier, 1, 20) as objschema,
      substr(objname, 1, 30) as objname,
      substr(first_eventqualifiertype, 1, 20),
      substr(first_eventqualifier, 1, 26),
      substr(second_eventqualifiertype, 1, 20),
      substr(second_eventqualifier, 1, 40),
      substr(third_eventqualifiertype, 1, 20),
      substr(third_eventqualifier, 1, 40),
      substr(eventstate, 1, 20)
    from table (sysproc.pd_get_diag_hist
      ('optstats', 'EX', 'NONE',
        current_timestamp - 1 year, cast(null as timestamp ))) as sl
```

3. Load the statistics log records into the table.

```
load from c1 of cursor replace into db2user.stats_log
```

4. Create indexes and then gather statistics on the table.

```
create index sl_ix1 on db2user.stats_log(eventtype, event1);
create index sl_ix2 on db2user.stats_log(objtype, event1);
create index sl_ix3 on db2user.stats_log(objname);

runstats on table db2user.stats_log
  with distribution and sampled detailed indexes all;
```

Statistics for expression-based indexes:

In an expression-based index, the key includes one or more expressions that consist of more than just a column name. An expression-based index can provide faster and more efficient access to data. However, before you implement this feature, consider how statistics are collected for expression-based indexes.

To benefit from expression-based indexes, you do not have to carry out any special activities. Statistics for expression-based key columns are collected along with other statistics on the table and its indexes. Typically, the same options that you apply to a table's columns are also suitable for the expression-based key columns. However, if you want to have more control, review the following topics:

RUNSTATS on expression-based indexes:

Issuing the **RUNSTATS** command on a table with expression-based indexes updates and stores the index statistics as usual. However, statistics for each expression-based key column within the index are also collected. The optimizer uses these statistics to choose query execution plans for queries that involve the expressions.

The statistics for the expression-based keys are collected and stored in a statistical view. This statistical view is automatically created when you create an expression-based index. Issuing the **RUNSTATS** command on a table and its expression-based index resembles issuing the **RUNSTATS** command on the table and then issuing it again on the statistical view. However, although you can issue the **RUNSTATS** command directly on the statistical view, you should issue the command on the table and its indexes instead. This results in the gathering and updating of the statistics that are stored in the statistical view and the statistics for the index and table. If you issue the **RUNSTATS** command on the statistical view, the statistical view statistics are updated, but the index and table statistics are untouched.

To issue the **RUNSTATS** command on an index with expression-based keys, explicitly include the index by name, or implicitly include the index by using the **AND INDEXES ALL** clause or the **FOR INDEXES ALL** clause. If you do not explicitly or implicitly include the index, the index and the statistical view statistics are not updated.

You cannot name expressions as columns for the **RUNSTATS** command. However, expressions are considered key columns. Therefore, to gather distribution statistics for expression-based columns, include the index in the **RUNSTATS** command, or use a statistics profile. In addition, specify that distribution statistics are to be gathered on all columns or on key columns. Statistical view statistics are updated for non-expression columns only if they are included in the **RUNSTATS** column specification. Statistical view statistics for the non-expression columns can be disregarded, as they are not referenced by the optimizer.

Expression-based key columns exist only in the index. They do not exist in the base table. Therefore, the **INDEXSAMPLE** clause, rather than the **TABLESAMPLE** clause, determines how the expression column data is sampled.

Example

The following conditions apply to all the examples:

- Table TBL1 is created with expression-based index IND1.
- Associated statistical view IND1_V is automatically created.

Example 1

The **RUNSTATS** command is issued on table TBL1 in two ways:

- No index is specified:

```
runstats on table TBL1
```

- An index that is not IND1 is specified:

```
runstats on table TBL1 with distribution on key columns and index IND2
```

The results in both cases are the same: the index statistics for IND1 are not updated, and the statistical view statistics for IND1_V are not updated, even though the **ON KEY COLUMNS** parameter was specified in the second case. (Specifying the **ALL COLUMNS** parameter would not change the results, either.) To gather statistics on expression-based key columns in an expression-based index, you must explicitly or implicitly include that index in the **RUNSTATS** command.

Example 2

The **RUNSTATS** command is issued on table TBL1 in three ways:

- Index IND1 is specified:

```
runstats on table TBL1 and index IND1
runstats on table TBL1 on columns (c1,c3) and indexes IND1,IND2
```

- The **ALL** parameter is specified:

```
runstats on table TBL1 on key columns and indexes all
```

In all of these cases, the index statistics for IND1 are updated. As well, the statistical view statistics for IND1_V are updated with basic column statistics for all expression columns. These results apply even though the **ON COLUMNS AND** clause was specified.

Expression-based indexes and statistics profiles:

The **RUNSTATS** command's statistics profile facility can automatically gather customized statistics for a particular table, including a table with an expression-based index. This facility simplifies statistics collection by storing the **RUNSTATS** command's options for convenient future use.

The **RUNSTATS** command provides the option to register and use a statistics profile, which specifies the type of statistics that are to be collected for a particular table. To register a profile and collect statistics at the same time, issue the **RUNSTATS** command with the **SET PROFILE** parameter. To register a profile only, issue the **RUNSTATS** command with the **SET PROFILE ONLY** parameter. To collect statistics with a profile that was already registered, issue the **RUNSTATS** command with the **USE PROFILE** parameter.

A statistics profile is useful if you want to gather nonstandard statistics for particular columns. However, an expression cannot be named as a column for a **RUNSTATS** command. Therefore, to gather nonstandard statistics, you would issue the **RUNSTATS** command on the base table and the statistical view. This strategy

raises the problem of keeping the table and statistical view statistics in sync. Instead, issuing the **RUNSTATS** command uses a table's statistics profile and, if that profile includes an index with expression-based keys in its specifications, then any statistics profile on the statistical view that is associated with that index is also applied. So, all statistics are gathered with a just one **RUNSTATS** operation on the table because all relevant profiles are applied.

If the table does not have a statistics profile that is associated with it (or you do not specify the **USE PROFILE** parameter), then no profile on the associated statistical views is applied while the **RUNSTATS** facility is running. This behavior applies only to the statistical views associated with indexes that includes expression-based keys and not statistical views in general.

Creating a statistics profile on a statistical view that is associated with an index, which includes expression-based keys, automatically creates a statistics profile on the base table (but only if one does not exist).

If a statistics profile is associated with the base table and a statistics profile is associated with the statistical view, the profiles are used as follows while the **RUNSTATS** command is running on a table with an expression-based index:

- The table's profile controls the statistics for the non-expression columns and the **RUNSTATS** command overall.
- The statistical view's statistics profile controls the statistics for the expression columns.

The **auto_runstats** database configuration parameter and real-time statistics (rts) feature apply the statistics profiles in the same manner as a manually issued **RUNSTATS** command. If the statistics profile for a statistical view exists and the underlying table has a statistical profile that includes an expression-based index, the **auto_runstats** database configuration parameter and the rts feature apply the statistics profile for the statistical view. Neither the **auto_runstats** database configuration parameter nor the rts feature act directly on the expression-based index's statistical view. The statistical view statistics are gathered when the **auto_runstats** database configuration parameter and the rts feature act on the table itself. In addition, the rts feature does not fabricate statistical view statistics for the expression-based index.

Example

The following initial conditions apply to the examples:

- Table TBL1 has no statistics profile.
- Table TBL1 has an expression-based index IND1.
- Index IND1 has an automatically generated statistical view IND1_V.
- The **NUM_FREQVALUES** database configuration parameter is set to the default value of 10.

Example 1

The following command sets a statistics profile on the statistical view IND1_V, which gathers extra distribution statistics:

```
RUNSTATS ON VIEW IND1_V WITH DISTRIBUTION DEFAULT NUM_FREQVALUES 40 SET PROFILE ONLY
```

Because there is no statistics profile on the table, a statistics profile is generated when the command is issued in the following form:

```
RUNSTATS ON TABLE TBL1 WITH DISTRIBUTION AND SAMPLED DETAILED INDEXES ALL
```

Now, the following command is issued:

```
RUNSTATS ON TABLE TBL1 USE PROFILE
```

The statistics profile on the table and the statistics profile on the statistic view are applied. Statistics are gathered for the table and the expression columns in the index. The columns in the table have the usual amount of frequent value statistics and the columns in the statistical view have more frequent value statistics.

Example 2

The following command sets a statistics profile that samples a large table with the aim to shorten the execution time for the **RUNSTATS** command and to reduce the impact on the overall system (the profile is set, but the command to sample the table is not issued):

```
RUNSTATS ON TABLE TBL1 WITH DISTRIBUTION AND INDEXES ALL TABLESAMPLE SYSTEM (2.5) INDEXSAMPLE SYSTEM (10) SET PROFILE ONLY UTIL_IMPACT_PRIORITY 30
```

It is decided that distribution statistics are not needed on the expression-based keys in index IND1. However, LIKE statistics are required on the second key column in the index. According to the definition for statistical view IND1_V in the catalogs, the second column in the view is named K01.

The following command is issued to modify the statistics profile to gather LIKE statistics on column K01 from statistical view IND1_V (the profile is set, but the command to gather statistics is not issued):

```
RUNSTATS ON VIEW IND1_V ON ALL COLUMNS AND COLUMNS(K01 LIKE STATISTICS) SET PROFILE ONLY
```

Now the statistics profile is set and the following command is issued to gather statistics that are based on the statistics profile:

```
RUNSTATS ON TABLE TBL1 USE PROFILE
```

The statistics profile on the table and the statistics profile on the statistic view are applied. Statistics are gathered for the table and the expression-based columns. The results are as follows:

- Distribution statistics are gathered for the base table columns but not for the expression-based key columns.
- The LIKE statistics are gathered for the specified expression-based key column.
- While the **RUNSTATS** command is running, the expression-based key column values are sampled at the rate dictated by the **INDEXSAMPLE SYSTEM(10)** parameter in the table's profile.
- The table's **UTIL_IMPACT_PRIORITY** parameter setting governs the priority of the entire **RUNSTATS** command operation.

Expression-based indexes and automatic statistics collection:

With automatic statistics collection, you can allow the database manager to determine whether to update the statistics for a table, including a table with expression-based indexes. The **auto_runstats** database configuration parameter and real-time statistics (rts) feature are enabled by default when you create a database.

The **auto_runstats** database configuration parameter and the rts feature apply the statistics profiles in the same manner as a manually issued **RUNSTATS** command. They apply the statistics profile of the statistical view if the following conditions are met:

- A statistical view with a statistics profile.
- A base table with a statistic profile.
- A base table that includes an expression-based index.

Neither the **auto_runstats** database configuration parameter nor the rts feature act directly on the automatically generated statistical view. The statistical view statistics are gathered when the **auto_runstats** database configuration parameter and the rts feature act on the table itself. The statistical view statistics are gathered regardless of whether statistics profile exist on the table or on the associated statistical views.

The rts feature does not fabricate statistical view statistics for the expression-based index.

Expression-based indexes and manual statistics updates:

You can manually update statistics for expression-based key columns by issuing the UPDATE statement against the SYSSTAT.TABLES, SYSSTAT.COLUMNS, and SYSSTAT.COLDIST catalog views.

When you issue the UPDATE statement against the SYSSTAT views, specify the statistical view's name and the name of the column as it appears in the statistical view.

Automatic statistics collection does not occur if you manually update a table's statistics by issuing an UPDATE statement against SYSSTAT catalog views. This condition includes manual updates of expression-based key column statistics for any expression-based indexes that the table has, even though those statistics are in a separate statistical view. Similarly, if you manually update any statistics in the statistical view that is associated with a table with an expression-based index, the underlying table is excluded from automatic statistics collection. To get automatic statistics collection to consider the table, issue the **RUNSTATS** command against the table rather than the statistical view.

Guidelines for collecting and updating statistics:

The **RUNSTATS** utility collects statistics on tables, indexes, and statistical views to provide the optimizer with accurate information for access plan selection.

Use the **RUNSTATS** utility to collect statistics in the following situations:

- After data is loaded into a table and appropriate indexes are created
- After creating an index on a table
- After a table is reorganized with the **REORG** utility
- After a table and its indexes are significantly modified through update, insert, or delete operations
- Before binding application programs whose performance is critical
- When you want to compare current and previous statistics
- When the prefetch value has changed
- After executing the **REDISTRIBUTE DATABASE PARTITION GROUP** command

- When you have XML columns. When **RUNSTATS** is used to collect statistics for XML columns only, existing statistics for non-XML columns that were collected during a load operation or a previous **RUNSTATS** operation are retained. If statistics on some XML columns were collected previously, those statistics are either replaced or dropped if the current **RUNSTATS** operation does not include those columns.

To improve **RUNSTATS** performance and save disk space used to store statistics, consider specifying only those columns for which data distribution statistics should be collected.

You should rebind application programs after executing **RUNSTATS**. The query optimizer might choose different access plans if new statistics are available.

If a full set of statistics cannot be collected at one time, use the **RUNSTATS** utility on subsets of the objects. If inconsistencies occur as a result of ongoing activity against those objects, a warning message (SQL0437W, reason code 6) is returned during query optimization. If this occurs, use **RUNSTATS** again to update the distribution statistics.

To ensure that index statistics are synchronized with the corresponding table, collect both table and index statistics at the same time. If a table was modified extensively since the last time that statistics were gathered, updating only the index statistics for that table leaves the two sets of statistics out of synchronization with each other.

Using the **RUNSTATS** utility on a production system might negatively affect workload performance. The utility now supports a throttling option that can be used to limit the performance impact of **RUNSTATS** execution during high levels of database activity.

When you collect statistics for a table in a partitioned database environment, **RUNSTATS** operates only on the database partition from which the utility is executed. The results from this database partition are extrapolated to the other database partitions. If this database partition does not contain a required portion of the table, the request is sent to the first database partition in the database partition group that contains the required data.

Statistics for a statistical view are collected on all database partitions containing base tables that are referenced by the view.

Consider the following tips to improve the efficiency of **RUNSTATS** and the usefulness of the statistics:

- Collect statistics only for columns that are used to join tables or for columns that are referenced in the WHERE, GROUP BY, or similar clauses of queries. If the columns are indexed, you can specify these columns with the **ONLY ON KEY COLUMNS** clause on the **RUNSTATS** command.
- Customize the values of the **num_freqvalues** and **num_quantiles** database configuration parameters for specific tables and columns.
- When you create an index for a populated table, use the COLLECT STATISTICS clause to create statistics as the index is created.
- When significant numbers of table rows are added or removed, or if data in columns for which you collect statistics is updated, use **RUNSTATS** again to update the statistics.

- Because **RUNSTATS** collects statistics on only a single database partition, the statistics are less accurate if the data is not distributed consistently across all database partitions. If you suspect that there is skewed data distribution, consider redistributing the data across database partitions by using the **REDISTRIBUTE DATABASE PARTITION GROUP** command before using the **RUNSTATS** utility.
- For Db2 V9.7 Fix Pack 1 and later releases, distribution statistics can be collected on an XML column. Distribution statistics are collected for each index over XML data specified on the XML column. By default, a maximum of 250 quantiles are used for distribution statistics for each index over XML data.

  When collecting distribution statistics on an XML column, you can change maximum number of quantiles. You can lower the maximum number of quantiles to reduce the space requirements for XML distribution statistics based on your particular data size, or you can increase the maximum number of quantiles if 250 quantiles are not sufficient to capture the distribution statistics of the data set for an index over XML data.

Collecting catalog statistics:

Use the **RUNSTATS** utility to collect catalog statistics on tables, indexes, and statistical views. The query optimizer uses this information to choose the best access plans for queries.

About this task

For privileges and authorities that are required to use this utility, see the description of the **RUNSTATS** command.

Procedure

To collect catalog statistics:

1. Connect to the database that contains the tables, indexes, or statistical views for which you want to collect statistical information.
2. Collect statistics for queries that run against the tables, indexes, or statistical views by using one of the following methods:
   - From the Db2 command line, execute the **RUNSTATS** command with appropriate options. These options enable you to tailor the statistics that are collected for queries that run against the tables, indexes, or statistical views.
   - From IBM Data Studio, open the task assistant for the **RUNSTATS** command.
3. When the runstats operation completes, issue a COMMIT statement to release locks.
4. Rebind any packages that access the tables, indexes, or statistical views for which you have updated statistical information.

Results

Note:

1. The **RUNSTATS** command does not support the use of nicknames. If queries access a federated database, use **RUNSTATS** to update statistics for tables in all databases, then drop and recreate the nicknames that access remote tables to make the new statistics available to the optimizer.
2. When you collect statistics for a table in a partitioned database environment, **RUNSTATS** only operates on the database partition from which the utility is executed. The results from this database partition are extrapolated to the other

database partitions. If this database partition does not contain a required portion of the table, the request is sent to the first database partition in the database partition group that contains the required data.

Statistics for a statistical view are collected on all database partitions containing base tables that are referenced by the view.

3. For Db2 V9.7 Fix Pack 1 and later releases, the following apply to the collection of distribution statistics on a column of type XML:
   - Distribution statistics are collected for each index over XML data specified on an XML column.
   - The **RUNSTATS** command must collect both distribution statistics and table statistics to collect distribution statistics for an index over XML data.
   - As the default, the **RUNSTATS** command collects a maximum of 250 quantiles for distribution statistics for each index over XML data. The maximum number of quantiles for a column can be specified when executing the **RUNSTATS** command.
   - Distribution statistics are collected on indexes over XML data of type VARCHAR, DOUBLE, TIMESTAMP, and DATE. XML distribution statistics are not collected on indexes over XML data of type VARCHAR HASHED.
   - Distribution statistics are not collected on partitioned indexes over XML data defined on a partitioned table.

Collecting statistics on a sample of the data:

Table statistics are used by the query optimizer to select the best access plan for a query, so it is important that statistics remain current. With the ever-increasing size of databases, efficient statistics collection becomes more challenging.

An effective approach is to collect statistics on a random sample of table and index data. For I/O-bound or processor-bound systems, the performance benefits can be enormous.

The Db2 product enables you to efficiently sample data for statistics collection, potentially improving the performance of the **RUNSTATS** utility by orders of magnitude, while maintaining a high degree of accuracy.

Two methods of sampling are available: row-level sampling and page-level sampling. For a description of these sampling methods, see “Data sampling in queries”.

Performance of page-level sampling is excellent, because only one I/O operation is required for each selected page. With row-level sampling, I/O costs are not reduced, because every table page is retrieved in a full table or index scan. However, row-level sampling provides significant performance improvements, even if the amount of I/O is not reduced, because collecting statistics is processor-intensive.

Row-level table sampling provides a better sample than page-level table sampling in situations where the data values are highly clustered. Compared to page-level table sampling, the row-level table sample set will likely be a better reflection of the data, because it will include *P* percent of the rows from each data page. With page-level table sampling, all the rows of *P* percent of the pages will be in the sample set. If the rows are distributed randomly over the table, the accuracy of row-sampled statistics will be similar to the accuracy of page-sampled statistics.

Each table sample is randomly generated across repeated invocations of the **RUNSTATS** command, unless the **REPEATABLE** parameter is used, in which case the previous table sample is regenerated. This option can be useful in cases where consistent statistics are required for tables whose data remains constant.

REPEATABLE does not apply to the index sampling (**INDEXSAMPLE**) - there is no similar functionality.

In IBM Data Studio Version 3.1 or later, you can use the task assistant for collecting statistics. Task assistants can guide you through the process of setting options, reviewing the automatically generated commands to perform the task, and running these commands. For more details, see Administering databases with task assistants.

Sub-element statistics:

If you specify LIKE predicates using the % wildcard character in any position other than at the end of the pattern, you should collect basic information about the sub-element structure.

As well as the wildcard LIKE predicate (for example, `SELECT...FROM DOCUMENTS WHERE KEYWORDS LIKE '%simulation%'`), the columns and the query must fit certain criteria to benefit from sub-element statistics.

Table columns should contain sub-fields or sub-elements separated by blanks. For example, a four-row table DOCUMENTS contains a KEYWORDS column with lists of relevant keywords for text retrieval purposes. The values in KEYWORDS are:

```
'database simulation analytical business intelligence'
'simulation model fruit fly reproduction temperature'
'forestry spruce soil erosion rainfall'
'forest temperature soil precipitation fire'
```

In this example, each column value consists of five sub-elements, each of which is a word (the keyword), separated from the others by one blank.

The query should reference these columns in WHERE clauses.

The optimizer always estimates how many rows match each predicate. For these wildcard LIKE predicates, the optimizer assumes that the column being matched contains a series of elements concatenated together, and it estimates the length of each element based on the length of the string, excluding leading and trailing % characters. If you collect sub-element statistics, the optimizer will have information about the length of each sub-element and the delimiter. It can use this additional information to more accurately estimate how many rows will match the predicate.

To collect sub-element statistics, execute the **RUNSTATS** command with the **LIKE STATISTICS** parameter.

Runstats statistics about sub-elements:

The **runstats** utility attempts to collect statistics for columns of type BINARY, VARBINARY, CHAR, and VARCHAR with a code page attribute of single-byte character set (SBCS), FOR BIT DATA, or UTF-8 when you specify the **LIKE STATISTICS** clause.

The **runstats** utility might decide not to collect the statistics if it determines that such statistics are not appropriate after analyzing column values.

SUB_COUNT
The average number of sub-elements.

SUB_DELIM_LENGTH
The average length of each delimiter separating the sub-elements. A delimiter, in this context, is one or more consecutive blank characters.

To examine the values of the sub-element statistics, query the SYSCAT.COLUMNS catalog view. For example:

```
select substr(colname, 1, 16), sub_count, sub_delim_length
  from syscat.columns where tabname = 'DOCUMENTS'
```

The **RUNSTATS** utility might take longer to complete if you use the **LIKE STATISTICS** clause. If you are considering this option, assess the improvements in query performance against this additional overhead.

General rules for updating catalog statistics manually:

When you update catalog statistics, the most important general rule is to ensure that valid values, ranges, and formats of the various statistics are stored in the views for those statistics.

It is also important to preserve the consistency of relationships among various statistics. For example, COLCARD in SYSSTAT.COLUMNS must be less than CARD in SYSSTAT.TABLES (the number of distinct values in a column cannot be greater than the number of rows in a table). Suppose that you want to reduce COLCARD from 100 to 25, and CARD from 200 to 50. If you update SYSSTAT.TABLES first, an error is returned, because CARD would be less than COLCARD.

In some cases, however, a conflict is difficult to detect, and an error might not be returned, especially if the impacted statistics are stored in different catalog tables.

Before updating catalog statistics, ensure (at a minimum) that:

- Numeric statistics are either -1 or greater than or equal to zero.
- Numeric statistics representing percentages (for example, CLUSTERRATIO in SYSSTAT.INDEXES) are between 0 and 100.

When a table is created, catalog statistics are set to -1 to indicate that the table has no statistics. Until statistics are collected, the Db2 server uses default values for SQL or XQuery statement compilation and optimization. Updating the table or index statistics might fail if the new values are inconsistent with the default values. Therefore, it is recommended that you use the runstats utility after creating a table, and before attempting to update statistics for the table or its indexes.

Note:

1. For row types, the table-level statistics NPAGES, FPAGES, and OVERFLOW are not updatable for a subtable.
2. Partition-level table and index statistics are not updatable.

Rules for updating column statistics manually:

There are certain guidelines that you should follow when updating statistics in the SYSSTAT.COLUMNS catalog view.

- When manually updating HIGH2KEY or LOW2KEY values, ensure that:
  - The values are valid for the data type of the corresponding user column.
  - The values are not NULL.
  - The length of the values must be the smaller of 33 or the maximum length of the target column data type, not including additional quotation marks, which can increase the length of the string to 68. This means that only the first 33 characters of the value in the corresponding user column will be considered in determining the HIGH2KEY or LOW2KEY values.
  - The values are stored in such a way that they can be used with the SET clause of an UPDATE statement, as well as for cost calculations. For character strings, this means that single quotation marks are added to the beginning and at the end of the string, and an extra quotation mark is added for every quotation mark that is already in the string. Examples of user column values and their corresponding values in HIGH2KEY or LOW2KEY are provided in Table 80.

Table 80. HIGH2KEY and LOW2KEY values by data type

Data type in user column	User data	Corresponding HIGH2KEY or LOW2KEY value
INTEGER	-12	-12
CHAR	abc	'abc'
CHAR	ab'c	'ab''c'

  - HIGH2KEY is greater than LOW2KEY whenever there are more than three distinct values in the corresponding column. If there are three or less distinct values in the column, HIGH2KEY can be equal to LOW2KEY.
- The cardinality of a column (COLCARD in SYSSTAT.COLUMNS) cannot be greater than the cardinality of its corresponding table or statistical view (CARD in SYSSTAT.TABLES).
- The number of null values in a column (NUMNULLS in SYSSTAT.COLUMNS) cannot be greater than the cardinality of its corresponding table or statistical view (CARD in SYSSTAT.TABLES).
- Statistics are not supported for columns that are defined with LONG or large object (LOB) data types.
- If you insert hex values in UTF-16 encoding but the database is created in UTF-8 encoding, the characters that are stored in the catalog table are invalid. The solution is to insert the hex values in UTF-8 encoding or in the UPDATE statement specify the values with GX, as you can see in the following example:

```
UPDATE SYSCAT.COLUMNS SET HIGH2KEY = GX'D48195A4868183A3',LOW2KEY = GX'048195A4868183A3'
```

Rules for updating table and nickname statistics manually:

There are certain guidelines that you should follow when updating statistics in the SYSSTAT.TABLES catalog view.

- The only statistical values that you can update in SYSSTAT.TABLES are CARD, FPAGES, NPAGES, AVGCOMPRESSEDROWSIZE, AVGROWCOMPRESSIONRATIO, PCTROWSCOMPRESSED, OVERFLOW and, for multidimensional clustering (MDC) tables, ACTIVE_BLOCKS.

- The value of the CARD statistic must meet the following criteria:
  - It must be greater than or equal to all COLCARD statistic values for the corresponding table in SYSSTAT.COLUMNS.
  - It must be greater than the value of the NPAGES statistic.
  - It must not be less than or equal to any “fetch” valuein the PAGE_FETCH_PAIRS column of any index (assuming that the CARD statistic is relevant to the index).
- The value of the FPAGES statistic must be greater than the value of the NPAGES statistic.
- The value of the NPAGES statistic must be less than or equal to any “fetch” value in the PAGE_FETCH_PAIRS column of any index (assuming that the NPAGES statistic is relevant to the index).
- Valid values for the AVGCOMPRESSEDROWSIZE statistic are -1 or between 0 and the value of the AVGROWSIZE statistic.
- Valid values for the AVGROWCOMPRESSIONRATIO statistic are -1 or greater than 1.
- Valid values for the PCTROWSCOMPRESSED statistic are -1 or 0 - 100, inclusive.

In a federated database system, use caution when manually updating statistics for a nickname over a remote view. Statistical information, such as the number of rows that a nickname returns, might not reflect the real cost of evaluating this remote view and therefore might mislead the Db2 optimizer. In certain cases, however, remote views can benefit from statistics updates; these include remote views that you define on a single base table with no column functions applied to the SELECT list. Complex views might require a complex tuning process in which you tune each query. Consider creating local views over nicknames so that the Db2 optimizer knows how to derive the cost of those views more accurately.

Detailed index statistics:

A **RUNSTATS** operation for indexes with the **DETAILED** parameter collects statistical information that allows the optimizer to estimate how many data page fetches are required, depending on the buffer pool size. This additional information helps the optimizer to better estimate the cost of accessing a table through an index.

Detailed statistics provide concise information about the number of physical I/Os that are required to access the data pages of a table if a complete index scan is performed under different buffer pool sizes. As the **RUNSTATS** utility scans the pages of an index, it models the different buffer sizes, and estimates how often a page fault occurs. For example, if only one buffer page is available, each new page that is referenced by the index results in a page fault. In the worst case, each row might reference a different page, resulting in at most the same number of I/Os as the number of rows in the indexed table. At the other extreme, when the buffer is big enough to hold the entire table (subject to the maximum buffer size), all table pages are read at once. As a result, the number of physical I/Os is a monotonic, nonincreasing function of the buffer size.

The statistical information also provides finer estimates of the degree of clustering of the table rows to the index order. The less clustering, the more I/Os are required to access table rows through the index. The optimizer considers both the buffer size and the degree of clustering when it estimates the cost of accessing a table through an index.

Collect detailed index statistics when:

- Queries reference columns that are not included in the index
- The table has multiple non-clustered indexes with varying degrees of clustering
- The degree of clustering among the key values is nonuniform
- Index values are updated in a nonuniform manner

It is difficult to identify these conditions without previous knowledge or without forcing an index scan under varying buffer sizes and then monitoring the resulting physical I/Os. Perhaps the least expensive way to determine whether any of these conditions exist is to collect and examine the detailed statistics for an index, and to retain them if the resulting PAGE_FETCH_PAIRS are nonlinear.

When you collect detailed index statistics, the **RUNSTATS** operation takes longer to complete and requires more memory and processing time. The **DETAILED** option (equivalent to the **SAMPLED DETAILED** parameter), for example, requires 2 MB of the statistics heap. Allocate an additional 488 4-KB pages to the **stat_heap_sz** database configuration parameter setting for this memory requirement. If the heap is too small, the **RUNSTATS** utility returns an error before it attempts to collect statistics.

CLUSTERFACTOR and PAGE_FETCH_PAIRS are not collected unless the table is of sufficient size (greater than about 25 pages). In this case, CLUSTERFACTOR will be a value between 0 and 1, and CLUSTERRATIO is -1 (not collected). If the table is relatively small, only CLUSTERRATIO, with a value between 0 and 100, is collected by the **RUNSTATS** utility; CLUSTERFACTOR and PAGE_FETCH_PAIRS are not collected. If the **DETAILED** clause is not specified, only CLUSTERRATIO is collected.

Collecting index statistics:

Collect index statistics to help the optimizer decide whether a specific index should be used to resolve a query.

About this task

The following example is based on a database named SALES that contains a CUSTOMERS table with indexes CUSTIDX1 and CUSTIDX2.

For privileges and authorities that are required to use the **RUNSTATS** utility, see the description of the **RUNSTATS** command.

Procedure

To collect detailed statistics for an index:

1. Connect to the SALES database.
2. Execute one of the following commands from the Db2 command line, depending on your requirements:
   - To collect detailed statistics on both CUSTIDX1 and CUSTIDX2:
     ```
     runstats on table sales.customers and detailed indexes all
     ```
   - To collect detailed statistics on both indexes, but with sampling instead of detailed calculations on each index entry:
     ```
     runstats on table sales.customers and sampled detailed indexes all
     ```

     The **SAMPLED DETAILED** parameter requires 2 MB of the statistics heap. Allocate an additional 488 4-KB pages to the **stat_heap_sz** database

configuration parameter setting for this memory requirement. If the heap is too small, the **RUNSTATS** utility returns an error before it attempts to collect statistics.

- To collect detailed statistics on sampled indexes, as well as distribution statistics for the table so that index and table statistics are consistent:

```
runstats on table sales.customers
  with distribution on key columns
    and sampled detailed indexes all
```

Rules for updating index statistics manually:

There are certain guidelines that you should follow when updating statistics in the SYSSTAT.INDEXES catalog view.

- The following rules apply to PAGE_FETCH_PAIRS:
  - Individual values in the PAGE_FETCH_PAIRS statistic must not be longer than 10 digits and must be less than the maximum integer value (2 147 483 647).
  - Individual values in the PAGE_FETCH_PAIRS statistic must be separated by a blank character delimiter.
  - There must always be a valid PAGE_FETCH_PAIRS statistic if CLUSTERFACTOR is greater than zero.
  - There must be exactly 11 pairs in a single PAGE_FETCH_PAIRS statistic.
  - Buffer size values in a PAGE_FETCH_PAIRS statistic (the first value in each pair) must appear in ascending order.
  - Any buffer size value in a PAGE_FETCH_PAIRS statistic cannot be greater than MIN(NPAGES, 524 287) for a 32-bit operating system, or MIN(NPAGES, 2 147 483 647) for a 64-bit operating system, where NPAGES (stored in SYSSTAT.TABLES) is the number of pages in the corresponding table.
  - Page fetch values in a PAGE_FETCH_PAIRS statistic (the second value in each pair) must appear in descending order, with no individual value being less than NPAGES or greater than CARD for the corresponding table.
  - If the buffer size value in two consecutive pairs is identical, the page fetch value in both of the pairs must also be identical.

  An example of a valid PAGE_FETCH_PAIRS statistic is:

```
PAGE_FETCH_PAIRS =
  '100 380 120 360 140 340 160 330 180 320 200 310 220 305 240 300
   260 300 280 300 300 300'
```

  where

```
NPAGES = 300
CARD = 10000
CLUSTERRATIO = -1
CLUSTERFACTOR = 0.9
```

- The following rules apply to CLUSTERRATIO and CLUSTERFACTOR:
  - Valid values for CLUSTERRATIO are -1 or between 0 and 100.
  - Valid values for CLUSTERFACTOR are -1 or between 0 and 1.
  - At least one of the CLUSTERRATIO and CLUSTERFACTOR values must be -1 at all times.
  - If CLUSTERFACTOR is a positive value, it must be accompanied by a valid PAGE_FETCH_PAIRS value.

- For relational indexes, the following rules apply to FIRSTKEYCARD, FIRST2KEYCARD, FIRST3KEYCARD, FIRST4KEYCARD, FULLKEYCARD, and INDCARD:
  - For a single-column index, FIRSTKEYCARD must be equal to FULLKEYCARD.
  - FIRSTKEYCARD must be equal to SYSSTAT.COLUMNS.COLCARD for the corresponding column.
  - If any of these index statistics are not relevant, set them to -1. For example, if you have an index with only three columns, set FIRST4KEYCARD to -1.
  - For multiple column indexes, if all of the statistics are relevant, the relationship among them must be as follows:

    ```
    FIRSTKEYCARD <= FIRST2KEYCARD <= FIRST3KEYCARD <= FIRST4KEYCARD
      <= FULLKEYCARD <= INDCARD == CARD
    ```
- For indexes over XML data, the relationship among FIRSTKEYCARD, FIRST2KEYCARD, FIRST3KEYCARD, FIRST4KEYCARD, FULLKEYCARD, and INDCARD must be as follows:

  ```
  FIRSTKEYCARD <= FIRST2KEYCARD <= FIRST3KEYCARD <= FIRST4KEYCARD
    <= FULLKEYCARD <= INDCARD
  ```
- The following rules apply to SEQUENTIAL_PAGES and DENSITY:
  - Valid values for SEQUENTIAL_PAGES are -1 or between 0 and NLEAF.
  - Valid values for DENSITY are -1 or between 0 and 100.

Distribution statistics:

You can collect two kinds of data distribution statistics: frequent-value statistics and quantile statistics.

- *Frequent-value statistics* provide information about a column and the data value with the highest number of duplicates, the value with the second highest number of duplicates, and so on, to the level that is specified by the value of the **num_freqvalues** database configuration parameter. To disable the collection of frequent-value statistics, set **num_freqvalues** to 0. You can also use the **NUM_FREQVALUES** clause on the **RUNSTATS** command for a specific table, statistical view, or column.
- *Quantile statistics* provide information about how data values are distributed in relation to other values. Called *K*-quantiles, these statistics represent the value *V* at or less than which at least *K* values lie. You can compute a *K*-quantile by sorting the values in ascending order. The *K*-quantile value is the value in the *K*th position from the low end of the range.

  To specify the number of "sections" (quantiles) into which the column data values should be grouped, set the **num_quantiles** database configuration parameter to a value between **2** and **32 767**. The default value is **20**, which ensures a maximum optimizer estimation error of plus or minus 2.5% for any equality, less-than, or greater-than predicate, and a maximum error of plus or minus 5% for any BETWEEN predicate. To disable the collection of quantile statistics, set **num_quantiles** to **0** or **1**.

  You can set **num_quantiles** for a specific table, statistical view, or column.

Note: The **RUNSTATS** utility consumes more processing resources and memory (specified by the **stat_heap_sz** database configuration parameter) if larger **num_freqvalues** and **num_quantiles** values are used.

When to collect distribution statistics

To decide whether distribution statistics for a table or statistical view would be helpful, first determine:

- Whether the queries in an application use host variables.

  Distribution statistics are most useful for dynamic and static queries that do not use host variables. The optimizer makes limited use of distribution statistics when assessing queries that contain host variables.

- Whether the data in columns is uniformly distributed.

  Create distribution statistics if at least one column in the table has a highly "nonuniform" distribution of data, and the column appears frequently in equality or range predicates; that is, in clauses such as the following:

  ```
  where c1 = key;
  where c1 in (key1, key2, key3);
  where (c1 = key1) or (c1 = key2) or (c1 = key3);
  where c1 <= key;
  where c1 between key1 and key2;
  ```

Two types of nonuniform data distribution can occur, and possibly together.

- Data might be highly clustered instead of being evenly spread out between the highest and lowest data value. Consider the following column, in which the data is clustered in the range (5,10):

  ```
    0.0
    5.1
    6.3
    7.1
    8.2
    8.4
    8.5
    9.1
   93.6
  100.0
  ```

  Quantile statistics help the optimizer to deal with this kind of data distribution.

  Queries can help you to determine whether column data is not uniformly distributed. For example:

  ```
  select c1, count(*) as occurrences
    from t1
  group by c1
  order by occurrences desc
  ```

- Duplicate data values might often occur. Consider a column in which the data is distributed with the following frequencies:

Table 81. Frequency of data values in a column

Data Value	Frequency
20	5
30	10
40	10
50	25
60	25
70	20
80	5

Both frequent-value and quantile statistics help the optimizer to deal with numerous duplicate values.

When to collect index statistics only

You might consider collecting statistics that are based only on index data in the following situations:

- A new index was created since the **RUNSTATS** utility was run, and you do not want to collect statistics again on the table data.
- There were many changes to the data that affect the first column of an index.

  Note: Quantile statistics are not collected for the first column of an index with random ordering.

What level of statistical precision to specify

Use the **num_quantiles** and **num_freqvalues** database configuration parameters to specify the precision with which distribution statistics are stored. You can also specify the precision with corresponding **RUNSTATS** command options when you collect statistics for a table or for columns. The higher you set these values, the greater the precision that the **RUNSTATS** utility uses when it creates and updates distribution statistics. However, greater precision requires more resources, both during the **RUNSTATS** operation itself, and for storing more data in the catalog tables.

For most databases, specify between 10 and 100 as the value of the **num_freqvalues** database configuration parameter. Ideally, frequent-value statistics should be created in such a way that the frequencies of the remaining values are either approximately equal to one another or negligible when compared to the frequencies of the most frequent values. The database manager might collect fewer than this number, because these statistics are collected only for data values that occur more than once. If you need to collect only quantile statistics, set the value of **num_freqvalues** to zero.

To specify the number of quantiles, set the **num_quantiles** database configuration parameter to a value between 20 and 50.

- First determine the maximum acceptable error when estimating the number of rows for any range query, as a percentage *P*.
- The number of quantiles should be approximately 100/*P* for BETWEEN predicates, and 50/*P* for any other type of range predicate (<, <=, >, or >=).

For example, 25 quantiles should result in a maximum estimate error of 4% for BETWEEN predicates and 2% for ">" predicates. In general, specify at least 10 quantiles. More than 50 quantiles should be necessary only for extremely nonuniform data. If you need only frequent-value statistics, set **num_quantiles** to 0. If you set this parameter to 1, because the entire range of values fits within one quantile, no quantile statistics are collected.

Optimizer use of distribution statistics:

The optimizer uses distribution statistics for better estimates of the cost of different query access plans.

Unless it has additional information about the distribution of values between the low and high values, the optimizer assumes that data values are evenly

distributed. If data values differ widely from each other, are clustered in some parts of the range, or contain many duplicate values, the optimizer will choose a less than optimal access plan.

Consider the following example: To select the least expensive access plan, the optimizer needs to estimate the number of rows with a column value that satisfies an equality or range predicate. The more accurate the estimate, the greater the likelihood that the optimizer will choose the optimal access plan. For the following query:

```
select c1, c2
  from table1
  where c1 = 'NEW YORK'
  and c2 <= 10
```

Assume that there is an index on both columns C1 and C2. One possible access plan is to use the index on C1 to retrieve all rows with `C1 = 'NEW YORK'`, and then to check whether `C2 <= 10` for each retrieved row. An alternate plan is to use the index on C2 to retrieve all rows with `C2 <= 10`, and then to check whether `C1 = 'NEW YORK'` for each retrieved row. Because the primary cost of executing a query is usually the cost of retrieving the rows, the best plan is the one that requires the fewest retrievals. Choosing this plan means estimating the number of rows that satisfy each predicate.

When distribution statistics are not available, but the runstats utility has been used on a table or a statistical view, the only information that is available to the optimizer is the second-highest data value (HIGH2KEY), the second-lowest data value (LOW2KEY), the number of distinct values (COLCARD), and the number of rows (CARD) in a column. The number of rows that satisfy an equality or range predicate is estimated under the assumption that the data values in the column have equal frequencies and that the data values are evenly distributed between LOW2KEY and HIGH2KEY. Specifically, the number of rows that satisfy an equality predicate (`C1 = KEY`) is estimated as CARD/COLCARD, and the number of rows that satisfy a range predicate (`C1 BETWEEN KEY1 AND KEY2`) can be estimated with the following formula:

```
    KEY2 - KEY1
------------------  x CARD
HIGH2KEY - LOW2KEY
```

These estimates are accurate only when the true distribution of data values within a column is reasonably uniform. When distribution statistics are unavailable, and either the frequency of data values varies widely, or the data values are very unevenly distributed, the estimates can be off by orders of magnitude, and the optimizer might choose a suboptimal access plan.

When distribution statistics are available, the probability of such errors can be greatly reduced by using frequent-value statistics to estimate the number of rows that satisfy an equality predicate, and by using both frequent-value statistics and quantile statistics to estimate the number of rows that satisfy a range predicate.

Collecting distribution statistics for specific columns:

For efficient **RUNSTATS** operations and subsequent query-plan analysis, collect distribution statistics on only those columns that queries reference in WHERE, GROUP BY, and similar clauses. You can also collect cardinality statistics on combined groups of columns. The optimizer uses such information to detect column correlation when it estimates selectivity for queries that reference the columns in a group.

About this task

The following example is based on a database named SALES that contains a CUSTOMERS table with indexes CUSTIDX1 and CUSTIDX2.

For privileges and authorities that are required to use the **RUNSTATS** utility, see the description of the **RUNSTATS** command.

When you collect statistics for a table in a partitioned database environment, **RUNSTATS** operates only on the database partition from which the utility is executed. The results from this database partition are extrapolated to the other database partitions. If this database partition does not contain a required portion of the table, the request is sent to the first database partition in the database partition group that contains the required data.

Procedure

To collect statistics on specific columns:

1. Connect to the SALES database.
2. Execute one of the following commands from the Db2 command line, depending on your requirements:
   - To collect distribution statistics on columns ZIP and YTDTOTAL:
     ```
     runstats on table sales.customers
       with distribution on columns (zip, ytdtotal)
     ```
   - To collect distribution statistics on the same columns, but with different distribution options:
     ```
     runstats on table sales.customers
       with distribution on columns (
         zip, ytdtotal num_freqvalues 50 num_quantiles 75)
     ```
   - To collect distribution statistics on the columns that are indexed in CUSTIDX1 and CUSTIDX2:
     ```
     runstats on table sales.customer
       on key columns
     ```
   - To collect statistics for columns ZIP and YTDTOTAL and a column group that includes REGION and TERRITORY:
     ```
     runstats on table sales.customers
       on columns (zip, (region, territory), ytdtotal)
     ```
   - Suppose that statistics for non-XML columns were collected previously using the **LOAD** command with the **STATISTICS** parameter. To collect statistics for the XML column MISCINFO:
     ```
     runstats on table sales.customers
       on columns (miscinfo)
     ```
   - To collect statistics for the non-XML columns only:
     ```
     runstats on table sales.customers
       excluding xml columns
     ```

     The **EXCLUDING XML COLUMNS** clause takes precedence over all other clauses that specify XML columns.
   - For Db2 V9.7 Fix Pack 1 and later releases, the following command collects distribution statistics using a maximum of 50 quantiles for the XML column MISCINFO. A default of 20 quantiles is used for all other columns in the table:

```
runstats on table sales.customers
   with distribution on columns ( miscinfo num_quantiles 50 )
   default num_quantiles 20
```

Note: The following are required for distribution statistics to be collected on the XML column MISCINFO:

- Both table and distribution statistics must be collected.
- An index over XML data must be defined on the column, and the data type specified for the index must be VARCHAR, DOUBLE, TIMESTAMP, or DATE.

Extended examples of the use of distribution statistics:

Distribution statistics provide information about the frequency and distribution of table data that helps the optimizer build query access plans when the data is not evenly distributed and there are many duplicates.

The following examples will help you to understand how the optimizer might use distribution statistics.

Example with frequent-value statistics

Consider a query that contains an equality predicate of the form `C1 = KEY`. If frequent-value statistics are available, the optimizer can use those statistics to choose an appropriate access plan, as follows:

- If `KEY` is one of the *N* most frequent values, the optimizer uses the frequency of `KEY` that is stored in the catalog.
- If `KEY` is not one of the *N* most frequent values, the optimizer estimates the number of rows that satisfy the predicate under the assumption that the (COLCARD - *N*) non-frequent values have a uniform distribution. That is, the number of rows is estimated by the following formula (1):

```
CARD - NUM_FREQ_ROWS
--------------------
    COLCARD - N
```

where CARD is the number of rows in the table, COLCARD is the cardinality of the column, and NUM_FREQ_ROWS is the total number of rows with a value equal to one of the *N* most frequent values.

For example, consider a column C1 whose data values exhibit the following frequencies:

Data Value	Frequency
1	2
2	3
3	40
4	4
5	1

The number of rows in the table is 50 and the column cardinality is 5. Exactly 40 rows satisfy the predicate `C1 = 3`. If it is assumed that the data is evenly distributed, the optimizer estimates the number of rows that satisfy the predicate

as 50/5 = 10, with an error of -75%. But if frequent-value statistics based on only the most frequent value (that is, $N = 1$) are available, the number of rows is estimated as 40, with no error.

Consider another example in which two rows satisfy the predicate `C1 = 1`. Without frequent-value statistics, the number of rows that satisfy the predicate is estimated as 10, an error of 400%:

```
estimated rows - actual rows
----------------------------  X 100
         actual rows

10 - 2
------  X 100 = 400%
   2
```

Using frequent-value statistics ($N = 1$), the optimizer estimates the number of rows containing this value using the formula (1) given previously as:

```
(50 - 40)
--------- = 3
 (5 - 1)
```

and the error is reduced by an order of magnitude:

```
 3 - 2
 ----- = 50%
   2
```

Example with quantile statistics

The following discussion of quantile statistics uses the term "K-quantile". The *K-quantile* for a column is the smallest data value, *V*, such that at least *K* rows have data values that are less than or equal to *V*. To compute a K-quantile, sort the column values in ascending order; the *K*-quantile is the data value in the *K*th row of the sorted column.

If quantile statistics are available, the optimizer can better estimate the number of rows that satisfy a range predicate, as illustrated by the following examples. Consider a column C1 that contains the following values:

```
  0.0
  5.1
  6.3
  7.1
  8.2
  8.4
  8.5
  9.1
 93.6
100.0
```

Suppose that *K*-quantiles are available for $K = 1, 4, 7$, and 10, as follows:

K	***K*-quantile**
1	0.0
4	7.1
7	8.5
10	100.0

- Exactly seven rows satisfy the predicate `C <= 8.5`. Assuming a uniform data distribution, the following formula (2):

```
   KEY2 - KEY1
------------------  X CARD
HIGH2KEY - LOW2KEY
```

  with LOW2KEY in place of KEY1, estimates the number of rows that satisfy the predicate as:

```
 8.5 - 5.1
----------  X 10 ≈ 0
93.6 - 5.1
```

  where ≈ means "approximately equal to". The error in this estimate is approximately -100%.

  If quantile statistics are available, the optimizer estimates the number of rows that satisfy this predicate by the value of *K* that corresponds to 8.5 (the highest value in one of the quantiles), which is 7. In this case, the error is reduced to 0.

- Exactly eight rows satisfy the predicate `C <= 10`. If the optimizer assumes a uniform data distribution and uses formula (2), the number of rows that satisfy the predicate is estimated as 1, an error of -87.5%.

  Unlike the previous example, the value 10 is not one of the stored *K*-quantiles. However, the optimizer can use quantiles to estimate the number of rows that satisfy the predicate as `r_1 + r_2`, where `r_1` is the number of rows satisfying the predicate `C <= 8.5` and `r_2` is the number of rows satisfying the predicate `C > 8.5 AND C <= 10`. As in the previous example, `r_1 = 7`. To estimate `r_2`, the optimizer uses linear interpolation:

```
       10 - 8.5
r_2 ≈ ----------  X (number of rows with value > 8.5 and <= 100.0)
       100 - 8.5

       10 - 8.5
r_2 ≈ ----------  X (10 - 7)
       100 - 8.5

       1.5
r_2 ≈ -----   X (3)
       91.5

r_2 ≈ 0
```

  The final estimate is r_1 + r_2 ≈ 7, and the error is only -12.5%.

Quantiles improve the accuracy of the estimates in these examples because the real data values are "clustered" in a range from 5 to 10, but the standard estimation formulas assume that the data values are distributed evenly between 0 and 100.

The use of quantiles also improves accuracy when there are significant differences in the frequencies of different data values. Consider a column having data values with the following frequencies:

Data Value	Frequency
20	5
30	5
40	15
50	50
60	15

Data Value	Frequency
70	5
80	5

Suppose that *K*-quantiles are available for *K* = 5, 25, 75, 95, and 100:

K	*K*-quantile
5	20
25	40
75	50
95	70
100	80

Suppose also that frequent-value statistics are available, based on the three most frequent values.

Exactly 10 rows satisfy the predicate `C BETWEEN 20 AND 30`. Assuming a uniform data distribution and using formula (2), the number of rows that satisfy the predicate is estimated as:

```
30 - 20
-------  X 100 = 25
70 - 30
```

an error of 150%.

Using frequent-value statistics and quantile statistics, the number of rows that satisfy the predicate is estimated as `r_1 + r_2`, where `r_1` is the number of rows that satisfy the predicate `(C = 20)` and `r_2` is the number of rows that satisfy the predicate `C > 20 AND C <= 30`. Using formula (1), `r_1` is estimated as:

```
100 - 80
-------- = 5
  7 - 3
```

Using linear interpolation, `r_2` is estimated as:

```
  30 - 20
  -------  X (number of rows with a value > 20 and <= 40)
  40 - 20

  30 - 20
= -------  X (25 - 5)
  40 - 20

= 10
```

This yields a final estimate of 15 and reduces the error by a factor of three.

Rules for updating distribution statistics manually:

There are certain guidelines that you should follow when updating statistics in the SYSSTAT.COLDIST catalog view.

- Frequent-value statistics:
  - VALCOUNT values must be unchanging or decreasing with increasing values of SEQNO.

  - The number of COLVALUE values must be less than or equal to the number of distinct values in the column, which is stored in SYSSTAT.COLUMNS.COLCARD.
  - The sum of the values in VALCOUNT must be less than or equal to the number of rows in the column, which is stored in SYSSTAT.TABLES.CARD.
  - In most cases, COLVALUE values should lie between the second-highest and the second-lowest data values for the column, which are stored in HIGH2KEY and LOW2KEY in SYSSTAT.COLUMNS, respectively. There can be one frequent value that is greater than HIGH2KEY and one frequent value that is less than LOW2KEY.
- Quantile statistics:
  - COLVALUE values must be unchanging or decreasing with increasing values of SEQNO.
  - VALCOUNT values must be increasing with increasing values of SEQNO.
  - The largest COLVALUE value must have a corresponding entry in VALCOUNT that is equal to the number of rows in the column.
  - In most cases, COLVALUE values should lie between the second-highest and the second-lowest data values for the column, which are stored in HIGH2KEY and LOW2KEY in SYSSTAT.COLUMNS, respectively.

Suppose that distribution statistics are available for column C1 with *R* rows, and that you want to modify the statistics to correspond with a column that has the same relative proportions of data values, but with (*F* x *R*) rows. To scale up the frequent-value or quantile statistics by a factor of *F*, multiply each VALCOUNT entry by *F*.

Statistics for user-defined functions:

To create statistical information for user-defined functions (UDFs), edit the SYSSTAT.ROUTINES catalog view.

The runstats utility does not collect statistics for UDFs. If UDF statistics are available, the optimizer can use them when it estimates costs for various access plans. If statistics are not available, the optimizer uses default values that assume a simple UDF.

Table 82 lists the catalog view columns for which you can provide estimates to improve performance. Note that only column values in SYSSTAT.ROUTINES (not SYSCAT.ROUTINES) can be modified by users.

Table 82. Function Statistics (SYSCAT.ROUTINES and SYSSTAT.ROUTINES)

Statistic	Description
IOS_PER_INVOC	Estimated number of read or write requests executed each time a function is called
INSTS_PER_INVOC	Estimated number of machine instructions executed each time a function is called
IOS_PER_ARGBYTE	Estimated number of read or write requests executed per input argument byte
INSTS_PER_ARGBYTE	Estimated number of machine instructions executed per input argument byte
PERCENT_ARGBYTES	Estimated average percent of input argument bytes that a function will actually process

Table 82. Function Statistics (SYSCAT.ROUTINES and SYSSTAT.ROUTINES) (continued)

Statistic	Description
INITIAL_IOS	Estimated number of read or write requests executed the first or last time a function is invoked
INITIAL_INSTS	Estimated number of machine instructions executed the first or last time a function is invoked
CARDINALITY	Estimated number of rows generated by a table function

For example, consider EU_SHOE, a UDF that converts an American shoe size to the equivalent European shoe size. For this UDF, you might set the values of statistic columns in SYSSTAT.ROUTINES as follows:

- INSTS_PER_INVOC. Set to the estimated number of machine instructions required to:
  - Call EU_SHOE
  - Initialize the output string
  - Return the result
- INSTS_PER_ARGBYTE. Set to the estimated number of machine instructions required to convert the input string into a European shoe size
- PERCENT_ARGBYTES. Set to 100, indicating that the entire input string is to be converted
- INITIAL_INSTS, IOS_PER_INVOC, IOS_PER_ARGBYTE, and INITIAL_IOS. Each set to 0, because this UDF only performs computations

PERCENT_ARGBYTES would be used by a function that does not always process the entire input string. For example, consider LOCATE, a UDF that accepts two arguments as input and returns the starting position of the first occurrence of the first argument within the second argument. Assume that the length of the first argument is small enough to be insignificant relative to the second argument and that, on average, 75% of the second argument is searched. Based on this information and the following assumptions, PERCENT_ARGBYTES should be set to 75:

- Half the time the first argument is not found, which results in searching the entire second argument
- The first argument is equally likely to appear anywhere within the second argument, which results in searching half of the second argument (on average) when the first argument is found

You can use INITIAL_INSTS or INITIAL_IOS to record the estimated number of machine instructions or read or write requests that are performed the first or last time that a function is invoked; this might represent the cost, for example, of setting up a scratchpad area.

To obtain information about I/Os and the instructions that are used by a UDF, use output provided by your programming language compiler or by monitoring tools that are available for your operating system.

Monitoring the progress of RUNSTATS operations:

You can use the **LIST UTILITIES** command or the **db2pd** command to monitor the progress of **RUNSTATS** operations on a database.

Procedure

Issue the **LIST UTILITIES** command and specify the **SHOW DETAIL** parameter:

```
list utilities show detail
```

or issue the **db2pd** command and specify the **-runstats** parameter:

```
db2pd -runstats
```

Results

The following is an example of the output for monitoring the performance of a **RUNSTATS** operation using the **LIST UTILITIES** command:

```
ID                                = 7
Type                              = RUNSTATS
Database Name                     = SAMPLE
Partition Number                  = 0
Description                       = YIWEIANG.EMPLOYEE
Start Time                        = 08/04/2011 12:39:35.155398
State                             = Executing
Invocation Type                   = User
Throttling:
   Priority                       = Unthrottled
```

The following is an example of the output for monitoring the performance of a **RUNSTATS** operation using the **db2pd** command:

```
db2pd -runstats

Table Runstats Information:

Retrieval Time: 08/13/2009 20:38:20
TbspaceID: 2        TableID: 4
Schema: SCHEMA    TableName: TABLE
Status: Completed     Access: Allow write
Sampling: No          Sampling Rate: -
Start Time: 08/13/2009 20:38:16   End Time: 08/13/2009 20:38:17
Total Duration: 00:00:01
Cur Count: 0                      Max Count: 0

Index Runstats Information:

Retrieval Time: 08/13/2009 20:38:20
TbspaceID: 2        TableID: 4
Schema: SCHEMA    TableName: TABLE
Status: Completed     Access: Allow write
Start Time: 08/13/2009 20:38:17   End Time: 08/13/2009 20:38:18
Total Duration: 00:00:01
Prev Index Duration [1]: 00:00:01
Prev Index Duration [2]:  -
Prev Index Duration [3]:  -
Cur Index Start: 08/13/2009 20:38:18
Cur Index: 2            Max Index: 2            Index ID: 2
Cur Count: 0                      Max Count: 0
```

Catalog statistics for modeling and what-if planning:

You can observe the effect on database performance of changes to certain statistical information in the system catalog for planning purposes.

The ability to update selected system catalog statistics enables you to:

- Model query performance on a development system using production system statistics
- Perform "what-if" query performance analysis

Do not manually update statistics on a production system. Otherwise, the optimizer might not choose the best access plan for production queries that contain dynamic SQL or XQuery statements.

To modify statistics for tables and indexes and their components, you must have explicit DBADM authority for the database. Users holding DATAACCESS authority can execute UPDATE statements against views that are defined in the SYSSTAT schema to change values in these statistical columns.

Users without DATAACCESS authority can see only rows that contain statistics for objects on which they have CONTROL privilege. If you do not have DATAACCESS authority, you can change statistics for individual database objects if you hold the following privileges on each object:

- Explicit CONTROL privilege on tables. You can also update statistics for columns and indexes on these tables.
- Explicit CONTROL privilege on nicknames in a federated database system. You can also update statistics for columns and indexes on these nicknames. Note that these updates only affect local metadata (datasource table statistics are not changed), and only affect the global access strategy that is generated by the Db2 optimizer.
- Ownership of user-defined functions (UDFs)

The following code is an example of updating statistics for the EMPLOYEE table:

```
update sysstat.tables
  set
    card = 10000,
    npages = 1000,
    fpages = 1000,
    overflow = 2
  where tabschema = 'MELNYK'
    and tabname = 'EMPLOYEE'
```

Care must be taken when manually updating catalog statistics. Arbitrary changes can seriously alter the performance of subsequent queries. You can use any of the following methods to return the statistics on your development system to a consistent state:

- Roll back the unit of work in which your manual changes were made (assuming that the unit of work has not yet been committed).
- Use the runstats utility to refresh the catalog statistics.
- Update the catalog statistics to specify that statistics have not been collected; for example, setting the NPAGES column value to -1 indicates that this statistic has not been collected.
- Undo the changes that you made. This method is possible only if you used the **db2look** command to capture the statistics before you made any changes.

If it determines that some value or combination of values is not valid, the optimizer will use default values and return a warning. This is quite rare, however, because most validation is performed when the statistics are updated.

Statistics for modeling production databases:

Sometimes you might want your development system to contain a subset of the data in your production system. However, access plans that are selected on development systems are not necessarily the same as those that would be selected on the production system.

In some cases, it is necessary that the catalog statistics and the configuration of the development system be updated to match those of the production system.

The **db2look** command in mimic mode (specifying the -m option) can be used to generate the data manipulation language (DML) statements that are required to make the catalog statistics of the development and production databases match.

After running the UPDATE statements that are produced by **db2look** against the development system, that system can be used to validate the access plans that are being generated on the production system. Because the optimizer uses the configuration of table spaces to estimate I/O costs, table spaces on the development system must be of the same type (SMS or DMS) and have the same number of containers as do those on the production system. The test system may have less physical memory than the production system. Setting memory related configuration parameters on the test system same as values of those on the production system might not be feasible. You can use the **db2fopt** command to assign values to be used by the optimizer during statement compilation. For example, if the production system is running with **sortheap**=20000 and the test system can only run with **sortheap**=5000, you can use **db2fopt** on the test system to set **opt_sortheap** to 20000. **opt_sortheap** instead of **sortheap**, will be used by the query optimizer during statement compilation when evaluating access plans.

Avoiding manual updates to the catalog statistics:

The Db2 data server supports manually updating catalog statistics by issuing UPDATE statements against views in the SYSSTAT schema.

This feature can be useful when mimicking a production database on a test system in order to examine query access plans. The **db2look** utility is very helpful for capturing the DDL and UPDATE statements against views in the SYSSTAT schema for playback on another system.

Avoid influencing the query optimizer by manually providing incorrect statistics to force a particular query access plan. Although this practice might result in improved performance for some queries, it can result in performance degradation for others. Consider other tuning options (such as using optimization guidelines and profiles) before resorting to this approach. If this approach does become necessary, be sure to record the original statistics in case they need to be restored.

Minimizing RUNSTATS impact

There are several approaches available to improve **RUNSTATS** performance.

To minimize the performance impact of this utility:

- Limit the columns for which statistics are collected by using the COLUMNS clause. Many columns are never referenced by predicates in the query workload, so they do not require statistics.
- Limit the columns for which distribution statistics are collected if the data tends to be uniformly distributed. Collecting distribution statistics requires more CPU and memory than collecting basic column statistics. However, determining whether the values for a column are uniformly distributed requires either having existing statistics or querying the data. This approach also assumes that the data remains uniformly distributed as the table is modified.
- Limit the number of pages and rows processed by using page- or row-level table sampling (by specifying the **TABLESAMPLE SYSTEM** or **TABLESAMPLE BERNOULLI** clause) and by using page- or row-level index sampling (by specifying **INDEXSAMPLE SYSTEM** or **INDEXSAMPLE BERNOULLI** clause). Start with a 10% page-level sample, by specifying `TABLESAMPLE SYSTEM(10)` and `INDEXSAMPLE SYSTEM(10)`. Check the accuracy of the statistics and whether system performance has degraded due to changes in access plan. If it has degraded, try a 10% row-level sample instead, by specifying `TABLESAMPLE BERNOULLI(10)`. Likewise, experiment with the **INDEXSAMPLE** parameter to get the right rate for index sampling. If the accuracy of the statistics is insufficient, increase the sampling amount. When using **RUNSTATS** page- or row-level sampling, use the same sampling rate for tables that are joined. This is important to ensure that the join column statistics have the same level of accuracy.
- Collect index statistics during index creation by specifying the COLLECT STATISTICS option on the CREATE INDEX statement. This approach is faster than performing a separate **RUNSTATS** operation after the index is created. It also ensures that the new index has statistics generated immediately after creation, to allow the optimizer to accurately estimate the cost of using the index.
- Collect statistics when executing the **LOAD** command with the **REPLACE** option. This approach is faster than performing a separate **RUNSTATS** operation after the load operation completes. It also ensures that the table has the most current statistics immediately after the data is loaded, to allow the optimizer to accurately estimate the cost of using the table.

In a partitioned database environment, the **RUNSTATS** utility collects statistics from a single database partition. If the **RUNSTATS** command is issued on a database partition on which the table resides, statistics are collected there. If not, statistics are collected on the first database partition in the database partition group for the table. For consistent statistics, ensure that statistics for joined tables are collected from the same database partition.

Data compression and performance

You can use data compression to reduce the amount of data that must be read from or written to disk, thereby reducing I/O cost.

Generally speaking, the more repetitive patterns that exist in data rows, the better your compression rates will be. If your data does not contain repetitive strings, as might be the case if you have mainly numeric data, or include inline BLOB data, compression might not yield as much in the way of storage savings. In addition, when data does not compress well, you still incur the performance impact of the database manager attempting to compress data, only to find that the storage benefits are not worth retaining any compression dictionaries. To check on the possible storage savings that you can achieve with compression, use the ADMIN_GET_TAB_COMPRESS_INFO table function.

Two forms of data compression are available:

Row compression

There are two types of row compression available:

Classic row compression

Classic row compression involves replacing repeating patterns that span multiple column values within a row with shorter symbol strings. A sampling of rows are scanned to find instances of repetitive data. From this scan, a table-level compression dictionary is created. This dictionary is used to replace the repetitive data with shorter symbol strings.

Table-level compression dictionaries are static; after they are first created, they do not change unless you rebuild them during a classic table reorganization.

Adaptive row compression

Adaptive row compression involves the use of two compression approaches: classic row compression and *page-level compression*. Page-level compression involves replacing repeating patterns that span multiple column values *within a single page* of data with shorter symbol strings. As a page fills with data, page compression logic scans the page for repetitive data. From this scan, a page-level compression dictionary is created. This dictionary is used to replace the repetitive data with shorter symbol strings.

Page-level dictionaries are dynamic; they are rebuilt automatically if necessary.

Value compression

Value compression involves removing duplicate entries for a value, storing only one copy, and keeping track of the location of any references to the stored value.

Row compression is also used for temporary tables. Compressing temporary tables reduces the amount of temporary disk space that is required for large and complex queries, thus increasing query performance. Compression for temporary tables is enabled automatically under the Db2 Storage Optimization Feature. Each temporary table that is eligible for row compression requires an additional 2 - 3 MB of memory for the creation of its compression dictionary. This memory remains allocated until the compression dictionary has been created.

You can also compress index objects, which reduces storage costs. This kind of compression is especially useful for large online transaction processing (OLTP) and data warehouse environments, where it is common to have many very large indexes. In both of these cases, index compression can significantly improve performance in I/O-bound environments while causing little or no performance degradation in CPU-bound environments.

If you enable compression on a table with an XML column, the XML data that is stored in the XDA object is also compressed. A separate compression dictionary for the XML data is stored in the XDA object. If you use Db2 Version 9.7 or later to add XML columns to a table created using Db2 Version 9.5, these XML columns are compressed. However, XML columns added to a table using Db2 Version 9.5 are not compressed; in such tables, only the data object is compressed.

Reducing logging overhead to improve DML performance

The database manager maintains log files that record all database changes. There are two logging strategies: circular logging and archive logging.

- With *circular logging*, log files are reused (starting with the initial log file) when the available files have filled up. The overwritten log records are not recoverable.
- With *archive logging*, log files are archived when they fill up with log records. Log retention enables rollforward recovery, in which changes to the database (completed units of work or transactions) that are recorded in the log files can be reapplied during disaster recovery.

All changes to regular data and index pages are written to the log buffer before being written to disk by the logger process. SQL statement processing must wait for log data to be written to disk:

- On COMMIT
- Until the corresponding data pages are written to disk, because the Db2 server uses write-ahead logging, in which not all of the changed data and index pages need to be written to disk when a transaction completes with a COMMIT statement
- Until changes (mostly resulting from the execution of data definition language statements) are made to the metadata
- When the log buffer is full

The database manager writes log data to disk in this way to minimize processing delays. If many short transactions are processing concurrently, most of the delay is caused by COMMIT statements that must wait for log data to be written to disk. As a result, the logger process frequently writes small amounts of log data to disk. Additional delays are caused by log I/O. To balance application response time with logging delays, set the **mincommit** database configuration parameter to a value that is greater than 1. This setting might cause longer COMMIT delays for some applications, but more log data might be written in one operation.

Changes to large objects (LOBs) and LONG VARCHARs are tracked through shadow paging. LOB column changes are not logged unless you specify log retain and the LOB column has been defined without the NOT LOGGED clause on the CREATE TABLE statement. Changes to allocation pages for LONG or LOB data types are logged like regular data pages. Inline LOB values participate fully in update, insert, or delete logging, as though they were VARCHAR values.

Inline LOBs improve performance

Some applications make extensive use of large objects (LOBs). In many cases, these LOBs are not very large-at most, a few kilobytes in size. The performance of LOB data access can now be improved by placing such LOB data within the formatted rows on data pages instead of in the LOB storage object.

Such LOBs are known as *inline LOBs*. Previously, the processing of such LOBs could create bottlenecks for applications. Inline LOBs improve the performance of queries that access LOB data, because no additional I/O is required to fetch, insert, or update this data. Moreover, inline LOB data is eligible for row compression.

This feature is enabled through the INLINE LENGTH option on the CREATE TABLE statement or the ALTER TABLE statement. The INLINE LENGTH option applies to structured types, the XML type, or LOB columns. In the case of a LOB column, the inline length indicates the maximum byte size of a LOB value (including four bytes for overhead) that can be stored in a base table row.

This feature is also implicitly enabled for all LOB columns in new or existing tables (when LOB columns are added), and for all existing LOB columns on database upgrade. Every LOB column has reserved row space that is based on its defined maximum size. An implicit INLINE LENGTH value for each LOB column is defined automatically and stored as if it had been explicitly specified.

LOB values that cannot be stored inline are stored separately in the LOB storage object.

Note that when a table has columns with inline LOBs, fewer rows fit on a page, and the performance of queries that return only non-LOB data can be adversely affected. LOB inlining is helpful for workloads in which most of the statements include one or more LOB columns.

Although LOB data is not necessarily logged, inline LOBs are always logged and can therefore increase logging overhead.

Establishing a performance tuning strategy

The Design Advisor

The Db2 Design Advisor is a tool that can help you significantly improve your workload performance. The task of selecting which indexes, materialized query tables (MQTs), clustering dimensions, or database partitions to create for a complex workload can be daunting. The Design Advisor identifies all of the objects that are needed to improve the performance of your workload.

Given a set of SQL statements in a workload, the Design Advisor generates recommendations for:

- New indexes
- New clustering indexes
- New MQTs
- Conversion to multidimensional clustering (MDC) tables
- The redistribution of tables

The Design Advisor can implement some or all of these recommendations immediately, or you can schedule them to run at a later time.

Use the **db2advis** command to launch the Design Advisor utility.

The Design Advisor can help simplify the following tasks:

Planning for and setting up a new database

While designing your database, use the Design Advisor to generate design alternatives in a test environment for indexing, MQTs, MDC tables, or database partitioning.

In partitioned database environments, you can use the Design Advisor to:

- Determine an appropriate database partitioning strategy before loading data into a database
- Assist in upgrading from a single-partition database to a multi-partition database
- Assist in migrating from another database product to a multi-partition Db2 database

Workload performance tuning

After your database is set up, you can use the Design Advisor to:

- Improve the performance of a particular statement or workload
- Improve general database performance, using the performance of a sample workload as a gauge
- Improve the performance of the most frequently executed queries, as identified, for example, by the IBM InfoSphere Optim Performance Manager
- Determine how to optimize the performance of a new query
- Respond to IBM Data Studio Health Monitor recommendations regarding shared memory utility or sort heap problems with a sort-intensive workload
- Find objects that are not used in a workload

IBM InfoSphere Optim Query Workload Tuner provides tools for improving the performance of single SQL statements and the performance of groups of SQL statements, which are called query workloads. For more information about this product, see the product overview page at http://www.ibm.com/software/data/optim/query-workload-tuner-db2-luw/index.html. In Version 3.1.1 or later, you can also use the Workload Design Advisor to perform many operations that were available in the Db2 Design Advisor wizard. For more information see the documentation for the Workload Design Advisor at http://www.ibm.com/support/knowledgecenter/SS62YD_4.1.1/com.ibm.datatools.qrytune.workloadtunedb2luw.doc/topics/genrecsdsgn.html.

Design Advisor output

Design Advisor output is written to standard output by default, and saved in the ADVISE_* tables:

- The ADVISE_INSTANCE table is updated with one new row each time that the Design Advisor runs:
  - The START_TIME and END_TIME fields show the start and stop times for the utility.
  - The STATUS field contains a value of COMPLETED if the utility ended successfully.
  - The MODE field indicates whether the **-m** parameter was used on the **db2advis** command.
  - The COMPRESSION field indicates the type of compression that was used.
- The USE_TABLE column in the ADVISE_TABLE table contains a value of Y if MQT, MDC table, or database partitioning strategy recommendations were made.

  MQT recommendations can be found in the ADVISE_MQT table; MDC recommendations can be found in the ADVISE_TABLE table; and database partitioning strategy recommendations can be found in the ADVISE_PARTITION table. The RUN_ID column in these tables contains a value that corresponds to the START_TIME value of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.

  When MQT, MDC, or database partitioning recommendations are provided, the relevant ALTER TABLE stored procedure call is placed in the ALTER_COMMAND column of the ADVISE_TABLE table. The ALTER TABLE stored procedure call might not succeed due to restrictions on the table for the ALTOBJ stored procedure.

- The USE_INDEX column in the ADVISE_INDEX table contains a value of Y (index recommended or evaluated) or R (an existing clustering RID index was recommended to be unclustered) if index recommendations were made.
- The COLSTATS column in the ADVISE_MQT table contains column statistics for an MQT. These statistics are contained within an XML structure as follows:

```
<?xml version=\"1.0\" encoding=\"USASCII\"?>
<colstats>
    <column>
         <name>COLNAME1</name>
         <colcard>1000</colcard>
         <high2key>999</high2key>
         <low2key>2</low2key>
    </column>
    ....

    <column>
         <name>COLNAME100</name>
         <colcard>55000</colcard>
         <high2key>49999</high2key>
         <low2key>100</low2key>
    </column>
</colstats>
```

You can save Design Advisor recommendations to a file using the **-o** parameter on the **db2advis** command. The saved Design Advisor output consists of the following elements:

- CREATE statements associated with any new indexes, MQTs, MDC tables, or database partitioning strategies
- REFRESH statements for MQTs
- **RUNSTATS** commands for new objects

An example of this output is as follows:

```
--<?xml version="1.0"?>
--<design-advisor>
--<mqt>
--<identifier>
--<name>MQT612152202220000</name>
--<schema>ZILIO2 </schema>
--</identifier>
--<statementlist>3</statementlist>
--<benefit>1013562.481682</benefit>
--<overhead>1468328.200000</overhead>
--<diskspace>0.004906</diskspace>
--</mqt>
.....
--<index>
--<identifier>
--<name>IDX612152221400000</name>
--<schema>ZILIO2 </schema>
--</identifier>
--<table><identifier>
--<name>PART</name>
--<schema>SAMP </schema>
--</identifier></table>
--<statementlist>22</statementlist>
--<benefit>820160.000000</benefit>
--<overhead>0.000000</overhead>
--<diskspace>9.063500</diskspace>
--</index>
.....
--<statement>
--<statementnum>11</statementnum>
--<statementtext>
```

```
--
-- select
-- c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice,
-- sum(l_quantity) from samp.customer, samp.orders,
-- samp.lineitem where o_orderkey in( select
-- l_orderkey from samp.lineitem group by l_orderkey
-- having sum(l_quantity) > 300 ) and c_custkey
-- = o_custkey and o_orderkey = l_orderkey group by
-- c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice
-- order by o_totalprice desc, o_orderdate fetch first
-- 100 rows only
--</statementtext>
--<objects>
--<identifier>
--<name>MQT612152202490000</name>
--<schema>ZILIO2 </schema>
--</identifier>
--<identifier>
--<name>ORDERS</name>
--<schema>SAMP </schema>
--</identifier>
--<identifier>
--<name>CUSTOMER</name>
--<schema>SAMP </schema>
--</identifier>
--<identifier>
--<name>IDX612152235020000</name>
--<schema>ZILIO2 </schema>
--</identifier>
--<identifier>
--<name>IDX612152235030000</name>
--<schema>ZILIO2 </schema>
--</identifier>
--<identifier>
--<name>IDX612152211360000</name>
--<schema>ZILIO2 </schema>
--</identifier>
--</objects>
--<benefit>2091459.000000</benefit>
--<frequency>1</frequency>
--</statement>
```

This XML structure can contain more than one column. The column cardinality (that is, the number of values in each column) is included and, optionally, the HIGH2KEY and LOW2KEY values.

The base table on which an index is defined is also included. Ranking of indexes and MQTs can be done using the benefit value. You can also rank indexes using (benefit - overhead) and MQTs using (benefit - 0.5 * overhead).

Following the list of indexes and MQTs is the list of statements in the workload, including the SQL text, the statement number for the statement, the estimated performance improvement (benefit) from the recommendations, and the list of tables, indexes, and MQTs that were used by the statement. The original spacing in the SQL text is preserved in this output example, but the SQL text is normally split into 80 character commented lines for increased readability.

Existing indexes or MQTs are included in the output if they are being used to execute a workload.

MDC and database partitioning recommendations are not explicitly shown in this XML output example.

After some minor modifications, you can run this output file as a CLP script to create the recommended objects. The modifications that you might want to perform include:

- Combining all of the **RUNSTATS** commands into a single **RUNSTATS** invocation against the new or modified objects
- Providing more usable object names in place of system-generated IDs
- Removing or commenting out any data definition language (DDL) for objects that you do not want to implement immediately

Using the Design Advisor

You can run the Design Advisor by invoking the **db2advis** command.

Procedure

1. Define your workload. See "Defining a workload for the Design Advisor".
2. Run the **db2advis** command against this workload.

   Note: If the statistics on your database are not current, the generated recommendations are less reliable.
3. Interpret the output from **db2advis** and make any necessary modifications.
4. Implement the Design Advisor recommendations, as appropriate.

Defining a workload for the Design Advisor

When the Design Advisor analyzes a specific workload, it considers factors such as the type of statements that are included in the workload, the frequency with which a particular statement occurs, and characteristics of your database to generate recommendations that minimize the total cost of running the workload.

About this task

A *workload* is a set of SQL statements that the database manager must process during a period of time. The Design Advisor can be run against:

- A single SQL statement that you enter inline with the **db2advis** command
- A set of dynamic SQL statements that were captured in a Db2 snapshot
- A set of SQL statements that are contained in a workload file

You can create a workload file or modify a previously existing workload file. You can import statements into the file from several sources, including:

- A delimited text file
- An event monitor table
- Explained statements in the EXPLAIN_STATEMENT table
- Recent SQL statements that were captured with a Db2 snapshot
- Workload manager activity tables
- Workload manager event monitor tables by using the **-wlm** option from the command line

After you import the SQL statements into a workload file, you can add, change, modify, or remove statements and modify their frequency.

Procedure

- To run the Design Advisor against dynamic SQL statements:
  1. Reset the database monitor with the following command:

     ```
     db2 reset monitor for database database-name
     ```

2. Wait for an appropriate amount of time to allow for the execution of dynamic SQL statements against the database.
3. Invoke the **db2advis** command using the **-g** parameter. If you want to save the dynamic SQL statements in the ADVISE_WORKLOAD table for later reference, use the **-p** parameter as well.

- To run the Design Advisor against a set of SQL statements in a workload file:
  1. Create a workload file manually, separating each SQL statement with a semicolon, or import SQL statements from one or more of the sources listed previously.
  2. Set the frequency of the statements in the workload. Every statement in a workload file is assigned a frequency of 1 by default. The frequency of an SQL statement represents the number of times that the statement occurs within a workload relative to the number of times that other statements occur. For example, a particular SELECT statement might occur 100 times in a workload, whereas another SELECT statement occurs 10 times. To represent the relative frequency of these two statements, you can assign the first SELECT statement a frequency of 10; the second SELECT statement has a frequency of 1. You can manually change the frequency or weight of a particular statement in the workload by inserting the following line after the statement: `- - # SET FREQUENCY n`, where *n* is the frequency value that you want to assign to the statement.
  3. Invoke the **db2advis** command using the **-i** parameter followed by the name of the workload file.
- To run the Design Advisor against a workload that is contained in the ADVISE_WORKLOAD table, invoke the **db2advis** command using the **-w** parameter followed by the name of the workload.

Using the Design Advisor to convert from a single-partition to a multi-partition database

You can use the Design Advisor to help you convert a single-partition database into a multi-partition database.

About this task

In addition to making suggestions about new indexes, materialized query tables (MQTs), and multidimensional clustering (MDC) tables, the Design Advisor can provide you with suggestions for distributing data.

Procedure

1. Use the **db2licm** command to register the partitioned database environment license key.
2. Create at least one table space in a multi-partition database partition group.

   Note: The Design Advisor can only suggest data redistribution to existing table spaces.
3. Run the Design Advisor with the partitioning option specified on the **db2advis** command.
4. Modify the **db2advis** output file slightly before running the DDL statements that were generated by the Design Advisor. Because database partitioning must be set up before you can run the DDL script that the Design Advisor generates, suggestions are commented out of the script that is returned. It is up to you to transform your tables in accordance with the suggestions.

Design Advisor limitations and restrictions

There are certain limitations and restrictions associated with Design Advisor, about indexes, materialized query tables (MQTs), multidimensional clustering (MDC) tables, and database partitioning.

Restrictions on index

- Indexes that are suggested for MQTs are designed to improve workload performance, not MQT refresh performance.
- A clustering RID index is for MDC tables. The Design Advisor will include clustering RID indexes as an option rather than create an MDC structure for the table.
- The Version 9.7 Design Advisor does not suggest you use partitioned indexes on a partitioned table. All indexes are must be used with an explicit NOT PARTITIONED clause.

Restrictions on MQT

- The Design Advisor does not suggest the use of incremental MQTs. If you want to create incremental MQTs, you can convert REFRESH IMMEDIATE MQTs into incremental MQTs with your choice of staging tables.
- Indexes that are for MQTs are designed to improve workload performance, not MQT refresh performance.
- If update, insert, or delete operations are not included in the workload, the performance impact of updating a REFRESH IMMEDIATE MQT is not considered.
- It is suggested that REFRESH IMMEDIATE MQTs have unique indexes created on the implied unique key, which is composed of the columns in the GROUP BY clause of the MQT query definition.

Restrictions on MDC

- An existing table must be populated with sufficient data before the Design Advisor considers MDC for the table. A minimum of twenty to thirty megabytes of data is suggested. Tables that are smaller than 12 extents are excluded from consideration.
- MDC requirements for new MQTs will not be considered unless the sampling option, -r, is used with the **db2advis** command.
- The Design Advisor does not make MDC suggestions for typed, temporary, or federated tables.
- Sufficient storage space (approximately 1% of the table data for large tables) must be available for the sampling data that is used during the execution of the **db2advis** command.
- Tables that have not had statistics collected are excluded from consideration.
- The Design Advisor does not make suggestions for multicolumn dimensions.

Restrictions on database partitioning

The Design Advisor provides advise about database partitioning only for Db2 Enterprise Server Edition.

Additional restrictions

Temporary simulation catalog tables are created when the Design Advisor runs. An incomplete run can result in some of these tables not being dropped. In this situation, you can use the Design Advisor to drop these tables by restarting the

utility. To remove the simulation catalog tables, specify both the -f option and the -n option (for -n, specifying the same user name that was used for the incomplete execution). If you do not specify the -f option, the Design Advisor will only generate the DROP statements that are required to remove the tables; it will not actually remove them.

Note: As of Version 9.5, the -f option is the default. This means that if you run **db2advis** with the MQT selection, the database manager automatically drops all local simulation catalog tables using the same user ID as the schema name.

You should create a separate table space on the catalog database partition for storing these simulated catalog tables, and set the DROPPED TABLE RECOVERY option on the CREATE or ALTER TABLESPACE statement to OFF. This enables easier cleanup and faster Design Advisor execution.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Printed in USA