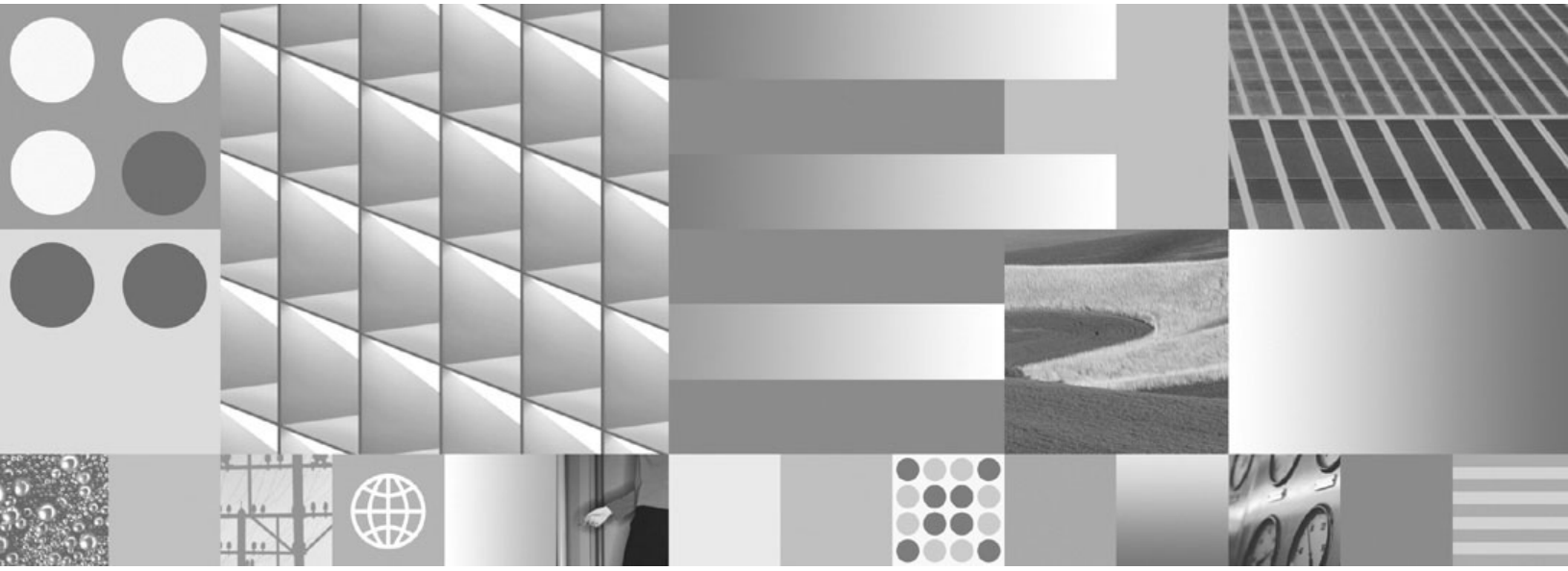
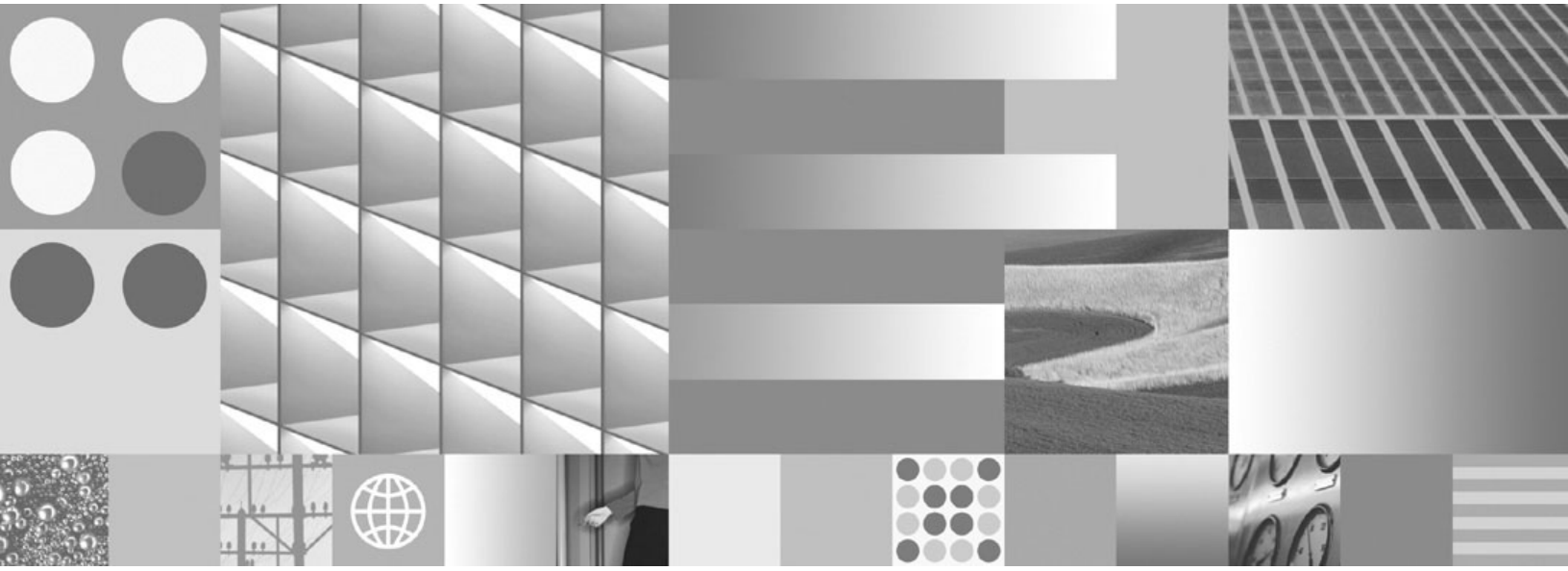
DB2 Version 9.5
for Linux, UNIX, and Windows



Tuning Database Performance

SC23-5867-00

DB2 Version 9.5
for Linux, UNIX, and Windows



Tuning Database Performance

SC23-5867-00

Note

Before using this information and the product it supports, read the general information under Appendix B, "Notices," on page 427.

Edition Notice

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at www.ibm.com/planetwide

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Part 1. Elements of performance

Performance is the way a computer system behaves given a particular workload. Performance is measured in terms of system response time, throughput, and availability. Performance is also affected by:

- The resources available in your system
- How well those resources are used and shared.

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

- Processing a larger, or more demanding, work load without increasing processing costs

  For example, to increase the work load without buying new hardware or using more processor time
- Obtaining faster system response times, or higher throughput, without increasing processing costs
- Reducing processing costs without degrading service to your users

Translating performance from technical terms to economic terms is difficult. Performance tuning certainly costs money in terms of user time as well as processor time, so before you undertake a tuning project, weigh its costs against its possible benefits. Some of these benefits are tangible:

- More efficient use of resources
- The ability to add more users to the system.

Other benefits, such as greater user satisfaction because of quicker response time, are intangible. All of these benefits should be considered.

Chapter 1. Performance tuning guidelines

The following guidelines should help you develop an overall approach to performance tuning.

Remember the law of diminishing returns: Your greatest performance benefits usually come from your initial efforts. Further changes generally produce smaller and smaller benefits and require more and more effort.

Do not tune just for the sake of tuning: Tune to relieve identified constraints. If you tune resources that are not the primary cause of performance problems, this has little or no effect on response time until you have relieved the major constraints, and it can actually make subsequent tuning work more difficult. If there is any significant improvement potential, it lies in improving the performance of the resources that are major factors in the response time.

Consider the whole system: You can never tune one parameter or system in isolation. Before you make any adjustments, consider how it will affect the system as a whole.

Change one parameter at a time: Do not change more than one performance tuning parameter at a time. Even if you are sure that all the changes will be beneficial, you will have no way of evaluating how much each change contributed. You also cannot effectively judge the trade-off you have made by changing more than one parameter at a time. Every time you adjust a parameter to improve one area, you almost always affect at least one other area that you may not have considered. By changing only one at a time, this allows you to have a benchmark to evaluate whether the change does what you want.

Measure and reconfigure by levels: For the same reasons that you should only change one parameter at a time, tune one level of your system at a time. You can use the following list of levels within a system as a guide:

- 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 may be corrected by applying service either to your hardware, or to your software, or to both. Do not spend excessive time monitoring and tuning your system when simply applying service may make it unnecessary.

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 may spend money on additional disk storage only to find that you do not have the processing power or the channels to exploit it.

Put fall-back procedures in place before you start tuning: As noted earlier, some tuning can cause unexpected performance results. If this leads to poorer performance, it should be reversed and alternative tuning tried. If the former setup is saved in such a manner that it can be simply recalled, the backing out of the incorrect information becomes much simpler.

Chapter 2. Developing a performance improvement process

The performance improvement process is an iterative, long term approach to monitoring and tuning aspects of performance. Depending on the result of monitoring, you and your performance team 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 the patterns of data access. Different kinds of applications have different performance requirements.

Consider the following outline of the performance improvement process as a guideline.

To develop a performance improvement process:

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

   Even if you think that more than one resource requires tuning, or if several tuning options are available for the resource you want to tune, make only one change at a time so that you can make sure that your tuning efforts are producing the effect you want. At some point, you can no longer improve performance by tuning the database server and applications. Then you need to upgrade your hardware.

Actual performance tuning requires trade-offs among system resources. For example, to provide improved I/O performance you might increase buffer pool sizes, but larger buffer pools require more memory, which might degrade other aspects of performance.

Chapter 3. 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 by listening to your users. You can usually determine where to start looking for a problem by asking a few simple questions. For example, you might ask your users:

- 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 problems, are they connected to the same local area network?
- Do the the problems seem to be related to a specific transaction or application program?
- Do you notice any pattern in the problem occurrence? For example, does the problem occur at a specific time of day, such as during lunch hour, or is it more or less continuous?

Chapter 4. Performance tuning limits

Tuning can make only a certain amount of change in the efficiency of a system. Consider how much time and money you should spend on improving system performance, and how much spending additional time and money will help the users of the system.

For example, tuning can often improve performance if the system encounters a performance bottleneck. If you are close to the performance limits of your system and the number of users increases by about ten percent, the response time is likely to increase by much more than ten percent. In this situation, you need to determine how to counterbalance this degradation in performance by tuning your system.

However, there is a point beyond which tuning cannot help. At this point, consider revising your goals and expectations within the limits of your environment. For significant performance improvements, you might need to add more disk storage, faster CPU, additional CPUs, more main memory, faster communication links, or a combination of these.

Chapter 5. DB2 architecture and process overview

General information about DB2® architecture and processes can help you understand detailed information provided for specific topics.

The following figure shows a general overview of the architecture and processes for IBM® DB2 Version 9.1.

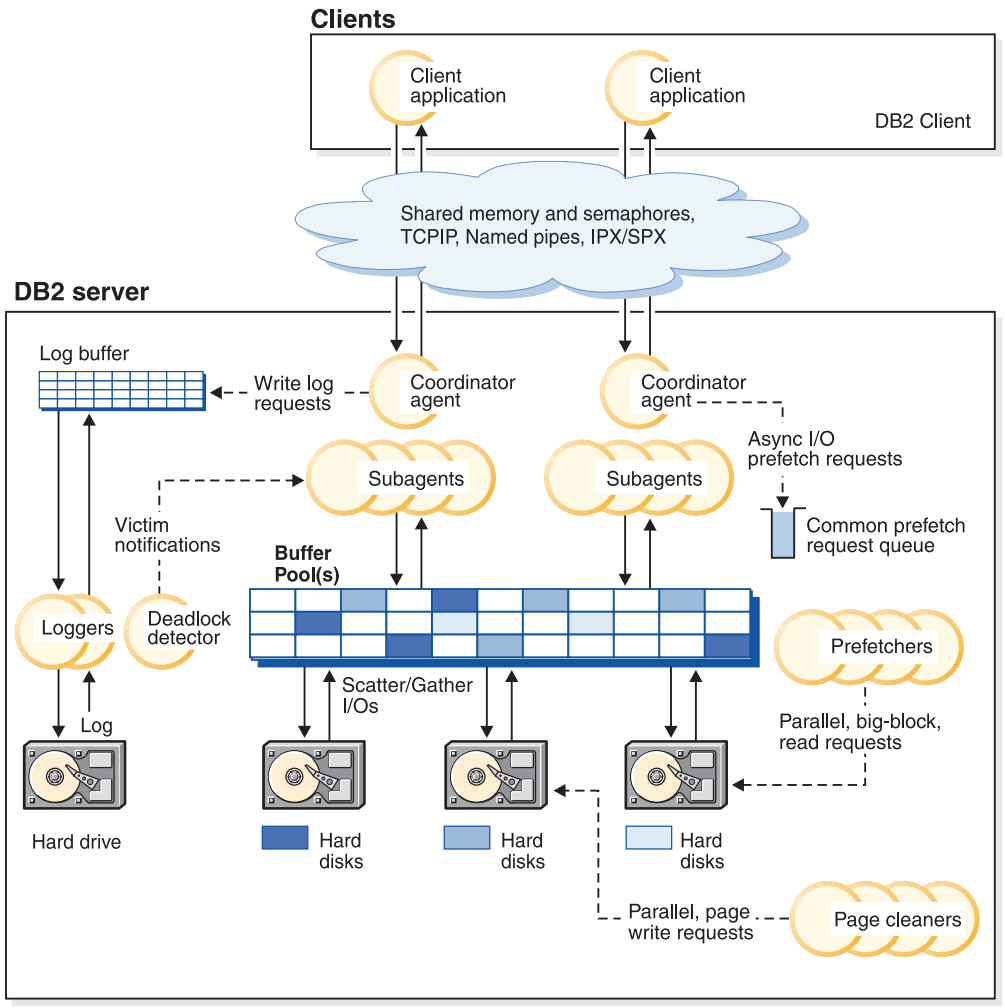


Figure 1. Architecture and Processes Overview

On the client side, either local or remote applications, or both, 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). In all figures in this section, EDUs are shown as circles or groups of circles. EDUs are implemented as threads on all platforms in Version 9.5. DB2 agents are the most

common type of EDUs. 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 the client application requests. Multiple subagents can be assigned if the machine where the server resides has multiple processors or is part of a partitioned database. For example, in a symmetric multiprocessing (SMP) environment, multiple SMP subagents can exploit the many processors.

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

Buffer pools are areas of database server memory where database pages of user table 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. If more of the data needed by applications is present in a buffer pool, less time is required to access the data than to find it on disk.

The configuration of the buffer pools, as well as prefetcher and page cleaner EDUs, controls how quickly data can be accessed and how readily available it is to applications.

- **Prefetchers** retrieve data from disk and move it into the buffer pool before applications need the data. For example, applications needing to scan through large volumes of data would have to wait for data to be moved from disk into the 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 to the buffer pool. If you have multiple disks for storage of the database data, the data can be striped across the disks. Striping data lets the prefetchers use multiple disks at the same time to retrieve data.
- **Page cleaners** move data from the buffer pool back out 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 the pages being retrieved by the prefetchers.

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

The DB2 Process Model

Knowledge of the DB2 process model can help you determine the nature of a problem because it helps you to understand how the database manager and its associated components interact.

The process model used by all DB2 servers facilitates the communication that occurs between database servers and clients and local applications. It also ensures that database applications are isolated from resources such as database control blocks and critical database files.

The DB2 server must perform many different tasks, for example processing database application requests or ensuring log records are written out to disk. Each task is typically performed by a separate engine dispatchable unit (EDU). In previous releases, most EDUs were implemented using separate processes in Linux® and UNIX® environments, and using operating system threads within a main DB2 server process in Windows. Starting with Version 9.5, the DB2 server on Linux and UNIX environments is now also threaded, and as such, EDUs are now implemented using operating system threads on both UNIX and Windows.

There are many advantages to using a multithreaded architecture in the DB2 server. A new thread requires less memory and operating system resources than a process, as some operating system resources can be shared among all threads within the same process. Also, on some platforms, the context switch time for threads is cheaper than for processes, which can improve performance. Most significantly though, using a threaded model on all platforms makes the DB2 server easier to configure, as it is simpler to allocate more EDUs as needed, and it is possible to dynamically allocate memory that needs to be shared by multiple EDUs (in a process-based model, one EDU cannot see memory allocated from a different EDU). See the Simplified Memory Configuration and the Agent and Process Model Configuration sections for further details on these enhancements.

In previous releases, on Linux and UNIX systems, the ps system command or the db2_local_ps command could be used to list all active DB2 EDUs. Starting in version 9.5, those commands will no longer list any EDU threads within the db2sysc process. As an alternative, you can now use the db2pd command, with the -edus option, to list all EDU threads that are active. This command works on both UNIX and Windows systems.

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

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, using private memory, interprocess communication (IPC), or remote communication protocols, as needed.

The DB2 architecture provides a **firewall** so that applications run in a different address space from DB2. The firewall protects the database and the database manager from applications, stored procedures, and user-defined functions (UDFs). A firewall maintains the integrity of the data in the databases, because it disables application programming errors from overwriting internal buffers or files of the database manager. The firewall also improves reliability, because application errors cannot crash the database manager.

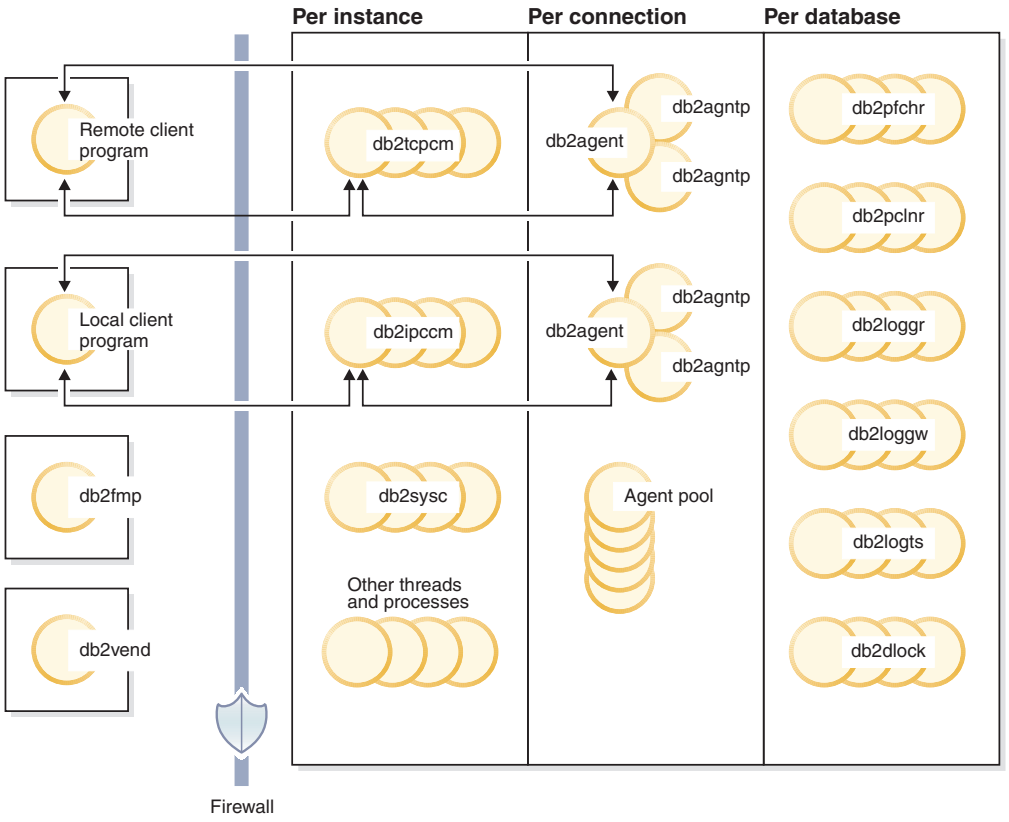


Figure 2. Process model for DB2 systems

The following list provides additional details on the objects shown in the figure:

Client Programs

Client programs run remotely or on the same machine as the database server. They make their first contact with the database through a listener. A coordinator agent (db2agent) is then assigned to them.

Listeners

Client programs make initial contact with communication listeners, which are started when DB2 is started. There is a listener for each configured communication 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 client applications, whether they are local or remote, are allocated a corresponding coordinator agent (**db2agent**). When the coordinator agent is created, it performs all database requests on behalf of the application.

In environments where the database partitioning feature (DPF) is enabled or those where *intra_query* parallelism has been enabled, the coordinator agent will distribute database requests to subagents (db2agntp and db2agnts respectively).

These agents perform the requests for the application. Once the coordinator agent is created, it handles all database requests on behalf of its application by coordinating subagents (db2agntp) that perform requests on the database. Subagents that are associated with an application but are currently idle are identified by the name **db2agnta**.

A coordinator agent may 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 process model will instantiate other types of agents to execute specific operations such as independent coordinator agents or subcoordinator agents. For example, the independent coordinator agent **db2agnti** is used to run event monitors and the subcoordinator agent **db2agnsc** is used to parallelize execution of the restart of a database following an abrupt shutdown.

Idle agents reside in an agent pool. These agents are available for requests from coordinator agents operating on behalf of client programs, or from subagents operating on behalf of existing coordinator agents. Having an appropriately-sized idle agent pool can help performance in configurations that have significant application workloads, as idle agents can be immediately used as needed, rather than having to allocate a completely new agent for each application connection, which involves creating a thread and allocating and initializing memory and other resources. Starting with version 9.5, DB2 can also automatically manage the size of the idle agent pool if desired.

db2fmp

The fenced mode process. It is responsible for executing fenced stored procedures and user-defined functions outside the firewall. The db2fmp process is always a separate process but may be multithreaded depending on the types of routines it executes.

db2vend

This is a process to execute vendor code on behalf of an EDU, for instance to execute the user-exit program for log archiving (UNIX-only).

Database EDUs

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

- **db2pfchr**, for buffer pool prefetchers
- **db2pclnr**, for buffer pool page cleaners
- **db2loggr**, for manipulating log files to handle transaction processing and recovery
- **db2loggw**, for writing log records to the log files.
- **db2logts**, tracks which table spaces have log records in which log files. This information is recorded in the DB2TSCHG.HIS file in the database directory. It is used to speed up the forward phase of table space rollforward recovery.

- **db2dlock**, for deadlock detection. In a multi-partitioned database environment, an additional thread called **db2glock** is used to coordinate the information gathered from the db2dlock EDU on each partition. db2glock runs only on the catalog partition.
- **db2taskd**, for distribution of background database tasks. The tasks are executed by threads called **db2taskp**.
- **db2hadrp**, HADR primary server thread
- **db2hadrs**, HADR standby server thread
- **db2lfr**, for log file readers that processes individual log files
- **db2shred** processes individual log records within log pages
- **db2redom**, for the redo master. During recovery, processes redo log records and assigns log records to redo workers for processing.
- **db2redow**, for the redo worker. During recovery, processes redo log records at the request of the redo master.
- **db2logmgr**, for the log manager. Manages log files for a recoverable database
- **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
    - **l** - local file event monitor
    - **t** - table event monitor
    - **gp** - global piped event monitor
    - **lp** - local piped event monitor

    where **%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**- the EDU ID of the agent that controls the backup or restore session
    - **%2**- a sequential value used to disambiguate among (possibly many) threads belonging 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.

Database Server Threads and Processes

The system controller (**db2sysc on UNIX, db2syscs.exe on Windows**) must exist in order for the database server to function. Also, the following threads and processes may be started to carry out various tasks:

- **db2resync**, the resync agent that scans the global resync list
- **db2wdog**, the watchdog on UNIX and Linux operating systems that handles abnormal terminations
- **db2fcms**, the fast communications manager sender daemon
- **db2fcmr**, the fast communications manager receiver daemon
- **db2pdbc**, the parallel system controller, which handles parallel requests from remote nodes (used only in a partitioned database environment).

- **db2cart**, for archiving log files when accessing a database configured with USEREXIT enabled
- **db2fmtlg**, for formatting log files, when accessing a database configured with LOGRETAIN enabled, but with USEREXIT disabled
- **db2panic**, the panic agent, which handles urgent requests after agent limits have been reached at a particular node (used only in a partitioned database environment)
- **db2srvlst**, manages lists of addresses for systems such as DB2 for z/OS.
- **db2fmd**, the fault monitor daemon
- **db2disp**, the client connection concentrator dispatcher
- **db2acd**, autonomic computing daemon hosting the health monitor and automatic maintenance utilities. This process was formerly called **db2hmon**.
- **db2licc**, manages installed DB2 licenses
- **db2thcln**, recycles resources when an EDU terminates (UNIX-only)
- **db2aiothr**, manages Asynchronous I/O requests for the database partition (UNIX-only)
- **db2alarm**, notifies EDUs when their requested timer has expired (UNIX-only)
- **db2sysc**, main system controller EDU, handles critical DB2 server events

Deadlocks

A deadlock is created when two applications are each locking data needed by the other, resulting in a situation when neither application can continue execution. For example, in the following diagram, there are two applications running concurrently: Application A and Application B. The first step of application A is to update the first row of Table 1, and the second step is to update the second row of Table 2. Application B updates the second row of Table 2 first, and then the first row of Table 1. At one point in time, T1, Application A is executing its first step, locking the first row of Table 1 to update it. At the same time, Application B locks the second row in Table 2 to make an update. At T2, Application A tries to execute the next step and requests a lock on the second row in Table 2 for an update. However, at the same time, Application B is trying to lock and update the first row in Table 1. Since Application A will not release its lock on the first row of Table 1 until it is able to complete an update of the second row in Table 2, and Application B will not release its lock on the second row on Table 2 until it can lock and update the first row of Table 1, a deadlock occurs. The applications can wait forever until one application releases the lock on the held data.

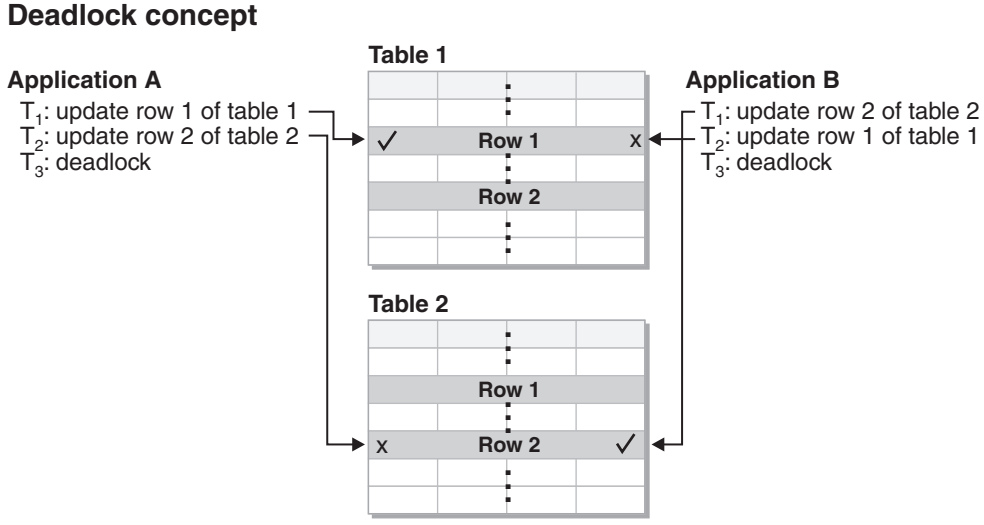


Figure 3. Deadlock between applications

Because applications do not voluntarily release locks on data that they need, a deadlock detector process is required to break deadlocks and allow application processing to continue. As its name suggests, the deadlock detector monitors the information about agents waiting on locks, awakening at intervals specified by the *dlchktime* 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 is complete, the locks that belonged to the victim process are released, and the other processes involved in the deadlock can continue.

To ensure good performance, select the proper interval for the deadlock detector. An interval that is too short causes unnecessary overhead, and an interval that is too long allows a deadlock to delay a process for an unacceptable amount of time. For example, a wake-up interval of 5 minutes may allow a deadlock to exist for almost 5 minutes, which can seem like a long time for short transaction processing. It is important to balance the possible delays in resolving deadlocks with the overhead of detecting them.

Note:

1. In a partitioned database environment, the *dlchktime* configuration parameter interval is applied only at the catalog node. If a large number of deadlocks are detected in a partitioned database environment, increase the value of the *dlchktime* parameter to account for lock waits and communication waits.
2. In a partitioned database, each database partition sends *lock graphs* to the database partition that contains the system catalog views. Global deadlock detection takes place on this database partition.

A different problem occurs when an application with more than one independent process that accesses the database is structured to make deadlocks likely. For example, an application in which several processes access the same table for reads and then writes. If the processes do read-only SQL or XQuery queries and then do SQL updates on the same table, the chance of deadlocks increases because of

potential contention between the processes for the same data. For instance, if two processes read the table and then update the table, process A might try to get an X lock on a row on which process B has an S lock. To avoid such deadlocks, applications that access data with the intention of modifying it should do one of the following:

- Use the FOR UPDATE OF clause when performing a select operation. This clause ensures that a U lock is imposed when process A attempts to read the data. Row blocking is disabled.
- Use the WITH RR USE AND KEEP UPDATE LOCKS clause or the WITH RS USE AND KEEP UPDATE LOCKS clause when performing the query. Either clause ensures that a U lock is imposed when process A attempts to read the data and allows row blocking.

At the same time a database is created, a detailed deadlocks event monitor is created. As with any monitor, there is some overhead associated with this event monitor.

To limit the amount of disk space that this event monitor consumes, the event monitor deactivates and a message is written to the administration notification log when it has reached its maximum number of output files. Removing output files that are no longer needed allows the event monitor to reactivate on the next database activation.

If you do not want the detailed deadlocks event monitor, the event monitor can be dropped using the command:

```
DROP EVENT MONITOR db2detaildeadlock
```

In a federated system environment in which an application accesses nicknames, the data requested by the application might not be available because of a deadlock at a data source. When this happens, DB2 relies on the deadlock handling facilities at the data source. If deadlocks occur across more than one data source, DB2 relies on data source timeout mechanisms to break the deadlock.

To log more information about deadlocks, set the database manager configuration parameter *diaglevel* to four. The logged information includes the locked object, the lock mode, and the application holding the lock. The current dynamic SQL and XQuery statements or static package names might also be logged. Dynamic SQL and XQuery statements are logged only at *diaglevel* four.

Default deadlock event monitor

At database creation time, a deadlock event monitor called DB2DETAILDEADLOCK is created and activated by default . It starts automatically when the instance activates. When this monitor is active, diagnostic information is collected on the first occurrence of a deadlock, allowing for investigation into the cause without requiring a reproduction.

To limit the amount of disk space that this event monitor consumes, the event monitor deactivates and a message is written to the administration notification log when it has reached its maximum number of output files. Removing output files that are no longer needed allows the event monitor to reactivate on the next database activation.

The command is created using the following statement:

```
db2 create event monitor db2detaildeadlock for deadlocks with details write to file
'db2detaildeadlock' maxfiles 20 maxfilesize 512 buffersize 17 blocked append autostart
```

The WITH DETAILS clause provides information such as the statement that was executing at the time the deadlock occurred, and the locklist (provided sufficient memory exists in the dbmon heap).

The output files are created under the directory 'db2event' in your database directory. If you do not specify a location when you create the database, then the location of the database directory can be determined by viewing the database manager configuration parameter *dftdbpath*. For example, on a sample database on AIX®, the event monitor output files might be found in the directory:
`NODE0000/SQL00001/db2event/db2detaildeadlock`

The event monitor writes to a file up to a maximum of 20 files, each with a size of 2M (512 4K pages). When the *maxfilesize* (2M) is reached, the output file is closed and a new one is opened. When the number of files created reaches *maxfiles* (20), the monitor shuts itself down and a message similar to the following is recorded in the administration notification log:

```
2004-12-01-22.58.24.968000 Instance: DB2 Node: 000 PID: 1116(db2syscs.exe) TID: 2540
Appid: *LOCAL.DB2.041202080328 database monitor sqm__evmgr::log_ev_err Probe:2
Databse:XXX ADM2001W The Event Monitor "DB2DETAILDEADLOCK" was deactivated because the
MAXFILES and MAXFILES CREATE EVENT MONITOR parameters' limits have been reached.
```

Removing output files that are no longer needed allows the monitor to start up again on the next database activation. If you do not want the detailed deadlocks event monitor, then the event monitor can be dropped using the command:

```
DROP EVENT MONITOR db2detaildeadlock
```

If you are concerned about deadlocks, do not drop this monitor.

Disk storage overview

Disk-storage performance factors

The hardware that makes up your system can influence the performance of your system. As an example of the influence of hardware on performance, consider some of the implications associated with disk storage.

Four aspects of disk-storage management affect performance:

- **Division of storage**

  How you divide a limited amount of storage between indexes and data and among table spaces determines to a large degree how each will perform in different situations.
- **Wasted storage**

  Wasted storage in itself may not affect the performance of the system that is using it, but wasted storage is a resource that could be used to improve performance elsewhere.
- **Distribution of disk I/O**

  How well you balance the demand for disk I/O across several disk storage devices, and controllers can affect how fast the database manager can retrieve information from disks.
- **Lack of available storage**

  Reaching the limit of available storage can degrade overall performance.

Part 2. Tables and indexes

Chapter 6. Table and index management for standard tables

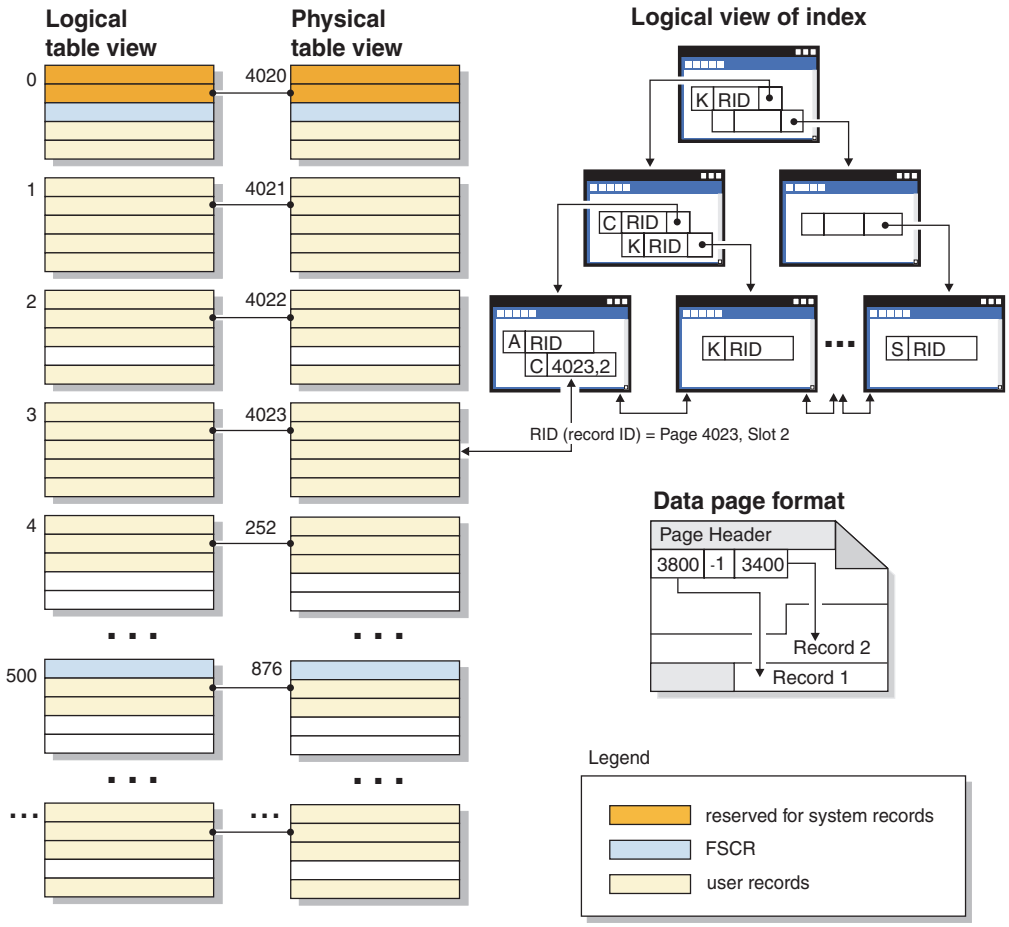


Figure 4. Logical table, record, and index structure for standard tables

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. The maximum number of records which can fit on one page can be found in Table 1. Most pages contain only user records. However, a small number of pages include special internal records, that are used by DB2 to manage the table. For example, in a standard table there is a Free Space Control Record (FSCR) on every 500th data page. These records map the free space for new records on each of the following 500 data pages (until the next FSCR). This available free space is used when inserting records into the table.

Logically, index pages are organized as a B-tree which can efficiently locate records in the table that have a given key value. The number of entities on an index page is not fixed but depends on the size of the key. For tables in DMS table spaces, record identifiers (RIDs) in the index pages use table space-relative page numbers,

not object-relative page numbers. This allows 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 header begins each data page. After the page header there is a slot directory. Each entry in the slot directory corresponds to a different record on the page. The entry itself is the byte-offset into the data page where the record begins. Entries of minus one (-1) correspond to deleted records.

Record identifiers and pages

Record identifiers (RIDs) are a page number followed by a slot number. Type-2 index records also contain an additional field called the ridFlag. The ridFlag stores information about the status of keys in the index, such as whether this key has been marked deleted. Once the index is used to identify a RID, the RID is used to get to the correct data page and slot number on that page. Once a record is assigned a RID, it does not change until a table reorganization.

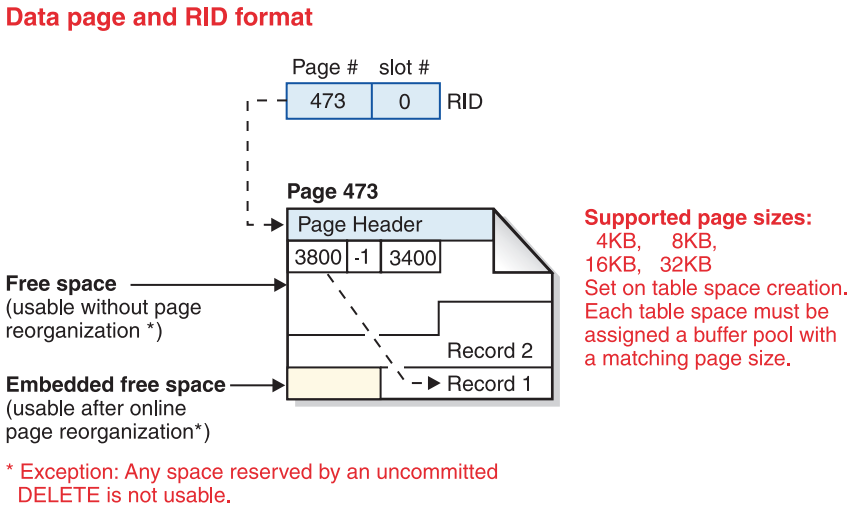


Figure 5. Data page and record-id (RID) format

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. RIDs are redefined based on movement of records on a data page to take advantage of the usable free space.

DB2 supports different page sizes. Use larger page sizes for workloads that tend to access rows sequentially. For example, sequential access is used for Decision Support applications or where temporary tables are extensively used. Use smaller page sizes for workloads that tend to be more random in their access. For example, random access is used in OLTP environments.

Index management in standard tables

DB2 indexes use an optimized B-tree implementation based on an efficient and high concurrency index management method using write-ahead logging.

The optimized B-tree implementation has bi-directional pointers on the leaf pages that allows a single index to support scans in either forward or reverse direction. Index page are usually split in half except at the high-key page where a 90/10 split

is used. That is, the high ten percent of the index keys are placed on a new page. This type of index page split is useful for workloads where INSERT requests are often completed with new high-keys.

Starting in Version 8.1, DB2 uses type-2 indexes. If you migrate from earlier versions of DB2, both type-1 and type-2 indexes are in use until you reorganize indexes or perform other actions that convert type-1 indexes to type-2. The index type determines how deleted keys are physically removed from the index pages.

- For type-1 indexes, keys are removed from the index pages during key deletion and index pages are freed when the last index key on the page is removed.
- For type-2 indexes, index keys are removed from the page during key deletion only if there is an X lock on the table. If keys cannot be removed immediately, they are marked deleted and physically removed later. For more information, refer to the section that describes type-2 indexes.

If you have enabled online index defragmentation by setting the MINPCTUSED clause to a value greater than zero when you created the index, index leaf pages can be merged online. The value that you specify is the threshold for the minimum percentage of space used on the index leaf pages. After a key is removed from an index page, if the percentage of space used on the page is at or below the value given, then 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. Online index defragmentation can improve space reuse, but if the MINPCTUSED value is too high then the time taken to attempt a merge increases and becomes less likely to succeed. The recommended value for this clause is fifty percent or less.

Note: Because online defragmentation occurs only when keys are removed from an index page, in a type-2 index it does not occur if keys are merely marked deleted, but have not been physically removed from the page.

The INCLUDE clause of the CREATE INDEX statement allows the inclusion of a specified column or columns on the index leaf pages in addition to the key columns. This can increase the number of queries that are eligible for index-only access. However, this can also increase the 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 that of updating key columns, but more than that of updating columns that do not appear in the index. Ordering the index B-tree is only done using the key columns and not the included columns.

Chapter 7. Table and index management for MDC tables

Table and index organization for multi-dimensional clustering (MDC) tables is based on the same logical structures as standard table organization. Like standard tables, MDC tables are organized into pages that contain rows of data, divided into columns, and the rows on each page are identified by row IDs (RIDs). In addition, however, the pages of MDC tables are grouped into extent-sized blocks. For example, in the illustration below, which shows a table with an extent size of four, the first four pages, numbered 0 through 3, are the first block in the table. The next set of pages, numbered 4 through 7, are the second block in the table.

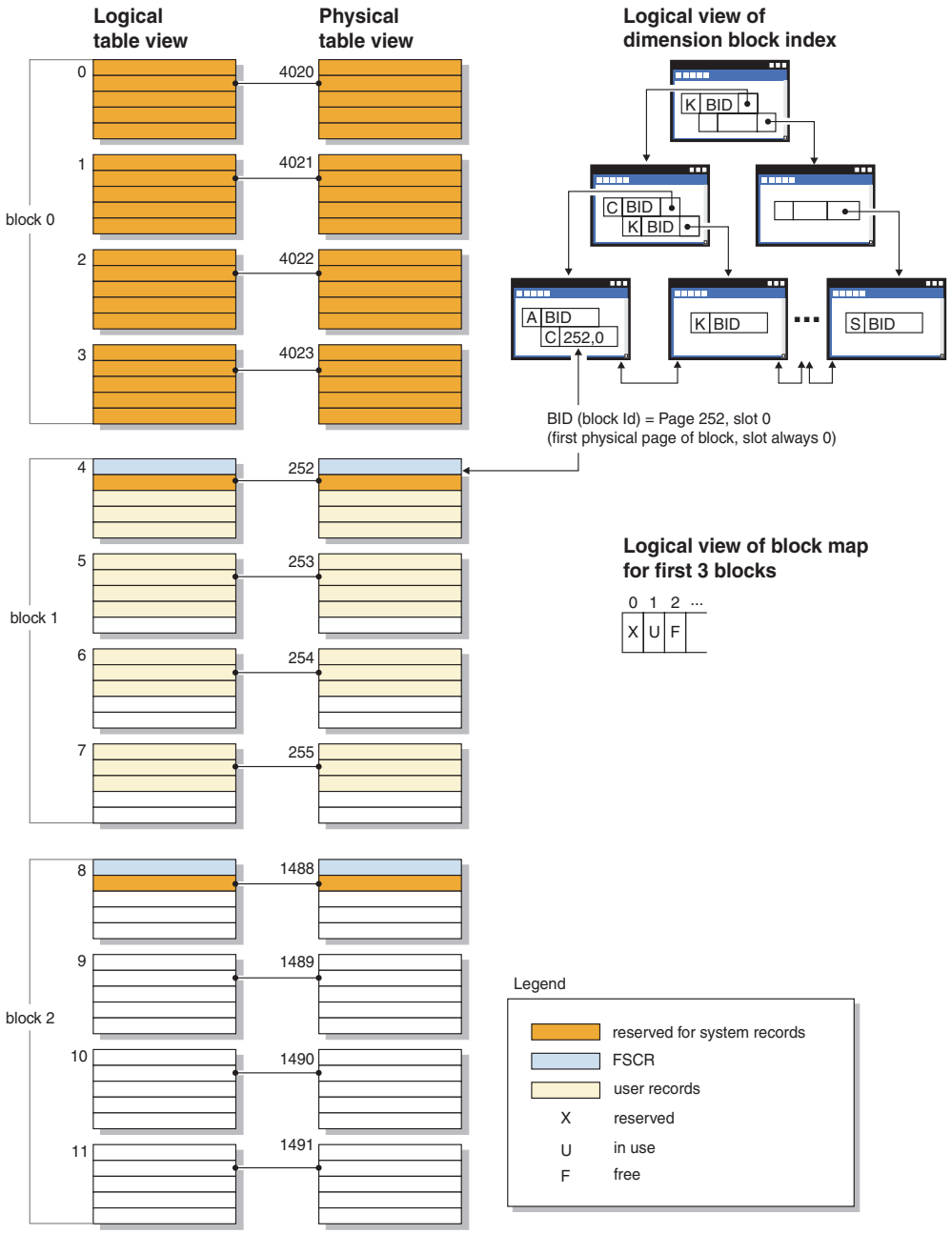


Figure 6. Logical table, record, and index structure for MDC tables

The first block contains special internal records that are used by DB2 to manage the table, including the free-space control record (FSCR). In subsequent blocks, the first page contains the FSCR. An FSCR maps the free space for new records that exists on each of the pages in 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 kinds of 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. The composite block index is used to maintain clustering during insert and update activity.

The optimizer considers access plans which utilize 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 results in more efficient performance and minimizes I/O.

In addition, you can create specific RID indexes if analysis of data access plans indicates that such indexes would improve query performance.

Along with the dimension block indexes and the composite block index, MDC tables maintain a block map that contains a bitmap that indicates the availability status of each block. The following attributes are coded in the bitmap list:

- X (reserved): the first block contains only system information for the table.
- U (in use): this block is used and associated with a dimension block index
- L (loaded): this block has been loaded by a current load operation
- C (check constraint): this block is set by the load operation to specify incremental constraint checking during the load.
- T (refresh table): this block is set by the load operation to specify that AST maintenance is required.
- F (free): If no other attribute is set, the block is considered free.

Because each block has an entry in the block map file, the file grows as the table grows. This file is stored as a separate object. In an SMS table space it is a new file type. In a DMS table space, it has a new object descriptor in the object table.

Chapter 8. Asynchronous index cleanup for MDC tables

You can enhance the performance of a rollout deletion, an efficient method of 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.

During a standard rollout delete, indexes are cleaned up synchronously with the delete. For tables that contain many record ID (RID) indexes, a significant portion of the delete time is spent removing index keys that reference the table rows being deleted. You can speed up the rollout by specifying that these indexes are to be cleaned up after the delete has been committed.

To take advantage of AIC for MDC tables, you need to 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 and issuing the SET CURRENT MDC ROLLOUT MODE statement. During a deferred index cleanup rollout, blocks are marked as rolled out without an update to the RID indexes until after the transaction commits. Block identifier (BID) indexes are still cleaned up during the delete because they don't require row-level processing.

Rollout AIC is invoked when a rollout delete is committed or, if the database was shut down, when the table is first accessed following a restart of the database. While AIC is in progress, any queries against the indexes work, including those accessing the index being cleaned up.

There one coordinating cleaner per MDC table. The index cleanup for multiple rollouts is consolidated in the cleaner. The cleaner spawns a cleanup agent for each RID index, and the 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 the priority by using the SET UTIL_IMPACT_PRIORITY command or the db2UtilityControl API.

Monitoring

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. Use the LIST UTILITIES monitor command to display a utility monitor entry for each index being cleaned up. You can also query the number of blocks in the table currently being cleaned up through deferred index cleanup rollout (BLOCKS_PENDING_CLEANUP) by using the SYSPROC.ADMIN_GET_TAB_INFO_V95 table function. To query the number of MDC table blocks pending cleanup at the database level, use the GET SNAPSHOT command.

In the following sample output for the LIST UTILITIES, progress is indicated by the number of pages in each index that have been cleaned up. Each phase listed in the output represents one of the RID indexes being cleaned for the table.

```
db2 LIST UTILITIES SHOW DETAILS output.
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
```

Chapter 9. Index structure

The database manager uses a B+ tree structure for index storage. A B+ tree has one or more levels, as shown in the following diagram, in which RID means row ID:

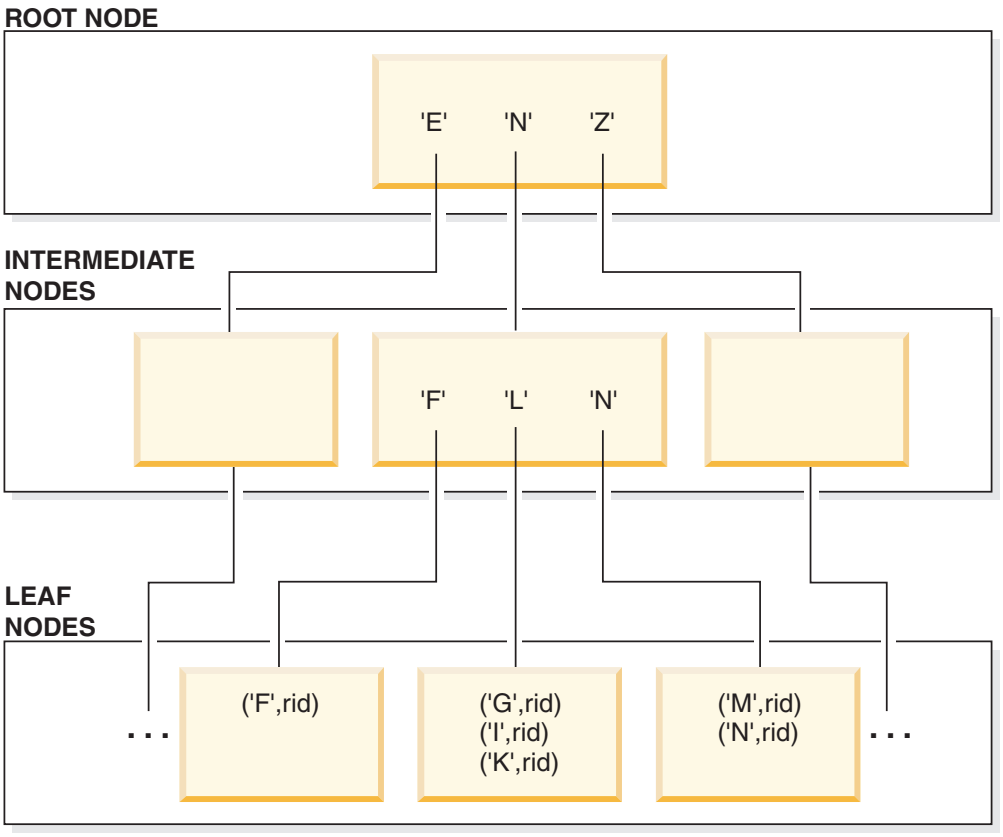


Figure 7. B+ Tree Structure

The top level is called the *root node.* The bottom level consists of *leaf nodes* in which the index key values are stored with pointers to the row in the table that contains the key value. Levels between the root and leaf node levels are called *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 contains one key for each node at 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, if an index has three levels as shown in the figure, then to find an index key value, the index manager searches the root node for the first key value greater than or equal to the key being looked for. The root node key points to a specific intermediate node. The index manager follows this procedure through the intermediate nodes until it finds the leaf node that contains the index key that it needs.

The figure shows the key being looked for as "I". The first key in the root node greater than or equal to "I" is "N". This 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". This points to a specific leaf node where the index key for "I" and its

corresponding row ID is found. The row ID identifies the corresponding row in the base table. The leaf node level can also contain pointers to previous leaf nodes. These pointers allow 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 only possible if the index was created with the ALLOW REVERSE SCANS clause.

For multi-dimensional clustering (MDC) tables, a block index is created automatically for each clustering dimension that you specify for the table. An additional composite block index is also created, which contains a key part for each column involved in any dimension of the table. These indexes contain pointers to block IDs (BIDs) instead of RIDs, and provide data-access improvements.

In DB2 Version 8.1 and later, indexes can be of either type 1 or type 2. A *type-1 index* is the older index style. Indexes that you created in earlier versions of DB2 are of this kind.

A type-2 index is somewhat larger than a type-1 index and provides features that minimize next-key locking. The one-byte *ridFlag* byte stored for each RID on the leaf page of a type-2 index is used to mark the RID as logically deleted so that it can be physically removed later. For each variable length column included in the index, one additional byte stores the actual length of the column value. Type-2 indexes might also be larger than type-1 indexes because some keys might be marked deleted but not yet physically removed from the index page. After the DELETE or UPDATE transaction is committed, the keys marked deleted can be cleaned up.

Part 3. Processes

Chapter 10. Reducing logging overhead to improve query performance

All databases maintain log files that keep records of database changes. There are two logging strategy choices:

- Circular logging, in which the log records fill the log files and then overwrite the initial log records in the initial log file. The overwritten log records are not recoverable.
- Retain log records, in which a log file is archived when it fills with log records. New log files are made available for log records. Retaining log files enables **roll-forward recovery**. Roll-forward recovery reapplies changes to the database based on completed units of work (transactions) that are recorded in the log. You can specify that roll-forward recovery is to the end of the logs, or to a particular point in time before the end of the logs.

Regardless of the logging strategy, all changes to regular data and index pages are written to the log buffer. The data in the log buffer is written to disk by the logger process. In the following circumstances, query processing must wait for log data to be written to disk:

- On COMMIT
- Before the corresponding data pages are written to disk, because DB2 uses write-ahead logging. The benefit of write-ahead logging is that when a transaction completes by executing the COMMIT statement, not all of the changed data and index pages need to be written to disk.
- Before some changes are made to metadata, most of which result from executing DDL statements
- On writing log records into the log buffer, if the log buffer is full

DB2 manages writing log data to disk in this way in order to minimize processing delay. In an environment in which many short concurrent transactions occur, most of the processing 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, with additional delay caused by log I/O overhead. To balance application response time against such logging delay, set the *mincommit* database configuration parameter to a value greater than 1. This setting might cause longer delay for COMMIT from 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 is defined on the CREATE TABLE statement without the NOT LOGGED clause. Changes to allocation pages for LONG or LOB data types are logged like regular data pages.

Chapter 11. Improving insert performance

When SQL statements use INSERT to place new information in a table, an INSERT search algorithm first searches the Free Space Control Records (FSCRs) to find a page with enough space. However, even when the FSCR indicates a page has enough free space, the space may not be usable because it is reserved by an uncommitted DELETE from another transaction. To ensure that uncommitted free space is usable, you should COMMIT transactions frequently.

The setting of the DB2MAXFSCRSEARCH registry variable determines the number of FSCRs in a table that are searched for an INSERT. The default value for this registry variable is five. If no space is found within the specified number of FSCRs, the inserted record is appended at the end of the table. To optimize INSERT speed, subsequent records are also appended to the end of the table until two extents are filled. After the two extents are filled, the next INSERT resumes searching at the FSCR where the last search ended.

Note: To optimize for INSERT speed at the possible expense of faster table growth, set the DB2MAXFSCRSEARCH registry variable to a small number. To optimize for space reuse at the possible expense of INSERT speed, set DB2MAXFSCRSEARCH to a larger number.

After all FSCRs in the entire table have been searched in this way, the records to be inserted are appended without additional searching. Searching using the FSCRs is not done again until space is created somewhere in the table, such as following a DELETE.

There are two other INSERT algorithm options, as follows:

- APPEND MODE

  In this mode, new rows are always appended to the end of the table. No searching or maintenance of FSCRs takes place. This option is enabled using the ALTER TABLE APPEND ON statement, and can improve performance for tables that only grow, like journals.
- A clustering index is defined on the table.

  In this case, 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, the attempt is made to put the record into the surrounding pages. If there is still no success, the FSCR search algorithm, described above, is used, except that a worst-fit approach is used instead of a first-fit approach. This worst-fit approach tends to choose pages with more free space. This method establishes a new clustering area for rows with this key value.

  When you define a clustering index on a table, use ALTER TABLE... PCTFREE before you either load or reorganize the table. The PCTFREE clause specifies the percentage of free space that should remain on the data page of the table after loading and reorganizing. This increases the probability that the cluster index operation will find free space on the appropriate page.

Chapter 12. Update processing

When an agent updates a page, the database manager uses the following protocol to minimize the I/O required by the transaction and ensure recoverability.

1. The page to be updated is pinned and latched with an exclusive lock. A log record is written to the log buffer describing how to redo and undo the change. As part of this action, a log sequence number (LSN) is obtained and is stored in the page header of the page being updated.
2. The change is made to the page.
3. The page is unlatched and unfixed.

   The page is considered to be "dirty" because changes to the page have not been written out to disk.
4. The log buffer is updated.

   Both the data in the log buffer and the "dirty" data page are forced to disk.

For better performance, these I/Os are delayed until a convenient point, such as during a lull in the system load, or until necessary to ensure recoverability, or to limit recovery time. Specifically, a "dirty" page is forced to disk at the following times:

- When another agent chooses it as a victim.
- When a page cleaner acts on the page as the result of:
  - Another agent choosing it as a victim.
  - The *chngpgs_thresh* database configuration parameter percentage value is exceeded. When this value is exceeded, asynchronous page cleaners wake up and write changed pages to disk.

    If proactive page cleaning is enabled, this value is irrelevant and does not trigger page cleaning.
  - The *softmax* database configuration parameter percentage value is exceeded. Once exceeded, asynchronous page cleaners 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 on the hate list drops too low. Page cleaners only react to this condition under the proactive page cleaning method.
  - When a dirty page currently contributes to, or is projected to contribute to an LSNGAP condition. Page cleaners only react to this condition under the proactive page cleaning method.
- When the page was updated as part of a table which has the NOT LOGGED INITIALLY clause invoked and a COMMIT statement is issued. When the COMMIT statement is executed, all changed pages are flushed to disk to ensure recoverability.

Chapter 13. Client-server processing model

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 database machine. Local applications are directly attached to the database at the server machine.

How DB2 manages client connections depends on whether the connection concentrator is on or off. The connection concentrator is ON when the *max_connections* database manager configuration parameter is set larger than the *max_coordagents* configuration parameter.

- If the connection concentrator is OFF, each client application is assigned a unique 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 similar applications with many relatively small transactions, the connection concentrator improves performance by allowing many more client applications to be connected. It also reduces system resource use for each connection.

Each of the circles in the DB2 server of the following figure represent engine dispatchable units (EDUs) which are implemented using operating system threads.

A means of communicating between an application and the database manager must be established before the work the application wants done at the database can be carried out.

At A1 in the figure below, a local client establishes communications first through the db2ipccm. At A2, the db2ipccm works with a db2agent EDU, which becomes the coordinator agent for the application requests from the local client. The coordinator agent then contacts the client application at A3 to establish shared memory communications between the client application and the coordinator. The application at the local client is connected to the database at A4.

At B1 in the figure below, a remote client establishes communications through the db2tcpcm EDU. If any other communications protocol is chosen, the appropriate communication manager is used. The db2tcpcm EDU establishes TCP/IP communication between the client application and the db2tcpcm. It then works with a db2agent at B2, 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, which is then connected to the database at B5.

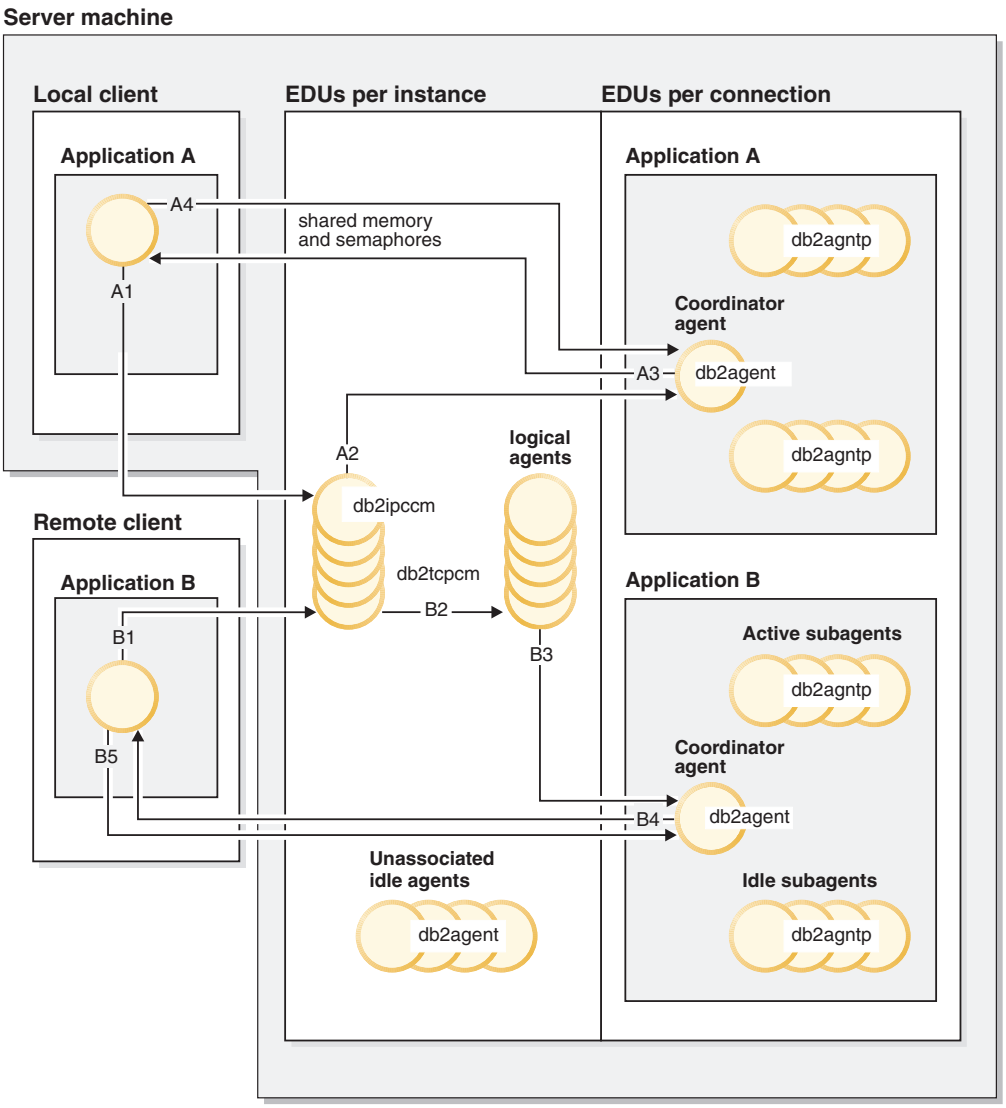


Figure 8. Process model overview

Other things to notice in this figure:

- 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, and enabled intra-partition parallelism environments, the coordinator agents distribute database requests to subagents (db2agntp). The subagents perform the requests for the application.
- There is an agent pool (db2agent) where idle and pooled agents wait for new work.
- Other EDUs manage client connections, logs, two-phase COMMITs, backup and restore tasks, and other tasks.

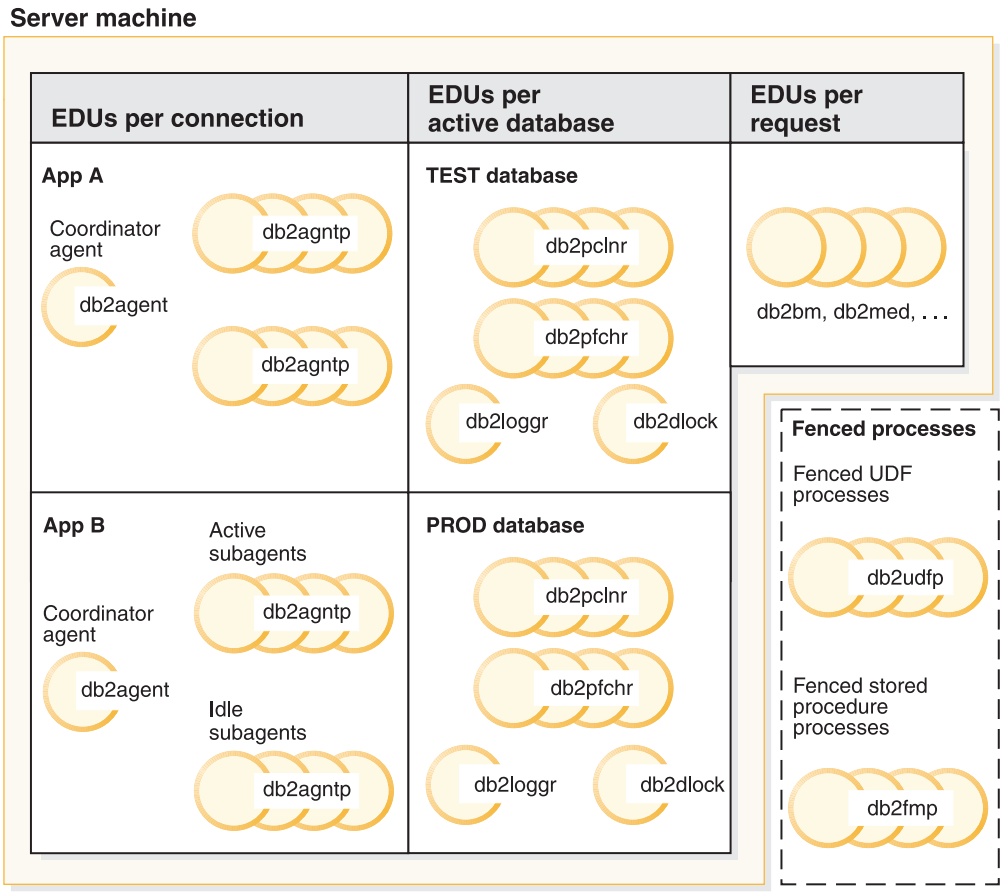


Figure 9. Process model, part 2

This figure shows additional engine dispatchable units (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).

Fenced user-defined functions (UDFs) and stored procedures, which are not shown in the figure, are managed to minimize costs associated with their creation and destruction. The default for the *keepfenced* database manager configuration parameter is "YES", which keeps the stored procedure process available for re-use at the next stored 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 need to be rigorously tested before being used.

The multiple database partition processing model is a logical extension of the single database partition processing model. In fact, a single common code base supports both modes of operation. The following figure shows the similarities and differences between the single database partition processing model as seen in the previous two figures, and the multiple database partition processing model.

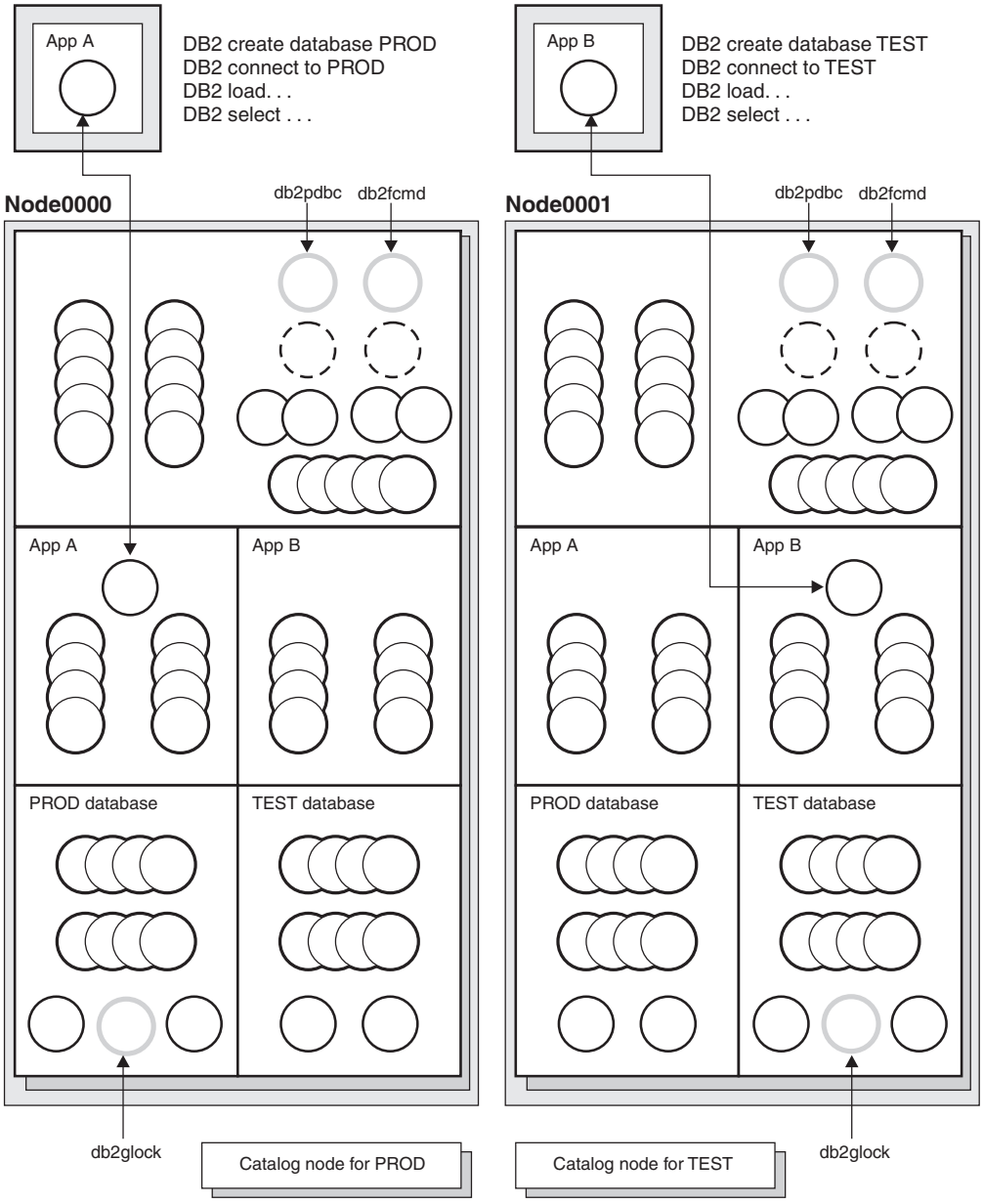


Figure 10. Process model and multiple database partitions

Most engine dispatchable units (EDUs) are the same between the single database partition processing model and the multiple database partition processing model.

In a multiple database partition (or node) environment, one of the database partitions is the catalog node. The catalog keeps all of the information relating to the objects in the database.

As shown in the figure above, because Application A creates the PROD database on Node0000, the catalog for the PROD database is created on this node. Similarly, because Application B creates the TEST database on Node0001, the catalog for the TEST database is created on this node. You might want to create your databases on different nodes to balance the extra activity associated with the catalogs for each database across the nodes in your system environment.

There are additional EDUs (db2pdbc and db2fcmd) associated with the instance and these are found on each node in a multiple partition database environment. These EDUs are needed to coordinate requests across database partitions and to enable the Fast Communication Manager (FCM).

There is also an additional EDU (db2glock) associated with the catalog node for the database. This EDU controls global deadlocks across the nodes where the active database is located.

Each CONNECT from an application is represented by a connection that is associated with a coordinator agent to handle the connection. The *coordinator agent* is the agent that communicates with the application, receiving requests and sending replies. It can either satisfy the request itself or coordinate multiple subagents to work on the request. The database partition where the coordinator agent exists is called the *coordinator node* of that application. The coordinator node can also be set with the SET CLIENT CONNECT_NODE command.

Parts of the database requests from the application are sent by the coordinator node to subagents at the other database partitions; and all results from the other database partitions are consolidated at the coordinator node before being sent back to the application.

The database partition where the CREATE DATABASE command was issued is called the “catalog node” for the database. It is at this database partition that the catalog tables are stored. Typically, all user tables are distributed across a set of nodes.

Note: Any number of database partitions can be configured to run on the same machine. This is known as a “multiple logical partition”, or “multiple logical node”, 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.

Part 4. Quick-start tips for performance tuning

When you start a new instance of DB2, consider the following suggestions for a basic configuration:

- Use the Configuration Advisor in the Control Center to get advice about reasonable beginning defaults for your system. The defaults shipped with DB2 should be tuned for your unique hardware environment.

  Gather information about the hardware at your site so you can answer the wizard questions. You can apply the suggested configuration parameter settings immediately or let the wizard create a script based on your answers and run the script later.

  This script also provides a list of the most commonly tuned parameters for later reference.
- Use other wizards in the Control Center and Client Configuration Assistant for performance-related administration tasks. These tasks are usually those in which you can achieve significant performance improvements by spending spend a little time and effort.

  Other wizards can help you improve performance of individual tables and general data access. These wizards include the Create Database, Create Table, Index, and Configure Multisite Update wizards. The Health Center provides a set of monitoring and tuning tools.
- Use the Design Advisor tool from the Control Center or using the `db2advis` command to find out what indexes, materialized query tables, multi-dimensional clustering tables, and database partitions will improve query performance.
- Use the ACTIVATE DATABASE command to start databases. In a partitioned database, this command activates the database on all database partitions and avoids the startup time required to initialize the database when the first application connects.

  Note: If you use the ACTIVATE DATABASE command, you must shut down the database with the DEACTIVATE DATABASE command. The last application that disconnects from the database does not shut it down.
- Consult the summary tables that list and briefly describe each configuration parameter available for the database manager and each database.

  These summary tables contain a column that indicates whether tuning the parameter results in high, medium,low, or no performance changes, either for better or for worse. Use this table to find the parameters that you might tune for the largest performance improvements.

Chapter 14. Operational performance

Memory allocation in DB2

Memory allocation and de-allocation occurs at various times in DB2. Memory may be allocated to a particular memory area when a specified event occurs, such as when an application connects, or it may be re-allocated based on a change in a configuration parameter setting.

The figure below shows the different areas of memory that the database manager allocates for various uses and the configuration parameters that allow you to control the size of this memory. Note that in an Enterprise Server Edition environment that comprises multiple logical database partitions, each database partition has its own Database Manager Shared Memory set.

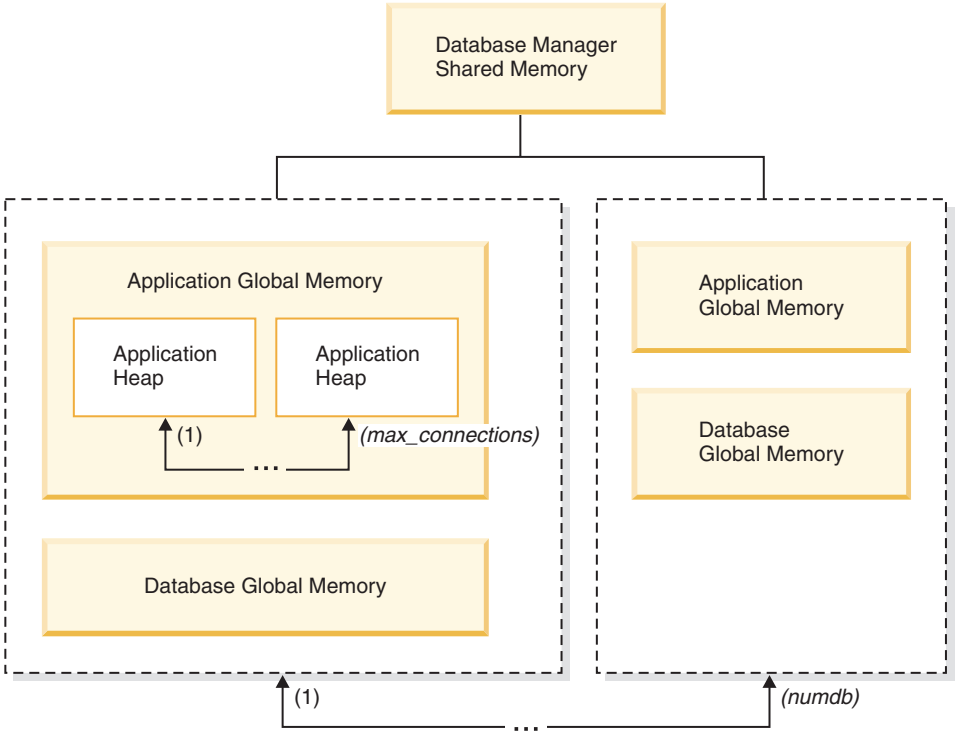


Figure 11. Types of memory used by the Database Manager

Memory is allocated for each instance of the database manager when the following events occur:

- **When the database manager is started (`db2start`):** Database manager global shared memory (also known as instance shared memory) is allocated and remains allocated until the database manager is stopped (`db2stop`). This area contains information that the database manager uses to manage activity across all database connections. DB2 automatically controls the database manager global shared memory size.
- **When a database is activated or connected to for the first time:** Database global memory is allocated. 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** 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 runtime based on the needs of the database. You can set **database_memory** to allocate more memory than is needed initially so that the additional memory can be dynamically distributed later.

The following memory areas can be dynamically adjusted, for example, to decrease memory allocated to one area and increase memory in another area.

- Buffer pools (using the ALTER BUFFERPOOL DDL statement)
- Database heap (including log buffers)
- Utility heap
- Package cache
- Catalog cache
- Lock list (This memory area can only be increased dynamically, and not decreased.)

In an environment in which the database manager intra-partition parallelism configuration parameter (intra_parallel) is enabled, in an environment in which the connection concentrator is enabled, or in an environment where the database partitioning feature (DPF) is enabled, the shared sort heap is also allocated as part of the database global memory. Also, if the **sheapthres** database manager configuration parameter is set to 0 (default), all sorts will use database global memory.

- **When an application connects to a database:** An application heap is allocated. Each application has its own application heap. If desired, you can limit the amount of memory any one application can allocate using the **applheapsz** configuration parameter, or limit overall application memory consumption using the **appl_memory** configuration parameter.

  The database manager configuration parameter **max_connections** sets an upper limit on the number of applications that can attach to the instance or connect to any databases that exist in the instance. Since each application that attaches to a database involves the allocation of some memory, allowing a larger number of concurrent applications will potentially use more memory.
- **When an agent is created:** Agent private memory is allocated for an agent when the agent is assigned as the result of a connect request or a new SQL request in a parallel environment. Agent private memory is allocated for the agent and contains memory that is used only by this specific agent, such as the private sort heap.

The figure also specifies the following configuration parameter settings, which 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 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 might be allocated increases if you increase the value of this parameter.
- **maxappls**

  This parameter specifies the maximum number of applications that can simultaneously connect to a single database. It affects the amount of memory

that might be allocated for agent private memory and application global memory for that database. Note that this parameter can be set differently for every database.

Two other parameters that need to be considered are **max_coordagents** and **max_connections** both of which apply at the instance level (per node on a DPF instance).

- **max_connections**

  This parameter limits the number of connections or instance attachments that can access the DB2 server at any one time (per node on a DPF instance).

- **max_coordagents**

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

The memory tracker, invoked by the db2mtrk command, allows you to view the current allocation of memory within the instance, including the following types of information for each memory pool:

- Current size
- Maximum size (hard limit)
- Largest size (high water mark)

Database manager shared memory

Database manager memory is organized into several different memory areas. The following figure shows how database manager memory is allocated. The configuration parameters shown allow you to control the sizes of the various memory areas.

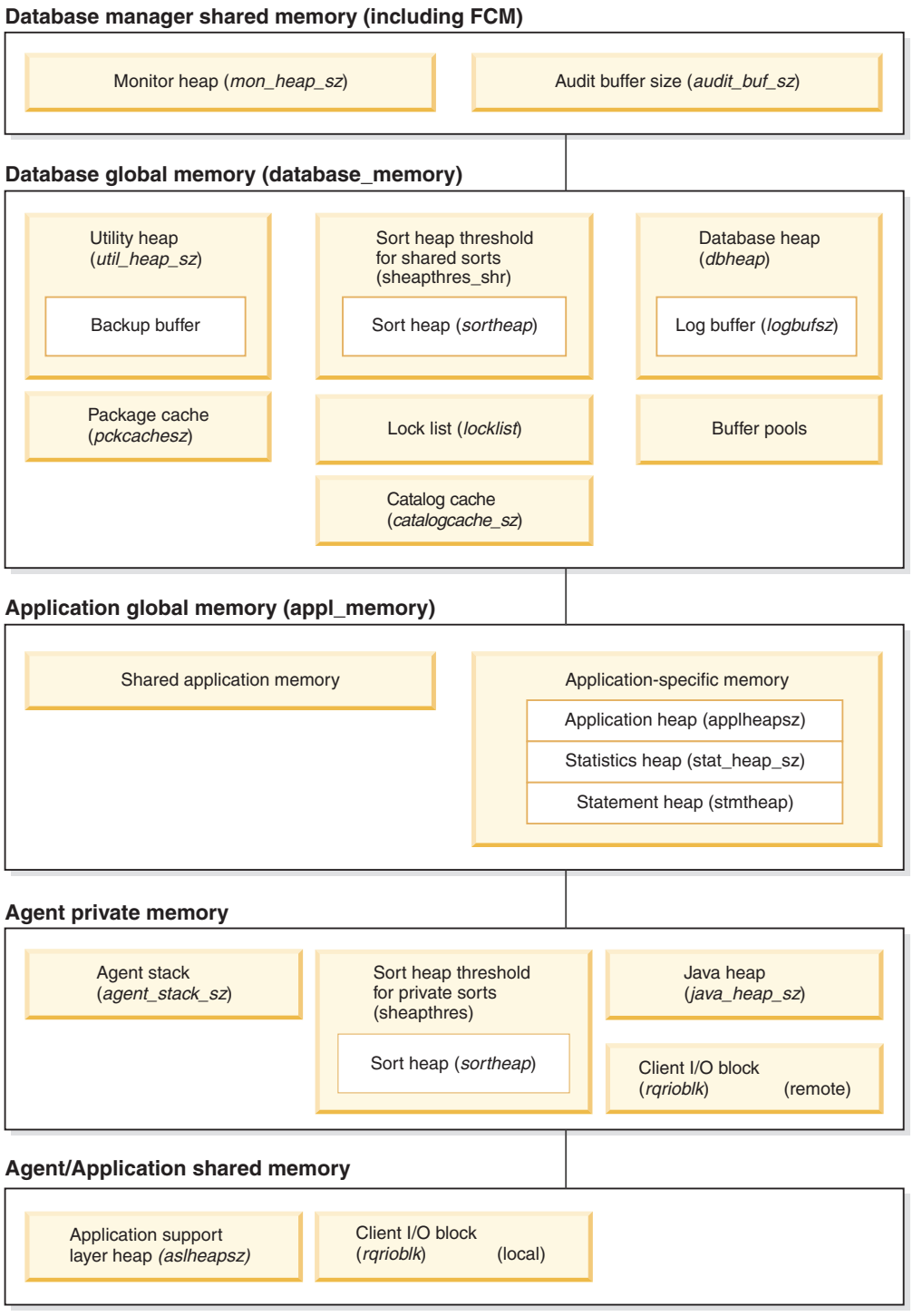


Figure 12. How memory is used by the database manager

Audit buffer
This memory area is used in database auditing activities. The size of this buffer is determined by the **audit_buf_sz** configuration parameter.

Monitor heap
This memory area is used for database system monitoring data. The size of this area is determined by the **mon_heap_sz** configuration parameter.

Fast communication manager (FCM) buffer pool

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 following configuration parameters:

- The **database_memory** parameter provides a lower bound for the size of the database global memory.
- The following parameters or factors control the size of the database global memory area:
  - The size of the buffer pools.
  - Maximum Storage for Lock List (**locklist**)
  - Database Heap (**dbheap**)
  - Utility Heap Size (**util_heap_sz**)
  - Package Cache Size (**pckcachesz**)
  - Shared Sort Heap (**sheapthres_shr**)
  - Catalog cache (**catalogcache_sz**)

Application Global Memory

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

- Application Heap Size (**applheapsz**)
- Statement Heap Size (**stmtheap**)
- Statistics Heap Size (**stat_heap_sz**)

Agent Private Memory

- Each agent requires its own private memory region. The DB2 server will create as many agents as it needs and accommodate given configured memory resources. You can control the maximum number of coordinator agents using the **max_coordagents** parameter.
- The maximum size of each agent's private memory region is determined by the values of the following parameters:
  - Private Sort Heap Size (**sheapthres** and **sortheap**)
  - Agent Stack Size (**agent_stack_sz**)

Agent/Application Shared Memory

- The total number of agent/application shared memory segments for local clients is limited by the lower of the following database configuration parameters:
  - The total of **maxappls** for all active databases
  - The value of **max_coordagents**.

  Note: In configurations where engine concentration is enabled (**max_connections** > **max_coordagents**), application memory consumption will be limited by **max_connections**.
- Agent/Application Shared Memory is also affected by the following database configuration parameters:
  - The Application Support Layer Heap Size (**aslheapsz**) parameter

– The Client I/O Block Size (**rqrioblk**) parameter

The FCM buffer pool and memory requirements

In a partitioned database system, the Database Manager Shared Memory and FCM Buffer Pool are as shown below.

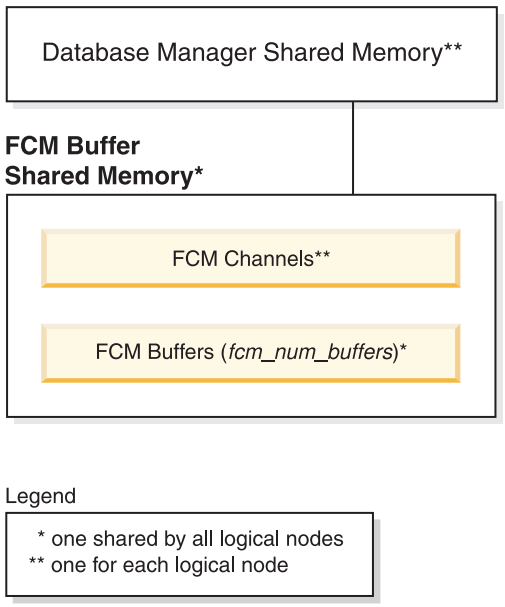


Figure 13. FCM buffer pool when multiple logical nodes are used

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

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

Tuning memory allocation parameters

The first rule for setting memory-allocation parameters is never to set them at their highest values unless such a value has been carefully justified. This rule applies even to systems with the maximum amount of memory. Many parameters that affect memory can allow the database manager easily and quickly to take up all of the available memory on a computer. In addition, managing large amounts of memory requires additional work on the part of the database manager and thus incurs more overhead.

Some UNIX operating systems allocate swap space when a process allocates memory and not when a process is paged out to swap space. For these systems, make sure that you provide as much paging space as total shared memory space.

For most configuration parameters, memory is only committed as it is required and the parameter settings determine the maximum size of a particular memory heap. In the following cases, however, the full amount of memory specified by the parameter is allocated:

- Maximum Storage for Lock List (*locklist*)
- Application Support Layer Heap Size (*aslheapsz*)
- Number of FCM Buffers (*fcm_num_buffers*)
- Number of FCM Channels (*fcm_num_channels*)
- Buffer Pools

Notes:

- Benchmark tests provide the best information about setting appropriate values for memory parameters. In benchmarking, typical and worst-case SQL statements are run against the server and the values of the parameters are modified until the point of diminishing return for performance is found. If performance versus parameter value is graphed, the point at which the curve begins to plateau or decline indicates the point at which additional allocation provides no additional value to the application and is therefore simply wasting memory.
- The upper limits of memory allocation for several parameters may be beyond the memory capabilities of existing hardware and operating systems. These limits allow for future growth.
- 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 several memory consumers including sort memory, package cache, lock list memory, and buffer pools.

The following table lists the self-tuning memory topics by category:

Table 1. Overview of self-tuning memory information

Category	Related topics
General information and restrictions	• "Self-tuning memory" on page 58 • "Self tuning memory operational details and limitations" on page 61 • self_tuning_mem-self_tuning_mem - Self-tuning memory configuration parameter
Enablement and disablement	• "Enabling self tuning memory" on page 59 • "Disabling self tuning memory" on page 59
Monitoring	• "Determining which memory consumers are enabled for self tuning" on page 60
DPF considerations	• "Self tuning memory in partitioned database environments" on page 62 • "Using self-tuning memory in partitioned database environments" on page 64

Table 1. Overview of self-tuning memory information (continued)

Category	Related topics
Configuration parameters that can be automatically tuned	• "database_memory - Database shared memory size" in *Configuration Parameter Reference* • "locklist - Maximum storage for lock list" in *Configuration Parameter Reference* • "maxlocks - Maximum percent of lock list before escalation" in *Configuration Parameter Reference* • "pckcachesz - Package cache size" in *Configuration Parameter Reference* • "sheapthres_shr - Sort heap threshold for shared sorts in" *Configuration Parameter Reference* • "sortheap - Sort heap size" in *Configuration Parameter Reference*

Self-tuning memory

Starting in DB2 Version 9, a new 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, package cache, locking memory and sort memory.

The tuner works within the memory limits defined by the **database_memory** configuration parameter. The value of **database_memory** itself can be automatically tuned as well. When self-tuning is enabled for **database_memory** (when you set it 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 the current database requirements. For example, if the current database requirements are high and there is sufficient free memory on the system, more memory will be consumed by 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 you do not enable the **database_memory** parameter for self-tuning (you do not set it to AUTOMATIC), the entire database uses the amount of memory that you specify for the parameter, distributing it across the database memory consumers as required. You can specify the amount of memory used by the database in two ways: by setting **database_memory** to a numeric value or by setting it to COMPUTED. In the second case, the total amount of memory is computed based on the sum of the initial values of the database memory heaps at database startup time.

In addition to tuning database shared memory by using the **database_memory** configuration parameter, you can enable other memory consumers for self-tuning as follows:

- For buffer pools, use the ALTER BUFFERPOOL and CREATE BUFFERPOOL statements.
- For the package cache, use the **pckcachesz** configuration parameter.
- For locking memory, use the **locklist** and **maxlocks** configuration parameters.
- For sort memory, use the **sheapthres_shr** and **sortheap** configuration parameters.

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

1. Enable self tuning for the database by setting *self_tuning_mem* to ON. You can set *self_tuning_mem* to ON using the UPDATE DATABASE CONFIGURATION command, the SQLFUPD API, or through the **Change Database Configuration Parameter** window in the Control Center.
2. To enable self tuning of memory areas controlled by memory configuration parameters, set the relevant configuration parameters to AUTOMATIC using the UPDATE DATABASE CONFIGURATION command, the SQLFUPD API, or through the **Change Database Configuration Parameter** window in the Control Center.
3. To enable self tuning of buffer pools, set the buffer pool size to AUTOMATIC. You can do this using the ALTER BUFFER POOL statement for existing buffer pools or the CREATE BUFFER POOL statement for new buffer pools. If the size of a buffer pool is set to AUTOMATIC in the DPF environment, that buffer pool should not have any entries defined in sysibm.sysbufferpoolnodes.

Note:

1. Because self tuning redistributes memory between different memory areas, there must be at least two memory areas enabled for self tuning to occur, for example the lock memory area and the database shared memory area. The only exception to this is the memory controlled by the *sortheap* configuration parameter. When *sortheap* alone is set to AUTOMATIC, self tuning of *sortheap* is enabled.
2. In order to enable the *locklist* configuration parameter for self tuning, *maxlocks* must also be enabled for self tuning, therefore *maxlocks* is set to AUTOMATIC when *locklist* is set to AUTOMATIC. In order to enable the *sheapthres_shr* configuration parameter for self tuning, *sortheap* must also be enabled for self tuning, therefore *sortheap* is set to AUTOMATIC when *sheapthres_shr* is set to AUTOMATIC.
3. Automatic tuning of *sheapthres_shr* or *sortheap* is allowed only when the database manager configuration parameter *sheapthres* is set to 0.
4. Self tuning memory runs only on the HADR primary server. When self tuning memory is activated on an HADR system, it will never run on the secondary server and it will run on the primary server only if the configuration is set properly. If a command is run that switches the HADR database roles, self tuning memory operations will also switch so that they run on the new primary server.

Disabling self tuning memory

Self tuning can be disabled for the entire database by setting *self_tuning_mem* to OFF. When *self_tuning_mem* is set to OFF, the memory configuration parameters and buffer pools that are set to AUTOMATIC remain AUTOMATIC and the memory areas remain at their current size.

You can set *self_tuning_mem* to OFF using the UPDATE DATABASE CONFIGURATION command, the SQLFUPD API, or through the **Change Database Configuration Parameter** window in the Control Center.

Self tuning can also be effectively deactivated for the entire database if only a single memory consumer is enabled for self tuning. This is because memory cannot be redistributed when only one memory area is enabled.

For example, to disable self tuning of the *sortheap* configuration parameter, you could enter the following:

```
UPDATE DATABASE CONFIGURATION USING SORTHEAP MANUAL
```

To disable self tuning of the *sortheap* configuration parameter and change the current value of *sortheap* to 2000 at the same time, enter the following:

```
UPDATE DATABASE CONFIGURATION USING SORTHEAP 2000
```

In some cases, one memory configuration parameter can only be enabled for self tuning if another related memory configuration parameter is also enabled. For example, self tuning of the *maxlocks* configuration parameter is only permitted when the *locklist* configuration parameter is also enabled. Similarly, self tuning of the *sheapthres_shr* configuration parameter can only be enabled if self tuning of the *sortheap* configuration parameter is also enabled. This means that disabling self tuning of the *locklist* or *sortheap* parameters disables self tuning of the *maxlocks* or *sheapthres_shr* parameters, respectively.

Self tuning can be disabled for a buffer pool by setting the buffer pool to a specific size. For example, the following statement will disable self tuning for `bufferpool1`:

```
ALTER BUFFERPOOL bufferpool1 SIZE 1000
```

Determining which memory consumers are enabled for self tuning

To view the self tuning settings for memory consumers controlled by configuration parameters, use one of the following methods.

- To view the self tuning settings for configuration parameters from the command line, 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)
```

- You can also use the db2CfgGet API to determine whether or not tuning is enabled. The following values are returned:

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

  `SQLF_ON_ACTIVE` describes a situation where self tuning is enabled and active while `SQLF_ON_INACTIVE` indicates that self tuning is enabled but is currently inactive.

- You can also view the configuration settings in the **Database Configuration** window in the Control Center.

To view the self tuning settings for buffer pools, use one of the following methods.

- To retrieve the list of buffer pools that are enabled for self tuning from the command line, enter:

  ```
  db2 "select BPNAME, NPAGES from sysibm.sysbufferpools"
  ```

  When self tuning is enabled for a buffer pool, the NPAGES field in the sysibm.sysbufferpools table for that particular buffer pool will be set to -2. When self tuning is disabled, the NPAGES field will be set to the buffer pool's current size.
- To determine the current size of buffer pools that have been enabled for self tuning, use the snapshot monitor as follows and examine the current size of the buffer pool (the value of the bp_cur_buffsz monitor element):

  ```
  db2 get snapshot for bufferpools on db_name
  ```
- To view the self tuning settings of your buffer pools using the Control Center, right-click on a buffer pool and view the attributes of the buffer pools in the object details pane.

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 therefore the performance benefits of trading buffer pool memory for sort area memory may not be immediately realized.

Self tuning memory operational details and limitations

Determining tuning requirements

In order to ensure a fair and relevant comparison between memory consumers, a new common metric has been developed. Each tuned memory consumer calculates the predicted benefit from additional memory, and reports this to the self tuning memory process. Self tuning memory uses these figures as the basis for memory tuning, taking memory from consumers with the least need and reallocating it to those memory areas that will benefit the most.

Frequency of memory tuning

When enabled, self tuning memory will periodically check the variability of database workload. If the workload is not constant (that is, if the queries being run do not exhibit similar memory characteristics), the memory tuner will reallocate memory less frequently - up to 10 minutes between tuning cycles - to achieve more stable trend prediction. For workloads with more constant memory profiles, the memory tuner will tune memory more frequently - as little as 30 seconds between tuning cycles - in order to converge more quickly.

Tracking the progress of self tuning memory

Your current memory configuration can be obtained using the GET DATABASE CONFIGURATION command, or using a snapshot. Changes made by self tuning are recorded in the memory tuning log files in the stmmlog directory. The memory tuning log files contain summaries of the resource demands for each memory consumer at each tuning interval. These intervals can be determined based on the timestamps in the log entries.

Expected time to converge on best configuration

Leaving this feature enabled should result in quick tuning of parameters to optimize memory usage. A system can be tuned from an initial configuration in as

little as one hour. In most cases, tuning will usually be complete in at most 10 hours. This worst case occurs when queries run against the database exhibit markedly different memory characteristics.

Limitations of self tuning memory

In cases where low amounts of memory are available (for example, because the value of *database_memory* is set very low, or because multiple databases, instances or other applications are running on the server) performance benefits of self tuning memory will be limited.

Because self tuning memory bases tuning decisions on database workload, workloads with changing memory characteristics limit the ability of self tuning memory to tune effectively. If your workload's memory characteristics are constantly changing, self tuning memory will tune memory less frequently, and will repeatedly tune towards shifting target conditions. In this case, self tuning memory will not achieve absolute convergence, but will instead try to maintain a memory configuration that is tuned to the current workload.

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 in 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. Once tuning decisions are made on the tuning partition, the memory adjustments are distributed to all other database partitions to ensure that all database partitions maintain similar configurations.

The single tuning partition model necessitates that the feature be used only on database partitions that have similar memory requirements. The following are guidelines to use when determining whether to enable self tuning memory on your partitioned database.

Cases where self tuning is recommended in partitioned databases

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 on identical hardware, including an even distribution of multiple logical nodes to multiple physical nodes
- Perfect or near-perfect distribution of data
- Workload running on the database partitions is distributed evenly across database partitions. This means that no one database partition has elevated memory requirements for one or more heaps.

In such an environment, it is desirable to have all database partitions configured equally, and self tuning memory will properly configure the system.

Cases where self tuning is recommended in partitioned databases with care

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 system might have a set of database partitions for data, and a much smaller set of coordinator partitions and a catalog partitions. In such environments, it might be beneficial to configure the coordinator partitions and catalog partitions differently than the database partitions that contain your data.

In this environment, it is still possible to benefit from the self tuning memory feature with some minor setup. Since the database partitions containing the data comprise the bulk of the database partitions, self tuning should be enabled on all of these database partitions and one of these database partitions should be specified as the tuning partition. Additionally, since the catalog and coordinator partitions might be configured differently, self tuning memory should be disabled on these partitions. To disable self tuning on the catalog and coordinator partitions, update the *self_tuning_mem* database configuration parameter on these partitions to OFF.

Cases where self tuning is not recommended in partitioned databases

In environments where the memory requirements of each database partition are different or when different database partitions are running on dramatically different hardware, it is advisable to disable the self tuning memory feature. This can be done by setting the *self_tuning_mem* database configuration parameter to OFF on all partitions.

Comparing memory requirements of different database partitions

The best way to determine if 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 partitions (differing by no more than 20%), then the partitions can be considered similar.

Collect the following data by issuing the command `get snapshot for database on <dbname>`.

```
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

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
```

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 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 based on a number of characteristics, such as the number of database partitions in the partition group and the number of buffer pools defined.

- To determine which database partition is currently specified as the tuning partition, use the following ADMIN_CMD:

  ```
  CALL SYSPROC.ADMIN_CMD( 'get stmm tuning dbpartitionnum' )
  ```

- To change the tuning partition, use the following ADMIN_CMD:

  ```
  CALL SYSPROC.ADMIN_CMD( 'update stmm tuning dbpartitionnum <db_partition_num>')
  ```

  When you issue this command, the tuning partition will be updated asynchronously or at the next database startup.

- To have the memory tuner automatically re-select the tuning partition, enter -1 for the <db_partition_num> value.

Starting the memory tuner on DPF systems

The memory tuner will only be started in a DPF environment if the database is activated by an explicit ACTIVATE DATABASE command because self-tuning requires all partitions to be active before it can properly tune the memory on a multi-partition system.

Disabling self-tuning for a given database partition

- To disable self-tuning for a subset of database partitions, set the *self_tuning_mem* configuration parameter to OFF for the database partitions you want to leave untuned.
-

  To disable self-tuning for a subset of the memory consumers controlled by configuration parameters on a particular database partition, set the value of the relevant configuration parameter or buffer pool size to MANUAL or a specific value on that database partition. However, it is recommended that self-tuning configuration parameter values be consistent across all running partitions.

- To disable tuning for a particular buffer pool on a database partition, issue an ALTER BUFFER POOL command specifying a size value and a value for the PARTITIONNUM parameter for the partition where self-tuning is to be disabled.

An ALTER BUFFERPOOL statement that specifies the size on a particular database partition using the PARTITIONNUM clause will create an exception entry for the given buffer pool in the SYSCAT.SYSBUFFERPOOLNODES catalog, or update the exception entry if one already exists. When an exception entry exists for a buffer pool in this catalog, that buffer pool will not participate in self-tuning when the default buffer pool size is set to AUTOMATIC. To remove an exception entry so that a buffer pool can be re-enabled for self-tuning:

1. Disable 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 with the PARTITIONNUM clause specified to set the size of the buffer pool on this database partition to the default buffer pool size.
3. Enable self-tuning by issuing another ALTER BUFFERPOOL statement setting the size to AUTOMATIC.

Enabling self-tuning memory in non-uniform environments

Ideally, your data should be distributed evenly across all of your database partitions and the workload 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 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 configuration in the remaining partitions should be configured manually.

Buffer pool management

A buffer pool provides working memory and cache for database pages. The buffer pool improves database system performance by allowing data to be accessed from memory instead of from disk. Because memory access is much faster than disk access, the less often the database manager needs to read from or write to a disk, the better the database will perform. 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 row of a table, the database manager looks for the page containing that row in the buffer pool. If the page does not exist in the buffer pool, the database manager will read the page from disk and place it in the buffer pool. Once the page is in the buffer pool, the data can be used by the database manager to process the query.

Memory is allocated for the buffer pool when a database is activated. The first application to connect may cause an implicit database activation. Buffer pools can also be created, dropped, and re-sized while the database is manager is running. The ALTER BUFFERPOOL statement can be used to increase the size of the buffer pool. By default, the IMMEDIATE keyword is specified for the ALTER BUFFERPOOL statement and memory is allocated as soon as you enter the command if the memory is available. If the memory is unavailable, the statement

will execute as DEFERRED and the memory will be allocated when the database is reactivated. If you decrease the size of the buffer pool, memory is deallocated when the transaction commits. The buffer pool memory is freed when the database is deactivated. The database may implicitly deactivate when the last application exits depending on how the database was activated.

Note: To reduce the necessity of increasing the size of the *dbheap* database configuration parameter when buffer pool sizes increase, nearly all buffer pool memory comes out of the database shared memory set and is sized automatically.

To ensure that an appropriate buffer pool is available in all circumstances, DB2 creates small system buffer pools, one with each page size: 4K, 8K, 16K, and 32K. The size of each buffer pool is 16 pages. These buffer pools are hidden from the user. They are not present in the system catalogs 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 the specified buffer pool is not started because it was created using the DEFERRED keyword, or because a buffer pool of the required page size is inactive due to insufficient memory being 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 the ordinary buffer pools cannot be brought up during a database connect

  This problem is likely to have a serious cause, such as out-of-memory condition. Although DB2 will be fully functional because of the system buffer pools, performance will degrade drastically. You should address this problem immediately. You 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 specified 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 the buffer pool after you create it. You must create a new buffer pool with a different page size.

The memory tracker, invoked by db2mtrk, allows you to view the amount of database memory allocated to the buffer pools. The following sample db2mtrk output shows one user-created buffer pool, identified by the number ″1″ in parentheses, and four system buffer pools, identified by the page size in parentheses:

```
> db2mtrk -d
Tracking Memory on: 2005/10/24 at 12:47:26

Memory for database: XMLDB

   utilh     pckcacheh catcacheh bph (1)   bph (S32K)  bph (S16K)  bph (S8K)
   64.0K     576.0K    64.0K     4.2M      576.0K      320.0K      192.0K

   bph (S4K)  shsorth   lockh     dbh       other
   128.0K     0         640.0K    4.2M      192.0K
```

Buffer pool management of data pages

Pages in the buffer pool can be either *in-use* or not, and they can be *dirty* or *clean*:

- In-use pages are pages that are currently being read or updated. If a page is being updated, it cannot be accessed by anyone other than 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 has not yet been written to disk.

Pages remain in the buffer pool until the database is shut down, until 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 to bring in another page:

- How recently the page was referenced
- The probability that the page will be referenced again
- The type of data on the page
- Whether the page was changed in memory but not written out to disk (Changed pages are always written to disk before being overwritten.)

Changed pages that are written 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 in update-intensive workloads, you might want to enable proactive page cleaning. Performance can improve because 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.

For more information see: "Proactive page cleaning" on page 71

Illustration of buffer pool data-page management

The following figure illustrates how the work of managing the buffer pool can be shared between page-cleaner agents and database agents, compared to the database agents performing all of the I/O.

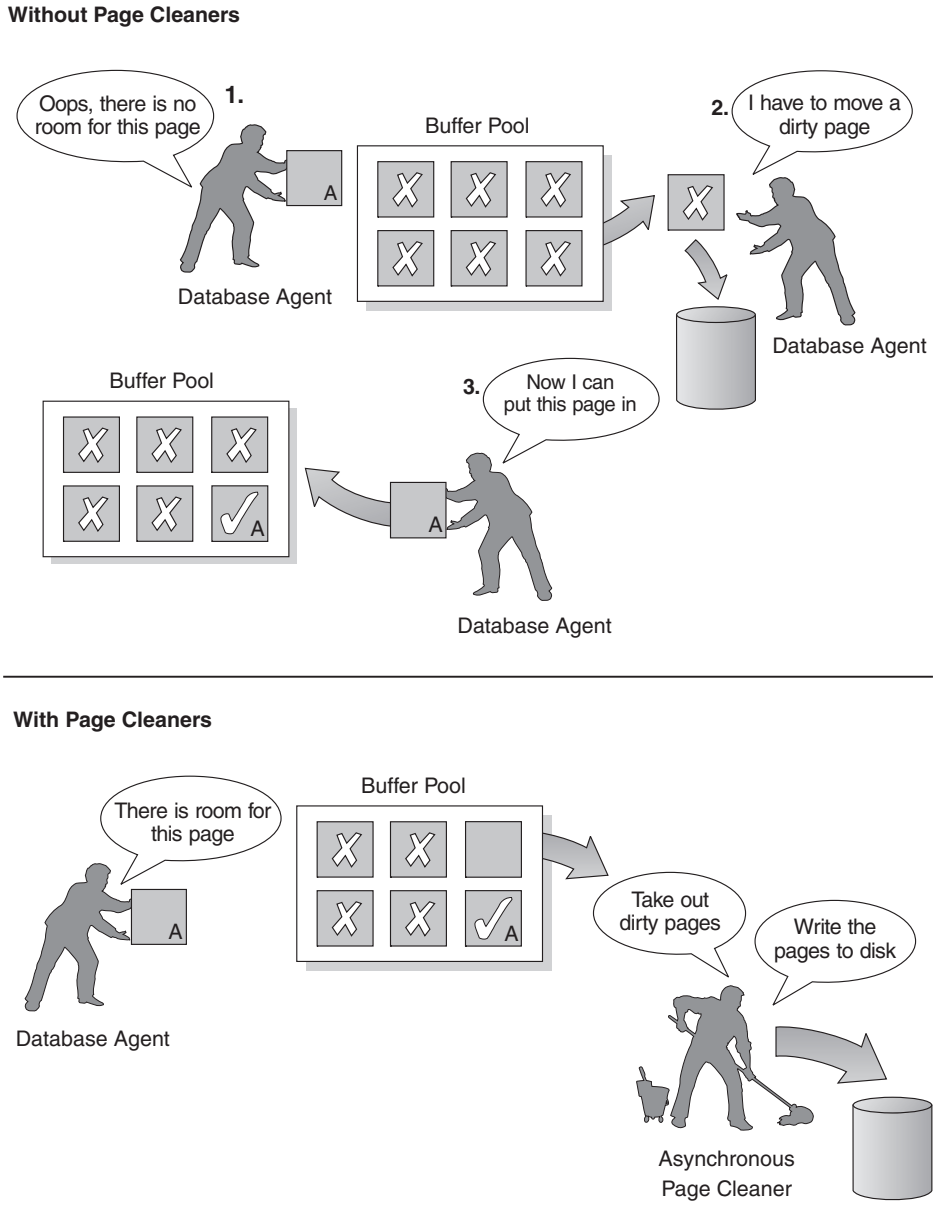


Figure 14. Asynchronous page cleaner. "Dirty" pages are written out to disk.

Page cleaning and fast recovery

If more pages have been written to disk, recovery of the database is faster after a system crash because the database manager can rebuild more of the buffer pool from disk instead of having to replay 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.

The page cleaners write dirty pages to disk such that the size of the log that would need to be replayed during recovery never exceeds the following:

```
logfilsiz * softmax / 100 (in 4K pages)
```

where:

- *logfilsiz* represents the size of the log files
- *softmax* represents the percentage of log files to be recovered following a database crash. For example, if the value of *softmax* is 250, then 2.5 log files will contain the changes that need to be recovered if a crash occurs.

To minimize log read time during recovery, use the database system monitor to track the number of times that page cleaning is performed. The system monitor *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 this alternate page cleaning, this condition should not occur and the *pool_lsn_gap_clns* monitor element is always 0.

The *log_held_by_dirty_pages* monitor element can be used to determine if 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. Each buffer pool has a minimum size, depending on the platform. You can use the ALTER BUFFERPOOL command to resize a buffer pool.

A new database has a default buffer pool called IBMDEFAULTBP with an overall size determined by the platform and a default page size based on the page size specified when the database was created. 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 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 migrate a database, you can create additional buffer pools. You can create the database with 8 KB page size as the default and the default buffer pool will be created with the default page size (in this case, an 8 KB page size). 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, then DB2 assigns the table space to that buffer pool.

When you create a buffer pool with the CREATE BUFFERPOOL statement, you can specify a specific buffer pool size. If you do not specify a size, it will be set to AUTOMATIC and managed by DB2. To change the buffer-pool 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 otherwise specified in the CREATE BUFFERPOOL statement, or the buffer-pool size was changed by the ALTER BUFFERPOOL statement for a particular database partition.

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 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 the temporary table spaces.

Advantages of many buffer pools

If any of the following conditions apply to your system, you should use only a single buffer pool:

- The total buffer space is less than 10 000 4 KB pages.
- People with the application knowledge to do specialized tuning are not available.
- You are working on a test system.

In all other circumstances, consider using more than one buffer pool for the following reasons:

- Temporary table spaces can be assigned to a separate buffer pool to provide better performance for queries that require temporary storage, especially sort-intensive queries.
- 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. This change will reduce the impact that frequent updates on the first set of tables have on frequent queries on the second set of tables.
- You can use smaller buffer pools for the data accessed by applications that are seldom used, especially for an application that requires very random access into a very large table. In such a case, 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 data, and free the extra memory for other uses, such as for other buffer pools.

- After separating different activities and data into separate buffer pools, good and relatively inexpensive performance diagnosis data can be produced from statistics and accounting traces.

Buffer pool memory allocation at startup

When you use the CREATE BUFFERPOOL command to create a buffer pool or use the ALTER BUFFERPOOL statement to alter buffer pools, the total memory that is required by all buffer pools must be available to the database manager so all of the buffer pools can be allocated when the database is started. If you create or modify 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 makes the change the next time the database is activated.

If this memory is not available when a database starts, the database manager will only start with system buffer pools (one for each page size) with a minimal size of 16 pages, and an SQL1478W (SQLSTATE01626) warning is returned. The database continues in this operational state until its configuration is changed and the database can be fully restarted. Performance may be suboptimal. The database manager starts with minimal-sized values only to allow you to connect to the database so that you can re-configure the buffer pool sizes or perform other critical tasks. As soon as you perform these tasks, restart the database. Do not operate the database for an extended time in such a state.

To avoid starting the database with system buffer pools only, you can use the DB2_OVERRIDE_BPF registry variable to adjust the memory required so that it fits into what is available.

Proactive page cleaning

Starting in Version 8.1.4, there is an alternate method of configuring page cleaning in your system. This alternate method differs from the default behavior in that page cleaners behave more proactively in choosing which dirty pages get written out at any given point in time. This new method of page cleaning differs from the default page cleaning method in two major ways:

1. Page cleaners do not respect the chngpgs_thresh configuration parameter.

   In this alternative method of page cleaning, page cleaners will no longer react in response to value of the chngpgs_thresh configuration parameter. Instead of attempting to keep some percentage of the buffer pool clean, the alternate method of page cleaning provides a mechanism whereby the agents are informed of the location of good victim pages that have just been written out, so that agents do not have to search the buffer pool to look for a victim. When the number of good victim pages drops below an acceptable value, the page cleaners are triggered, and proceed to search the entire buffer pool, writing out potential victim pages, and informing the agents of the location of these pages.
2. Page cleaners no longer respond to LSN gap triggers issued by the logger.

   When the amount of log space encompassing the log record which has updated the oldest page in the buffer pool and the current log position exceeds that allowed by the softmax parameter, it is said that the database is in an 'LSN gap' situation. Under the default method of page cleaning, when the logger detects than an LSN gap has occurred, it will trigger the page cleaners to write

out all the pages which are contributing to the LSN gap situation. That is, it will write out those pages which are older than what is allowed by the softmax parameter. Page cleaners will be idle for some period of time while no LSN gap is occurring. Then, once an LSN gap occurs, the page cleaners are activated to write a large number of pages before going back to sleep. This can result in the saturation of the I/O subsystem, which then affects other agents which are reading or writing pages. Furthermore, by the time an LSN gap is triggered, it is possible that the page cleaners will not be able to clean fast enough and DB2 might run out of log space.

The alternate method of page cleaning modulates this behavior by spreading out the same number of writes over a greater period of time. The cleaners do this by proactively cleaning not only pages the pages that are currently in an LSN gap situation, but also the pages that will come into an LSN gap situation soon, based on the current level of activity.

To use the new method of page cleaning, set the DB2_USE_ALTERNATE_PAGE_CLEANING registry variable to ″ON″.

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 improve performance by reducing the I/O wait time. In addition, parallel I/O enhances prefetching efficiency.

There are two categories of prefetching:

- **Sequential prefetch:** A mechanism that reads consecutive pages into the buffer pool before the pages are required by the application.
- **List prefetch:** Sometimes called *list sequential prefetch*. Prefetches a set of non-consecutive data pages efficiently.

These two methods of reading data pages are in addition to a database manager agent read. A database manager agent read is used when only one or a few consecutive pages are retrieved, but only one page of data is transferred.

Prefetching and Intra-Partition Parallelism

Prefetching is important to the performance of intra-partition parallelism, which uses multiple subagents when scanning an index or a table. Such parallel scans introduce larger data-consumption rates, which require higher prefetch rates.

The cost of inadequate prefetching is higher for parallel scans than serial scans. If prefetching does not occur for a serial scan, the query runs more slowly because the agent always needs to wait for I/O. If prefetching does not occur for a parallel scan, all subagents might need to wait because one subagent is waiting for I/O.

Because of its importance, prefetching is performed more aggressively with intra-partition parallelism. 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. In addition, multiple parallel I/O operations to read several ranges of pages into the buffer pool can help reduce I/O wait time.

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 can easily determine that sequential prefetch will improve I/O performance. In these cases, the database manager automatically starts sequential prefetch. The following example, which probably requires a table scan, would be a good candidate for sequential prefetch:

```
SELECT NAME FROM EMPLOYEE
```

Implications of the PREFETCHSIZE for table spaces

To define the number of prefetched pages for each table space, use the PREFETCHSIZE clause in either the CREATE TABLESPACE or ALTER TABLESPACE statements. The value that you specify is maintained in the PREFETCHSIZE column of the SYSCAT.TABLESPACES system catalog table.

It is a 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 for your table space, which is the number of pages that the database manager writes to a container before it uses a different container. For example, if the extent size is 16 pages and the table space has two containers, you might set the prefetch quantity to 32 pages. If there are 5 physical disks per container, then you might set the prefetch quantity 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 less than you specify 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 you tune PREFETCHSIZE for your table spaces. You might gather information about whether:

- There are I/O waits for your query, using monitoring tools available for your operating system.
- Prefetch 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 and the query is prefetching data, you might increase the value of PREFETCHSIZE. If the prefetcher is not the cause of the I/O wait, increasing the PREFETCHSIZE value will not improve the performance of your query.

In all types of prefetch, multiple I/O operations might be performed in parallel when the prefetch size is a multiple of the extent size for the table space and the extents of the table space are in separate containers. For better performance, configure the containers to use separate physical devices.

Sequential detection

In some cases it is not immediately obvious that sequential prefetch will improve performance. In these cases, the database manager can monitor I/O and activate prefetching if sequential page reading is occurring. In this case, prefetching is activated and deactivated by the database manager as appropriate. This type of sequential prefetch is known as *sequential detection* and applies to both index and

data pages. Use the *seqdetect* configuration parameter to control whether the database manager performs sequential detection.

For example, if sequential detection is turned on, the following SQL statement might benefit from sequential prefetch:

```
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, then the data-page reads will be almost sequential and prefetching might improve performance, so data-page prefetch will occur.

Index-page prefetch might also occur in this example. If many index pages must be examined and the database manager detects that sequential page reading of the index pages is occurring, then index-page prefetching occurs.

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 is overlapped with I/O. Most platforms provide high-performance primitives that read contiguous pages from disk into non-contiguous 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, the buffer pools are page-based, which means that contiguous pages on disk are prefetched into non-contiguous 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 consist of both a page area and a block area. The page area is required for non-sequential prefetching workloads. The block area consist of blocks where each block contains a specified number of contiguous pages, which is referred to as the *block size*.

The optimal usage 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 interact for efficient use of buffer-pool memory. Buffer-pool memory can be wasted in the following circumstances:

- If the extent size, which determines the prefetch request size, is smaller than BLOCK_SIZE specified for the buffer pool.
- If some pages requested 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 much of a block would be wasted, the I/O server does non-block-based prefetching into the page area of the buffer pool. This is not optimal performance.

For optimal performance, bind table spaces of the same extent size to a buffer pool with a block size that equals the table-space extent size. Good performance can be achieved if the extent size is larger than the block size, but not when the extent size is smaller than the block size.

To create block-based buffer pools, use the CREATE and ALTER BUFFERPOOL statements.

Note: Block-based buffer pools are intended for sequential prefetching. If your applications do not use sequential prefetching, then the block area of the buffer pool is wasted.

List prefetching

List prefetch, or *list sequential prefetch*, is a way to access data pages efficiently even when the data pages needed are not contiguous. List prefetch 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 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, given the previously defined index IX1:

```
INDEX IX1:  NAME    ASC,
            DEPT    ASC,
            MGR     DESC,
            SALARY  DESC,
            YEARS   ASC
```

and the following search criteria:

```
WHERE NAME BETWEEN 'A' and 'I'
```

If the data is not clustered according to this index, list prefetch includes a step that sorts the list of RIDs obtained from the index scan.

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, the 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 processing activity. These prefetch requests contain a description of the I/O needed to satisfy the query. The possible prefetch methods determine when and how the database manager generates the prefetch requests.

Configuring enough I/O servers with the *num_ioservers* configuration parameter can greatly enhance the performance of queries for which prefetching of data can be used. 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 the number of I/O servers than to underestimate. If you specify extra I/O servers, these servers are not used, and their memory pages are paged out. As a result, performance does not suffer. Each I/O server process is numbered. The database manager always uses the lowest numbered process, so some of the upper 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.

It is worthwhile to consider setting the value of the *num_ioservers* database configuration parameter to AUTOMATIC so that DB2 can pick intelligent values based on the system configuration.

Configuration for asynchronous I/O

On some platforms, DB2 uses asynchronous I/O (AIO) to improve performance of activities such as page cleaning and prefetching. AIO is most effective if data in containers is distributed across multiple disks. Performance also benefits from tuning the underlying operating system AIO infrastructure.

For example, on AIX®, you might tune AIO on the operating system. When AIO works on either SMS or DMS file containers, operating system processes called AIO servers manage the I/O. A small number of such servers might restrict the benefit of AIO by limiting the number of AIO requests. To configure the number of AIO servers on AIX, use the *smit* AIO *minservers* and *maxservers* parameters.

Illustration of prefetching with parallel I/O: The following figure illustrates how I/O servers are used to prefetch data into a buffer pool.

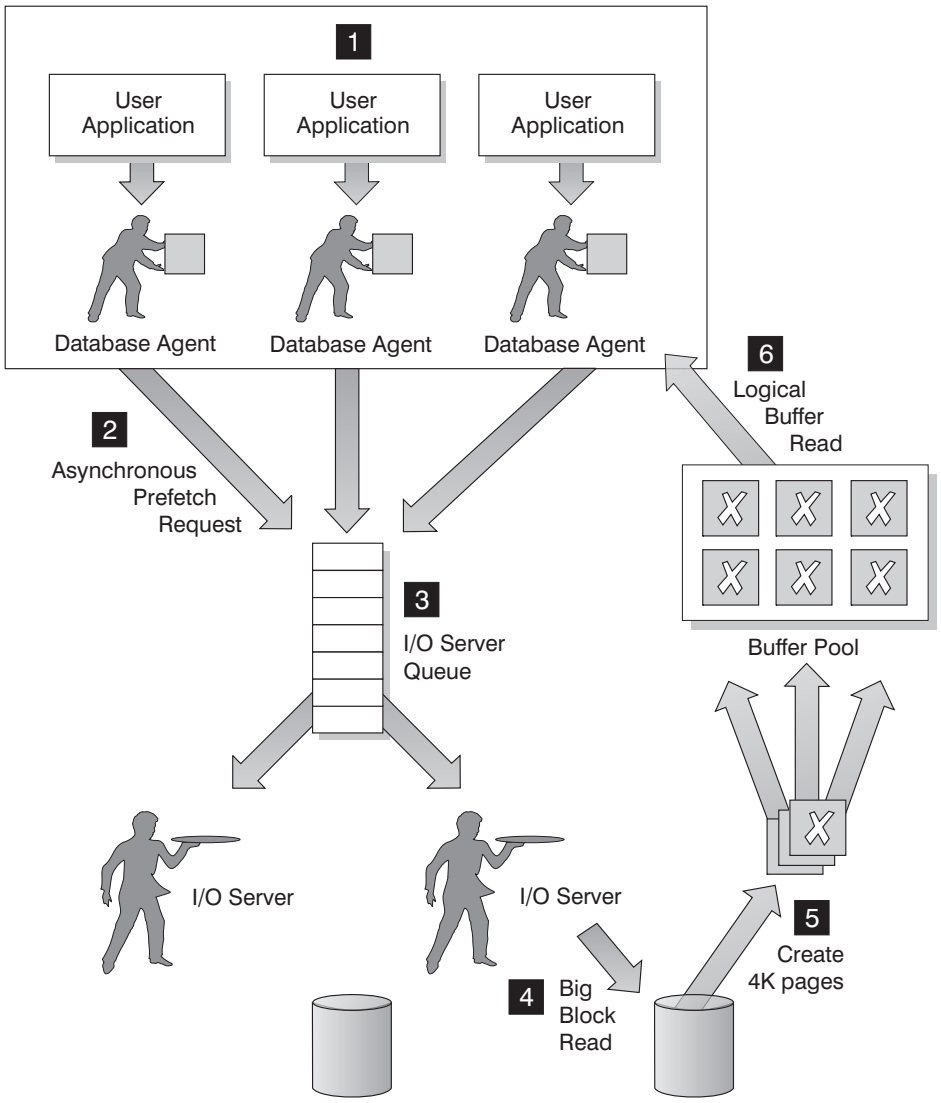


Figure 15. Prefetching data using I/O servers

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 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 fetch data from a table space at the same time depends on the number of prefetch requests in the queue and the number of I/O servers configured 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*, in which 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. Performing I/O in parallel can result in significant improvements to 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 I/O in parallel 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:

- **Sequential prefetch**

  For sequential prefetch, parallel I/O is initiated when the prefetch size is a multiple of the extent size for a table space. Each prefetch request is then broken into many small requests along the extent boundaries. These small requests are then assigned to different I/O servers.
- **List prefetch**

  For list prefetch, each list of pages is divided into smaller lists according to the container in which the data pages are stored. These smaller lists are then assigned to different I/O servers.
- **Database or table space backup and restore**

  For backing up or restoring data, the number of parallel I/O requests are equal to the backup buffer size divided by the extent size up to a maximum value equal to the number of containers.
- **Database or table space restore**

  For restoring data, the parallel I/O requests are initiated and split the same way as that used for sequential prefetch. Instead of restoring the data into the buffer pool, the data is moved directly from the restore buffer to disk.
- **Load**

  When you load data, you can specify the level of I/O parallelism with the LOAD command DISK_PARALLELISM option. If you do not specify this option, the database manager uses a default value based on the cumulative number of table space containers for all table spaces associated with the table.

For optimal performance of parallel I/O, ensure that:

- There are enough I/O servers. Specify slightly more I/O servers than the number of containers used for all table spaces within the database.
- The extent size and prefetch size are sensible for the table space. To prevent over-use of the buffer pool, 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 prefetch. 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, it is the only container to store additional data. The database manager cannot use parallel prefetch to access this additional data.
- There is adequate I/O capacity when using intra-partition parallelism. On SMP machines, intra-partition parallelism can reduce the elapsed time for query by running the query on multiple processors. Sufficient I/O capacity is required to keep each processor busy. Usually additional physical drives are required to provide the I/O capacity.

  The prefetch size must be larger for prefetching to occur at higher rates and 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.

Tuning sort performance: Because queries often require sorted or grouped results, sorting is often required, and the proper configuration of the sort heap areas is crucial to good query performance. Sorting is required when:

- No index exists to satisfy a requested ordering (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:

- The settings for the following database 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 for sorting available across the entire instance for all sorts
- The number of statements in a workload which require a large amount of sorting.
- The existence or absence of indexes that could help avoid unnecessary sorting
- The use of application logic that does not minimize sorting
- Parallel sorting, which improves the performance of sorts but can only occur if the statement uses intra-partition parallelism.
- Whether the sort is *overflowed* or *non-overflowed*. If the sorted data cannot fit entirely into the sort heap, which is a block of memory that is allocated each timea sort is performed, it overflows into a temporary table owned by the database. Sorts that do not overflow always perform better than those that do.
- Whether the results of the sort are *piped* or *non-piped*. If sorted information can return directly without requiring a temporary table to store a final, sorted list of

data, it is a piped sort. If the sorted information requires a temporary table to be returned, it is a non-piped sort. A piped sort always performs better than a non-piped sort.

Also note that in a piped sort, the sort heap is not freed until the application closes the cursor associated with that sort. A piped sort can continue to use up memory until the cursor is closed.

In general, overall sort memory available across the instance (*sheapthres*) should be as large as possible without causing excessive paging. 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 sorting 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 the queries that have sort operations.
3. Use these statements as input to the Design Advisor, which will identify and optionally create indexes to reduce the need for sorting.

Use the database system monitor and benchmarking techniques to help set the *sortheap* 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: With the improvement in the DB2 partial-key binary sorting technique to include non-integer data type keys, some additional memory is required when sorting long keys. If long keys are used for sorts, you may need to increase 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 above.

      This is a recommended initial setting. You can then use benchmark techniques to refine this value.

You can also use the self tuning memory feature to automatically and dynamically allocate and de-allocate 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".

Maintaining the organization of your tables and indexes

Over time, data in your tables can become fragmented, increasing the size of your tables and indexes as the 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 your tables and indexes compacts your data, reclaiming wasted space and improving data access.

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 reorganization on identified objects.
4. Monitor progress of reorganization
5. For online table reorganization, you may pause the reorganization process if necessary, which will allow you to then resume at a later time.
6. Evaluate the outcome of the reorganization, determining whether the operation succeeded or failed. For offline table reorganization and any index reorganization the operation is synchronous and the outcome of the reorganization will be apparent upon return of the command. Online table reorganization is processed asynchronously, therefore, to evaluate the outcome you will need to refer to the history file.
7. If you performed an online table reorganization, you can choose to recover it. See: "Recovery of a failed online table reorganization" on page 89
8. Collect statistics on reorganized objects.
9. Rebind applications that access reorganized objects.

Table reorganization

After many changes to table data, logically sequential data may be on non-sequential physical data pages so that the database manager must perform additional read operations to access data. Additional read operations are also required if a significant number of rows have been deleted. In such a case, you might consider reorganizing the table to match the index and to reclaim space. You can reorganize the system catalog tables as well as database tables.

Note: Because reorganizing a table usually takes more time than running statistics, you might execute RUNSTATS to refresh the current statistics for your data and rebind your applications. If refreshed statistics do not improve performance, reorganization might help. For detailed information about the options and behavior of the REORG TABLE utility, refer to its command reference.

Consider the following factors, which might indicate that you should reorganize a table:

- A high volume of insert, update, and delete activity on tables accessed by queries

- Significant changes in the performance of queries that use an index with a high cluster ratio
- Executing RUNSTATS to refresh statistical information does not improve performance
- The REORGCHK command indicates a need to reorganize your table
- The tradeoff between the cost of increasing degradation of query performance and the cost of reorganizing your table, which includes the CPU time, the elapsed time, and the reduced concurrency resulting from the REORG utility locking the table until the reorganization is complete.

Reducing the need to reorganize tables

To reduce the need for reorganizing a table, perform these tasks after you create the table:

- Alter table to add PCTFREE
- Create clustering index with PCTFREE on index
- Sort the data
- Load the data

After you have performed these tasks, the table with its clustering index and the setting of PCTFREE on table helps preserve the original sorted order. If enough space is allowed in table pages, new data can be inserted on the correct pages to maintain the clustering characteristics of the index. As more data is inserted and the pages of the table become full, records are appended to the end of the table so that the table gradually becomes unclustered.

If you perform a REORG TABLE or a sort and LOAD after you create a clustering index, the index attempts to maintain a particular order of data, which improves the CLUSTERRATIO or CLUSTERFACTOR statistics collected by the RUNSTATS utility.

Note: Creating multi-dimensional clustering (MDC) tables might reduce the need to reorganize tables. For MDC tables, clustering is maintained on the columns that you specify as arguments to the ORGANIZE BY DIMENSIONS clause of the CREATE TABLE statement. However, REORGCHK might recommend reorganization of an MDC table if it considers that there are too many unused blocks or that blocks should be compacted.

Choosing a table reorganization method

There are two different methods of table reorganization: *classic* or offline REORG and *inplace* or online REORG.

The INPLACE option of the REORG command specifies an online reorganization. If this is not specified, an offline REORG is run.

Each method of reorganizing has advantages and trade-offs. These are summarized below. When choosing a reorganization method, consider which method offers advantages that align with your priorities, for example if recoverability in case of failure is more important than having the reorganization complete quickly, the online reorganization method might be the best approach to take.

Advantages of offline reorganization

- Provides fastest table reorganization, especially if reorganization of LOB/LONG data is not required.

- Tables and indexes are perfectly clustered upon completion.
- Indexes are rebuilt once the table has been reorganized. No separate step is required to rebuild indexes.
- Shadow copy can be built in temporary table space. This reduces the amount of space required in table space containing target table or index.
- Permits specification and use of an index other than the clustering index to re-cluster the data, while online reorganization must use the existing clustering index if one exists.

Disadvantages of offline reorganization

- Limited table access. Applications allowed only read access to the table and only during the sort and build phase of the reorganization.
- Large space requirement since shadow copy approach is used.
- Less control over the REORG process than with online REORG: offline reorganization cannot be paused and restarted.

Advantages of online reorganization

- Allows applications full access to the table during REORG except during truncate phase.
- You have more control over the REORG process: process running asynchronously in background and can be paused, resumed, and stopped. For example, if a large number of updates or deletes are being run against the table, you can pause the REORG process.
- Recoverable process if failures occur.
- Requires less working storage since table is incrementally processed.
- Benefits of reorganization can be realized immediately as REORG progresses.

Disadvantages of online reorganization

- Can result in imperfect data clustering or imperfect index clustering, depending on the type of transactions accessing the table during the REORG.
- Pages reorganized in the beginning of the reorganization might have more updates, and therefore be more fragmented than tables that are reorganized later in the process.
- Slower performance than offline reorg. For a normal clustering REORG (not just space reclaiming) it might take 10-20 times longer to perform an online REORG. (A table with applications running against it concurrently, or with a large number of indexes defined could take significantly longer.)
- Online REORG is a recoverable process, but this comes at the expense of increased logging requirements. It is possible that a significant amount of log space is required (up to several times the size of the table). This is dependent on the number of rows moved during the REORG, the number of indexes defined on the table, and the size of the indexes.
- Indexes are maintained, not rebuilt, so subsequent index reorganization might be required.

Table 2. Comparison of online and offline reorganization

Characteristic	Offline Reorganization	Online reorganization
Performance	Fast	Slow
Clustering factor of data at completion	Good	Not perfectly clustered

Table 2. Comparison of online and offline reorganization (continued)

Characteristic	Offline Reorganization	Online reorganization
Concurrency (access to the table)	Ranges from NO ACCESS to READ ONLY	Ranges from READ ONLY to full access
Data storage space requirement	Significant	Not significant
Logging storage space requirement	Not significant	Could be significant
User control (ability to pause, restart process)	Less control	More control
Recoverability	All or nothing: succeeds or fails.	Recoverable
Index rebuild	Done	Not done
Supported for all types of tables	Yes	No
Specify index other than clustering index	Yes	No
Use of temporary table space	Yes	No

Table 3. Table types supported for online and offline reorganization

Table type	Offline reorganization supported	Online reorganization supported
Multi-dimensional clustering tables (MDC)	Yes[1]	No
Range-clustered tables (RCT)	No[2]	No
Append mode tables	No	No[3]
Tables with LONG/LOB data	Yes[4]	No
Tables with type-1 indexes	Yes[5]	No
System catalog tables: SYSIBM.SYSTABLES, SYSIBM.SYSSEQUENCES, SYSIBM.SYSDBAUTH, SYSIBM.SYSROUTINEAUTH	Yes	No

1. Since clustering is automatically maintained via MDC block indexes, reorganization of an MDC table involves space reclamation only. No index can be specified since block index is used.
2. The range area of an RCT always remains clustered.
3. After append mode is turned off, online reorganization can be performed.
4. Reorganizing LONG/LOB data can take a significant amount of time. Reorganizing LONG/LOB data does not help query performance; it should only be done for space reclamation.
5. After reorganization, all indexes are rebuilt as type-2 indexes.

Refer to the REORG TABLE syntax descriptions for detailed information about executing these table reorganization methods.

Monitoring the progress of table reorganization

Information about the current progress of table reorganization is written to the history file for database activity. The history file contains a record for each reorganization event. To view this file, execute the `db2 list history` command for the database that contains the table you are reorganizing.

You can also use table snapshots to monitor the progress of table reorganization. Table reorganization monitoring data is recorded regardless of the Database Monitor Table Switch setting.

If an error occurs, an SQLCA dump is written to the history file. For an inplace table reorganization, the status is recorded as PAUSED.

Offline table reorganization

Offline 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:

1. *SORT*
   If an index is specified with the REORG TABLE command, or if a clustering index is 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 only applies to a clustering REORG. Space reclaiming reorganizations begin at the build phase.
2. *BUILD*
   In this phase, a reorganized copy of the entire table is build, either in the table space that the table being reorganized resides, or in a temporary table space specified with the REORG command.
3. *REPLACE*
   In this phase, the original table object is replaced by either copying back from the temporary table space, or by pointing to the newly built object within the table space of the table being reorganized.
4. *RECREATE ALL INDEXES*
   All indexes defined on the table are recreated.

You can monitor the progress of the REORG TABLE and determine which phase the reorganization is currently in using the snapshot monitor or snapshot administrative views.

The locking conditions are more restrictive in offline table REORG than in online REORG. Read access is available on the table while the copy is being built. However, exclusive access to the table is required while replacing the original table with the reorganized copy and while rebuilding the indexes.

An IX table space lock is required during entire 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. However, no other application can update any data. (Read access is allowed.) The U lock is upgraded to a Z lock once the REPLACE phase is started. During this phase, no other applications can access the data. This lock is held until REORG completes.

A number of files are created by the offline reorganization process. These files are stored in your database directory and are prefixed with the table space and object ids, for example, 0030002.ROR is the state file for the reorganization of a table with a table space id of 3 and a table id of 2.

The following are temporary files created during offline REORG for an SMS table:

- .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 MDC tables)

The following temporary file is created during an index reorganization:

- .IN1 Shadow copy file

The following temporary files 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 associated with the reorganization process should 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 required for the table and improves data access and query performance.

You must have SYSADM, SYSCTRL, SYSMAINT, or DBADM authority, or you must have CONTROL privilege on the table to reorganize a table. You must have a database connection to reorganize a table.

Once you have identified the tables that require reorganization, you can run the REORG utility on those tables and, optionally, on any indexes defined on those tables.

1. To reorganize a table using the CLP, issue the REORG TABLE command:

   ```
   db2 reorg table test.employee
   ```

   To reorganize a table using the temporary table space `mytemp`, enter:

   ```
   db2 reorg table test.employee use mytemp
   ```

   To reorganize the table and have the rows reordered according to the index `myindex`, enter:

   ```
   db2 reorg table test.employee index myindex
   ```

2. To reorganize a table using an SQL call statement , issue the REORG TABLE command using the ADMIN_CMD procedure:

   ```
   call sysproc.admin_cmd ('reorg table employee index myindex')
   ```

3. To reorganize a table using a DB2 administrative API, use the db2REORG API.

After reorganizing a table, you should collect statistics on the 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 up until the REPLACE phase has begun. This means that if your system crashes during the SORT or BUILD phases, the REORG TABLE operation is rolled back and will not be redone on crash recovery. Instead, the REORG TABLE command will have to be reissued after the recovery.

If your system crashes after entering the REPLACE phase, then the REORG TABLE operation has to complete. This is because all of the REORG TABLE work has been done and the original table may no longer be available. During crash recovery, the temporary file for the reorganized object is required but not the temporary table space used for the sort. Recovery will restart the REPLACE phase from the beginning and all the data in the copy object is required for recovery. There is a difference between SMS and DMS table spaces in this case: the SMS object will have to be copied from one object to the other, while in DMS, the newly reorganized object is simply pointed to and the original table 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 normal rules to determine when they are rebuilt, either at database restart or upon first access to the indexes.

A crash in the index rebuild phase means we already have the new object so nothing is redone. (As above, indexes are not rebuilt, but are marked invalid during crash recovery, and the database will follow the normal rules to determine when they are rebuilt, either at database restart or upon first access to the indexes.)

For rollforward recovery, REORG TABLE is redone if the old version of the table is on disk. Rollforward uses the RIDs that are logged during the BUILD phase to reapply the order of operations that created the reorganized table, repeating the BUILD and REPLACE phases. Again, indexes are marked invalid as described above. This means that there is no index scan or scan sort performed, and the only temporary table space required is for the copy of the reorganized object, if a temporary table space was used originally. During rollforward, multiple REORG TABLE operations can be redone concurrently due to parallel recovery. In this case, the disk space usage will be greater than at runtime.

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 virtually no difference in performance between REORG TABLE run in NO ACCESS mode and REORG TABLE run in ALLOW READ ACCESS mode. The only difference is that for READ ACCESS mode, DB2 upgrades the lock on the table before replacing it, therefore the utility might have to wait for existing scans to complete and release their locks. The table is unavailable during the index rebuild phase of the REORG TABLE operation in both cases.

Tips for improving performance

If enough space exists in the table space, use the same table space for the original table and the reorganized copy of the table, instead of using a temporary table space. This saves the time taken to copy the 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 TABLE operation.
- Ensure the prefetch size of the table spaces where the reorganized table resides is set properly.
- Enable INTRA_PARALLEL so that the index rebuilding is done with parallel processing.
- Tune the *sortheap* and *sheapthres* database configuration parameters to control space for sorts. Since each processor will perform a private sort, the value of *sheapthres* should be at least *sortheap* x *the number of processors used*.
- Ensure dirty index pages are getting cleaned from the buffer pool as soon as possible by tuning the number of page cleaners.

Online table reorganization

Online or inplace table reorganization allows the user to reorganize a table while permitting full access to that table. While online REORG TABLE provides uninterrupted user access to the data, the performance of online REORG TABLE is slower than offline REORG TABLE.

During an online table reorganization, the entire table is not reorganized at once. Instead, portions of the table are reorganized sequentially. Data is not copied out to a temporary table space: rows are moved within the existing table object to re-establish clustering, reclaim free spaces, and eliminate overflow rows.

There are four primary phases for online REORG TABLE:

1. SELECT N pages
   During this phase, DB2 selects N pages, where N is the extent size with a minimum of 32 sequential pages for REORG TABLE processing.
2. Vacate the range
   With the N pages selected, online REORG TABLE moves all rows within this range to free pages within the table. Each row that is moved leaves behind an RP (REORG TABLE pointer) record that contains the RID of the row's new location. The row is inserted to free pages in the table as a RO (REORG TABLE overflow) record that contains the data.
   Once REORG TABLE finishes moving a set of rows, it waits for all existing data accesses that are occurring in the table (for example, by currently executing applications) to complete. These existing accesses, called old scanners, use old the RIDs when accessing the table data. Any accesses that start during this waiting period, called new scanners, use the new RIDs to access the data. Once all of the old scanners have completed, REORG TABLE cleans up the moved rows, deleting the RP records and converting the RO records to normal records.
3. Fill the range
   After all rows have been vacated, the rows are written back, in a reorganized format, sorted according to any indexes used, and obeying any PCTFREE restrictions defined. When all the pages in the range are filled, the next N sequential pages are selected in the table, and the process begins again.
4. Truncate the table
   When all pages in the table have been reorganized, the table will be truncated to reclaim space by default. If the NOTRUNCATE option is specified, the reorganized table is not truncated.

Files created during an online table REORG

During an online table reorganization, an `.OLR` state file is created for each database partition. This file is a binary file with the name `xxxxyyy.OLR`, where `xxxx` is the pool ID and `yyyy` is the object ID in hex format. This file contains information required to resume an online reorganization from a paused state.

The state file includes the following information:

- The type of REORG
- The life LSN of the table being reorganized
- The next range to be vacated
- Whether the reorganization is clustering the data or just reclaiming space
- The ID of the index used to cluster the data

A checksum is held 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 will have to be initiated, and a new state file will be created.

If the `.OLR` state file is deleted, the REORG TABLE process cannot be resumed and an SQL2219 error is returned. A new REORG TABLE process will have to be initiated.

The files associated with the reorganization process should not be manually removed from your system.

Reorganizing tables online:

An online or inplace table reorganization allows users to access the table while it is being reorganized.

You must have SYSADM, SYSCTRL, SYSMAINT, or DBADM authority, or you must have CONTROL privilege on the table to reorganize a table. You must have a database connection to reorganize a table.

You can perform an online table reorganization using a CLP command, using an SQL call statement, or through a DB2 API.

1. To reorganize a table online using the CLP, issue the REORG TABLE command using the INPLACE option :

   ```
   db2 reorg table test.employee inplace
   ```

2. To reorganize a table online using an SQL call statement , issue the REORG TABLE command using the ADMIN_CMD procedure:

   ```
   call sysproc.admin_cmd ('reorg table employee inplace')
   ```

3. To reorganize a table using a DB2 administrative API, use the db2REORG API.

After reorganizing a table, you should collect statistics on the table so that the optimizer has the most accurate data for evaluating query access plans.

Recovery of a failed 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 one or more database partitions in a partitioned database environment encounters an error, the sqlcode returned will be the one from the first node that reports an error.

If the failure is during runtime, the online table reorganization is paused and rolled back. If the system goes down, on restart, crash recovery begins and the reorganization is paused and rolled back. Later on, you can resume the reorganization by specifying the RESUME option with the REORG TABLE command. Since the online table reorganization process is fully logged, it is guaranteed that reorganization is recoverable.

Under some circumstances, for example, online table reorganization might exceed the *num_log_span* limit. In this case, DB2 will force REORG TABLE and put it into PAUSE status. In the snapshot output, the state of the REORG TABLE utility will appear as "PAUSED".

The online table reorganization pause is interrupt-driven, which means that it can be paused either by the user (using the pause option of the REORG TABLE command, or the force application command) or by DB2 in certain circumstances, for example in the case of a system crash.

Pausing and restarting an online table reorganization:

An online table reorganization that is in progress can be paused and restarted by the user.

To pause or restart an online table reorganization you must have one of the following:

- sysadm
- sysctrl
- sysmaint
- dbadm
- CONTROL privilege on the table.

1. To pause an online table reorganization, issue the REORG TABLE command using the PAUSE option:

   ```
   db2 reorg table homer.employee inplace pause
   ```

2. To restart a paused online table reorganization, issue the REORG TABLE command with the RESUME option:

   ```
   db2 reorg table homer.employee inplace resume
   ```

   Note:

   - When an online table reorganization is paused, you cannot begin a new reorganization of that table. You must either resume the paused reorganization, or stop the paused reorganization before beginning a new reorganization process.
   - When a RESUME request is issued, the reorganization process respects the TRUNCATE option specified in the original REORG command, regardless of what TRUNCATE options are specified with subsequent START or any intermediate RESUME requests. However, if a reorganization is in the TRUNCATE phase and the user issues a RESUME request specifying NOTRUNCATE, the table is not truncated and the reorganization completes.
   - A reorganization 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 is how locking is controlled, as it is crucial to application concurrency.

At a given point during an online REORG TABLE the operation may hold the following locks:

- To ensure write access to the table space, an IX lock is acquired on the table spaces affected by the REORG TABLE operation.
- A table lock is acquired and held during the entire REORG TABLE operation. The level of locking is dependant on the access mode allowed on that table during reorganization:
  If ALLOW WRITE ACCESS is specified, an IS lock on the table will be acquired
  If ALLOW READ ACCESS is specified, an S lock on the table will be acquired.
- During the truncation phase, an S lock on the table is requested while the reorganization moves rows out of the truncate range, as new rows can be inserted by old scanners (data accesses that exist during the REORG TABLE operation and access the old RIDs for the records). DB2 waits until this lock is achieved before beginning truncation. At the very end of the truncate phase when REORG TABLE does the physical truncation of the table, the S lock will be upgraded to a special Z lock. This means that the REORG TABLE operation cannot complete until there are no table locks held by any existing applications, including IN locks from a UR scanner.
- A row lock might also 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 therefore no further locking is needed.
  If an IS lock is held on the table, then a row-level S lock is acquired before the row is moved, and released after the move is complete.
- Internal locks might be required to control access to the object of an online REORG TABLE and other online DB2 utilities, such as online backup.

Locking has an impact on the performance of both REORG TABLE and on concurrent user applications. It is strongly recommended that you examine lock snapshot data to understand the locking activity when performing 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 get information about the status of your table reorganization operations.

You must be connected to the database and have the following authorization:

- SYSMON authority
- SELECT or CONTROL privilege on the SNAPTAB_REORG administrative view or EXECUTE privilege on the SNAP_GET_TAB_REORG table function.
- To access information about reorganization operations using SQL, use the SNAPTAB_REORG administrative view. For example, the following select statement returns details on table reorganization operations for all database partitions on the currently connected database:

```
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
```

If no tables have been reorganized, 0 rows are returned.

- To access information about reorganization operations using the snapshot monitor, issue the GET SNAPSHOT FOR TABLES ON *database* command and examine the values of the table reorganization monitor elements (prefixed with "reorg_").

Since offline REORG TABLE is synchronous, any errors in an offline table reorganization are returned to the caller of the utility (either an application or the command line.)

Error handling in online table reorganization is slightly different from error handling in an offline table reorganization. Since online table reorganization is asynchronous, no SQL messages are written to the CLP. To view SQL errors returned during an online table reorganization, issue a "LIST HISTORY REORG" command.

An online table reorganization runs in the background as the `db2Reorg` process. This means that a successful return from the REORG TABLE process to the calling application does not mean that the reorganization completed successfully. Even if the calling application terminates its database connection, the `db2Reorg` process will continue.

Index reorganization

As tables are updated with deletes and inserts, index performance degrades in the following ways:

- Fragmentation of leaf pages

  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, which is referred to as a *badly clustered* index.

  When leaf pages are badly clustered, sequential prefetching is inefficient and results in more I/O waits.
- The index develops more than its maximally efficient number of levels.

  In this case, the index should be reorganized.

If you set the MINPCTUSED parameter when you create an index, the database server automatically merges index leaf pages if a key is deleted and the free space is less than the specified percent. This process is called *online index defragmentation*. However, to restore index clustering, free space, and reduce leaf levels, you can use one of the following methods:

- Drop and recreate the index.
- Use the REORG INDEXES command to reorganize indexes online.

  You might choose this method in a production environment because it allows users to read from and write to the table while its indexes are being rebuilt.
- Use the REORG TABLE command with options that allow you to reorganize both the table and its indexes off-line.

Online index reorganization

When you use the REORG INDEXES command with the ALLOW WRITE ACCESS option, all indexes on the specified table are rebuilt while read and write access to the table is allowed. Any changes made to the underlying table that would affect indexes while the reorganization is in progress are logged in the DB2 logs. In

addition, the same changes are placed in the internal memory buffer space, if there is any such memory space available for use. The reorganization will process the logged changes to catch up with current writing activity while rebuilding the indexes. The internal memory buffer space is a designated memory area allocated on demand from the utility heap to store the changes to the index being created or reorganized. The use of the memory buffer space allows the index reorganization to process the changes by directly reading from memory first, and then reading through the logs if necessary, but at a much later time. The allocated memory is freed once the reorganization operation completes. Following the completion of the reorganization, the rebuilt index might not be perfectly clustered. If PCTFREE is specified for an index, that percent of space is preserved on each page during reorganization.

For partitioned tables, online index reorganization and cleanup of individual indexes is supported. For reorganization of individual indexes specify the index name: `REORG INDEX index_name for TABLE table_name`

Online index reorganization in ALLOW WRITE mode is not supported for spatial indexes or multi-dimensionally clustered (MDC) tables.

Note: The CLEANUP ONLY option of the REORG INDEXES/INDEX command does not fully reorganize indexes. The CLEANUP ONLY ALL option removes keys that are marked deleted and are known to be committed. It also frees pages in which all keys are marked deleted and are known to be committed. When pages are freed, adjacent leaf pages are merged if doing so can leave at least PCTFREE free space on the merged page. PCTFREE is the percentage of free space defined for the index when it is created. The CLEANUP ONLY PAGES option deletes only pages in which all keys are marked deleted and are known to be committed.

When reorganizing indexes on partitioned tables using the CLEANUP ONLY option, any access level is supported. If the CLEANUP ONLY option is not specified, the default access level ALLOW NO ACCESS is the only supported access level.

Index reorganization has the following requirements:

- SYSADM, SYSMAINT, SYSCTRL or DBADM authority, or CONTROL privilege on the indexes and table
- An amount of free space in the table space where the indexes are stored equal to the current size of the index

  Consider placing indexes subject to reorganization in a large table space when you issue the CREATE TABLE statement.
- Additional log space

  Index reorganization logs its activity. As a result, the reorganization might fail, especially if the system is busy and other concurrent activity is logged.

  Note: If a REORG INDEXES ALL with the ALLOW NO ACCESS option fails, the indexes are marked invalid and the operation is not undone. However, if a REORG with the ALLOW READ ACCESS or a REORG with the ALLOW WRITE ACCESS option fails, the original index object is restored.

Determining when to reorganize tables and indexes

After many changes to table data, logically sequential data might be located on non-sequential physical data pages, especially if many update operations have created overflow records. When the data is organized in this way, the database

manager must perform additional read operations to access sequential data. Additional read operations are also required if a significant number of rows have been deleted.

Table reorganization defragments the data, eliminating wasted space, and reorders the rows to incorporate overflow records, improving data access and ultimately query performance. You can also specify that the data be reordered according to a particular index, so that queries can access the data with minimal data reads.

Many changes to table data will cause updates to the indexes and index performance can degrade. Index leaf pages can become fragmented or badly clustered and the index could develop more level than necessary for optimal performance. All of these issues will cause more I/O's and will degrade performance

Any of the following factors might indicate that you should reorganize a table or index:

- A high volume of insert, update, and delete activity on 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 RUNSTATS to refresh statistical information does not improve performance
- The REORGCHK command indicates a need to reorganize your table or index (NOTE: In some cases REORGCHK will always recommend a table reorganization, even after a table reorganization has been performed. For example, using a 32kb page size with an average record length of 15 bytes and a maximum of 253 records per page means that each page has 32700- (15 x 253)=28905 non-usable bytes. This means that approximately 88% of the page is free space. Users should analyze the recommendations of REORGCHK and balance the benefit against the cost to perform the reorganization.
- The db.tb_reorg_req (Reorganization Required) health indicator is in ATTENTION state. The collection details for this health indicator describes the list of tables and indexes that could benefit from reorganization.

The REORGCHK command returns statistical information about data organization and can advise you about whether particular tables or indexes need to be reorganized. However, running specific queries against the catalog statistics tables at regular intervals or specific times can provide a performance history that allows you to spot trends that might have wider implications for performance.

To determine whether you need to reorganize tables or indexes, query the catalog statistics tables and monitor the following statistics:

1. Overflow of rows

   Query the OVERFLOW column in the SYSSTAT.TABLES view to monitor the overflow value. The values in this column represent the number of rows that do not fit on their original pages. Row data can overflow when variable length columns in the table cause the record length to expand such that they no longer fit into their assigned locations on the data page. Length changes can also occur if a column is added to the table definition and later materialized by updating the row. In such cases, a pointer is kept at the original location in the row and the actual 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 row. This two-step process increases the processing time and might also increase the number of I/Os required.

Reorganizing the table data will eliminate the row overflows; therefore, as the number of overflow rows increases, the potential benefit of reorganizing your table data increases.

2. Fetch statistics

   Query the three following columns in the SYSCAT.INDEXES and SYSSTAT.INDEXES catalog statistics tables to determine the effectiveness of the prefetchers when the table is accessed in index order. These statistics 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 in the table. 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 specified by the PREFETCHSIZE setting for the table space. A large number indicates that the prefetchers are performing effectively. For a clustered index and table, this 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 between sequential page accesses when fetching table rows using the 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. Such disorganization is usually caused by inserts that occur out of sequence, either at the end of the table or in overflow pages. This causes fetches that slow query performance when the index is used to access a range of values.
   - The AVERAGE_SEQUENCE_FETCH_GAP column stores the average gap between table page sequences when fetching using the 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. These occur when many pages are accessed randomly, which interrupts the prefetchers. A large number indicates a table that is disorganized or poorly clustered to the index.

3. Number of index leaf pages that contain RIDs marked deleted but not removed

   In type-2 indexes, RIDs are not usually physically deleted when the RID is 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 SYSCAT.INDEXES and SYSSTAT.INDEXES statistics tables. 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 might be reclaimed by executing REORG INDEXES with the CLEANUP ALL option. To reclaim only the space in pages on which all RIDs are marked deleted, execute REORG INDEXES with the CLEANUP ONLY PAGES option.

4. Cluster-ratio and cluster-factor statistics for indexes

   A cluster-ratio statistic is stored in the CLUSTERRATIO column of the SYSTCAT.INDEXES catalog table. This value, between 0 and 100, represents the degree of data clustering with the index. If you collect DETAILED index statistics, a finer cluster-factor statistic between 0 and 1 is stored in the CLUSTERFACTOR column instead, and the value of CLUSTERRATIO is -1.

Only one of these two clustering statistics can be recorded in the SYSCAT.INDEXES catalog table. To compare CLUSTERFACTOR values with the CLUSTERRATIO values, multiply the CLUSTERFACTOR by 100 to obtain a percentage.

Note: In general, only one of the indexes in a table can have a high degree of clustering.

Index scans that are not index-only accesses might perform better with higher cluster ratios. A low cluster ratio leads to more I/O for this type of scan, since after the first access of each data page, it is less likely that the page is still in the buffer pool the next time it is accessed. Increasing the buffer size might also improve the performance of an unclustered index.

If table data was initially clustered on a certain index, and the clustering statistics information indicates that the data is now poorly clustered for that same index, you may want to reorganize the table to cluster the data again.

5. Number of leaf pages

   Query the NLEAF column in the SYSCAT.INDEXES table to find out the number of leaf pages occupied by an index. The number tells you how many index page I/Os are needed for a complete scan of an index.

   Ideally, an index should take up the minimum amount of space possible to reduce the I/Os required for an index scan. Random update activity can cause page splits that increase the size of the index. When indexes are rebuilt during the reorganization of a table, each index can be built with the minimum amount of space.

   Note: By default, ten percent free space is left on each index page when the indexes are built. To increase the free space amount, specify the PCTFREE parameter when you create the index. Whenever you reorganize the index, the PCTFREE value is used. Free space greater than ten percent might reduce frequency of index reorganization because the additional space can accommodate additional index inserts.

6. Number of empty data pages

   To calculate the number of empty pages in a table, query the FPAGES and NPAGES columns in SYSCAT.TABLES and subtract the NPAGES number from the FPAGES number. The FPAGES column stores the total number of pages in use; the NPAGES column stores the number of pages that contain rows. Empty pages can occur when entire ranges of rows are deleted.

   As the number of empty pages increases, the need for a table reorganization increases. Reorganizing the table reclaims the empty pages and reduces the amount of space used by a table. In addition, because empty pages are read into the buffer pool for a table scan, reclaiming unused pages can improve the performance of a table scan.

   When the total number of pages (FPAGES) in a table is smaller than or equal to NPARTITIONS *1 extent size, table reorganization is not recommended. NPARTITIONS is the number of data partitions if this is a partitioned table, otherwise it is 1. In a partitioned database environment, after the number of database partitions in a database partition group of the table is factored in, the condition changes to FPAGES <= the number of database partitions in a database partition group of a table * NPARTITIONS * 1 extent size

Before reorganizing, consider the trade-off between the cost of increasing degradation of query performance and the cost of reorganizing your table or

indexes, which includes the CPU time, the elapsed time, and the reduced concurrency resulting from the REORG utility locking the table until the reorganization is complete.

Costs of table and index reorganization

Performing a reorganization on a table or index incurs a certain amount of overhead that must be considered when deciding whether to reorganize an object.

The costs of reorganizing tables and indexes include:

- CPU consumption of the executing utility
- Reduction in concurrency while running the REORG utility. Concurrency is reduced because of the locking requirements of REORG.
- Extra storage requirements. (Offline table reorganization requires extra storage space to hold a shadow copy of the table. Online or inplace table reorganizations require additional logging space. Online index reorganization requires extra storage space to hold a shadow copy of the index or indexes and additional log space. Offline index reorganization will use less log space and does not involve a shadow copy.)

In some cases, a reorganized table might be larger than the original table, increasing the space requirements accordingly. A table may grow after reorganization in the following situations:

- In a clustering REORG TABLE where an index is used to determine the order of the rows, if the table records are of a variable length (for example, using varchars) then you might end up using more space since some pages may contain fewer rows than in the original table.
- If the table had columns added prior to the reorganization but after table creation the additional column might be realized for the first time in some rows after the reorganization.
- If the amount of free space left on each page (represented by the value of the PCTFREE attribute) been increased since the last reorganization.
- If the table has a LOB, then there is a possibility that the LOB uses more space than previously.

Space requirements for an offline table reorganization

Since 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 that the original table resides in or in a user-specified temporary table space.

Additional temporary table space storage may be required for sort processing if a table scan sort is used. The additional space required could be as large as the size of the table being reorganized. If the clustering index is SMS type or unique DMS type, the recreation of this index will not require a sort. Instead, this index is rebuilt by scanning the newly reorganized data. Any other indexes that require recreating will involve a sort, potentially requiring space the temporary table space up to the size of the table being reorganized.

Offline table REORG generates few control log records, and therefore consumes a relatively small amount of log space. If the REORG does not use an index, then the only log records are the table data log records. If an index is specified, or if a clustering index exists on the table, then the RIDs of the rows are logged in the order that they are placed into the new version of the table. Each log record for the

RIDs holds a maximum of 8000 RIDs, with each RID consuming 4 bytes. This can be a contributing factor in running out of log space during an offline table reorganization. Note that RIDs are only logged if the database is recoverable (LOGRETAIN=ON).

Log space requirements for an online table reorganization

The log space required for an online table REORG is typically larger than what is required for an offline table REORG. The amount of space 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 utilization for your tables.

For every row in the table, it will likely be moved twice during an online table reorganization. Given one index, each row has to update the index key to reflect the new location, and once all accesses to the old location are complete, the key is updated again to remove the reference to the old RID. When the row is moved back, these updates to the index key are performed again. All of this activity is logged to make online table reorganization fully recoverable, therefore there is a minimum of two data log records (each instance including the row data) and four index log records (each instance including the key data). The clustering index in particular is prone to filling up the index pages, causing index splits and merges which also must be logged.

Since online table reorganization frequently issues internal commits, it usually does not hold significant logs as active. If there is a time when an online REORG does hold a large number of active logs, it is during the truncate phase, since it acquires an S table lock at the truncate phase. If the table reorganization cannot acquire the lock, it waits and holds the log, and other transactions might fill up the logs quickly.

Reducing the need to reorganize tables and indexes

You can employ different strategies to reduce the frequency with which you need to reorganize your tables and indexes, avoiding the cost of performing unnecessary reorganization operations.

Reducing the need to reorganize tables

- Use multi-partition tables. The smaller the table is, the less likely it will need to be reorganized.
- Create multi-dimensional clustering (MDC) tables, for which clustering is automatically maintained on the columns specified in the ORGANIZE BY DIMENSION clause of the CREATE TABLE statement.
- Turn on APPEND mode on tables. If the index key values of these new rows are always new high key values for example, then the clustering attribute of the table will try to place them at the end of the table. In this case, placing the table in append mode may be a better choice than a clustering index.
- After you create a table:
  - Alter table to add PCTFREE
  - Create a clustering index with PCTFREE specified on index.
  - Sort the data before loading it into the table

  A clustering index with an appropriate setting for PCTFREE on a table helps preserve the original sorted order. If enough space is allowed in table pages, new data can be inserted on the correct pages to maintain the clustering

characteristics of the index. As more data is inserted and the pages of the table become full, records are appended to the end of the table so that the table gradually becomes unclustered.

If you perform a REORG TABLE or a sort and LOAD after you create a clustering index, the index attempts to maintain a particular order of data, which improves the CLUSTERRATIO or CLUSTERFACTOR statistics collected by the RUNSTATS utility.

Reducing the need to reorganize indexes

- Create clustering indexes with PCTFREE or LEVEL2 PCTFREE on index pages. The range is from 0 to 99% with a default value of 10%.
- Create indexes with MINPCTUSED. The possible range is from 0 to 99%, with a recommended value of 50%. However, consider using the CLEANUP ONLY ALL option of the REORG INDEXES command to merge leaf pages instead.

Automatic reorganization

After many changes to table data, the table and indexes can become fragmented. Logically sequential data may be on non-sequential physical pages and so the database manager has to perform additional read operations to access data.

Among other information, the statistical information collected by RUNSTATS shows the data distribution within a table. In particular, analysis of these statistics can indicate when and what kind of reorganization is necessary. Automatic reorganization determines the need for reorganization on tables and indexes by using the REORGCHK formulas. It periodically evaluates tables and indexes that have had their statistics updated to see if reorganization is required. If so, it internally schedules an index reorganization or a classic table reorganization for the table. This requires that your applications function without write access to the tables being reorganized.

The automatic reorganization feature can be enabled or disabled by using the AUTO_REORG, AUTO_TBL_MAINT, and AUTO_MAINT database configuration parameters.

In a partitioned database environment, the determination to carry out automatic reorganization and the inititation of automatic reorganization, is done on the catalog partition. The database configuration parameters need to be enabled on the catalog partition only. The reorganization runs on all of the database partitions on which the target tables reside.

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.

Tables and indexes considered for automatic reorganization are configurable by you using the Automatic Maintenance wizard from the Control Center or Health Center.

Enabling automatic table and index reorganization

Having well-organized table and index data is critical to efficient data access and optimal workload performance. After many changes to table data, logically sequential data may be on non-sequential physical data pages so that the database manager must perform additional read operations to access data. Additional read operations are also required if a significant number of rows have been deleted. Use

automated table reorganization to enable DB2 to manage offline table and index reorganization so that you don't have to worry about when and how to reorganize the data. You can enable DB2 to reorganize the system catalog tables as well as database tables.

You can turn this feature on using either the graphical user interface tools or the command line interface.

- To set up your database for automatic reorganization using the graphical user interface tools:
  1. Open the Configure Automatic Maintenance wizard either from the Control Center by right-clicking on a database object or from the Health Center by right-clicking on the database instance that you want to configure for automatic reorganization. Select **Configure Automatic Maintenance** from the pop-up window.
  2. Within this wizard, you can enable automatic reorganization to defragment data, specify the tables that you want to automatically reorganize, and specify a maintenance window for the execution of the REORG utility.
- To set up your database for automatic reorganization using the command line interface, set each of the following configuration parameters to ″ON″:
  - AUTO_MAINT
  - AUTO_TBL_MAINT
  - AUTO_REORG

Using relational indexes to improve performance

Indexes can be used to improve performance when accessing table data. Relational indexes are used when working with relational data. For XML data access, indexes over XML data are used.

Although the optimizer decides whether to use a relational index to access relational table data, except in the following case, you must decide which indexes might improve performance and create these indexes. Exceptions are the dimension block indexes and the composite block index that are created automatically for each dimension that you specify when you create a multi-dimensional clustering (MDC) table.

You must also execute the RUNSTATS utility to collect new statistics about the relational indexes in the following circumstances:

- After you create a relational index
- After you change the prefetch size

You should also 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.

Note: To determine whether a relational index is used in a specific package, use the Explain facility. To plan relational indexes, use the Design Advisor from the Control Center or the `db2advis` tool to get advice about relational indexes that might be used by one or more SQL statements.

Advantages of a relational index over no index

If no index exists on a table, a table scan must be performed for each table referenced in an SQL query. The larger the table, the longer a table scan takes

because a table scan requires each table row to 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 generally are smaller and require less time to read than an entire table, particularly as tables grow larger. In addition, the entire index may not need to be scanned. The predicates that are applied to the index reduce the number of rows to be read from the data pages.

If an ordering requirement on the output can be matched with an index column, then scanning the index in column order will allow the rows to be retrieved in the correct order without a sort.

Each relational index entry contains a search-key value and a pointer to the row containing that value. If you specify the ALLOW REVERSE SCANS parameter in the CREATE INDEX statement, the values can be searched in both ascending and descending order. It is therefore possible to bracket the search, given the right predicate. A relational index can also be used to obtain rows in an ordered sequence, eliminating the need for the database manager to sort the rows after they are read from the table.

In addition to the search-key value and row pointer, a relational index can contain include columns, which are non-indexed columns in the indexed row. Such columns might make it possible for the optimizer to get required information only from the index, without accessing the table itself.

Note: 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.

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:

- Each index requires storage or disk 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 performed on 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.
- The LOAD utility rebuilds or appends to any existing relational indexes.

  The `indexfreespace` MODIFIED BY parameter can be specified on the LOAD command to override the index PCTFREE used when the relational index was created.
- Each relational index potentially adds an alternative access path for an SQL query for the optimizer to consider, which increases the compilation time.

Choose indexes carefully to address the needs of the application program.

Relational index planning tips

The relational indexes that you create should depend on the relational data and the queries that access it.

Use the Design Advisor from the Control Center or the `db2advis` tool to find the best indexes for a specific query or for the set of queries that defines a workload. This tool recommends relational indexes with such performance enhancing features as INCLUDE columns, inherited unique indexes, and ALLOW REVERSE SCANS indexes.

The following guidelines can help you determine how to create useful relational indexes for various purposes:

- To avoid some sorts, define primary keys and unique keys, wherever possible, by using the CREATE UNIQUE INDEX statement.
- To improve data-retrieval, add INCLUDE columns to unique indexes. Good candidates are columns that:
  - Are accessed frequently and therefore 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.
- To access small tables efficiently, use relational indexes to optimize frequent queries to tables with more than a few data pages, as recorded in the NPAGES column in the SYSCAT.TABLES catalog view. You should:
  - Create an index on any column you will use when joining tables.
  - Create an index on any column from which you will be searching for particular values on a regular basis.
- To search efficiently, decide between ascending and descending ordering of keys depending on the order that will be used most often. Although the values can be searched in reverse direction if you specify the ALLOW REVERSE SCANS parameter in the CREATE INDEX statement, scans in the specified index order perform slightly better than reverse scans.
- To save index maintenance costs and space:
  - Avoid creating relational indexes that are partial keys of other index keys on the columns. For example, if there is an index on columns a, b, and c, then a second index on columns a and b is not generally useful.
  - Do not create relational indexes arbitrarily on all columns. Unnecessary indexes not only use space, but also cause large prepare times. This is especially important for complex queries, when an optimization class with dynamic programming join enumeration is used.

    Use the following general rule for the typical number of relational indexes that you define for a table. This number is based on the primary use of your database:

    - For online transaction processing (OLTP) environments, create one or two indexes
    - For read-only query environments, you might create more than five indexes
    - For mixed query and OLTP environments, you might create between two and five indexes.
- To improve performance of delete and update operations on the parent table, create relational indexes on foreign keys.
- To improve performance of DELETE and UPDATE operations involving IMMEDIATE and INCREMENTAL MQTs, create unique relational indexes on the implied unique key of the MQT, which is the columns in the GROUP BY clause of the MQT definition.
- For fast sort operations, create relational indexes on columns that are frequently used to sort the relational data.

- To improve join performance with a multiple-column relational index, if you have more than one choice for the first key column, use the column most often specified with the "=" (equijoin) predicate or the column with the greatest number of distinct values as the first key.
- To help keep newly inserted rows clustered according to an index and avoid page splits, define a clustering index. A clustering index should significantly reduce the need for reorganizing the table.

  Use the PCTFREE keyword when you define the table to specify how much free space should be left on the page to allow inserts to be placed appropriately on pages. You can also specify the `pagefreespace` MODIFIED BY clause of the LOAD command.
- To enable online index defragmentation, use the MINPCTUSED option when you create relational indexes. MINPCTUSED specifies the threshold for the minimum amount of used space on an index leaf page as well as enabling online index defragmentation. This might reduce the need for reorganization at the cost of a performance penalty during key deletions if these deletions physically remove keys from the index page.

Consider creating a relational index in the following circumstances:

- Create a relational index on columns that are used in WHERE clauses of the queries and transactions that are most frequently processed.

  The WHERE clause:

  ```
  WHERE WORKDEPT='A01' OR WORKDEPT='E21'
  ```

  will generally benefit from an index on `WORKDEPT`, unless the `WORKDEPT` column contains many duplicate values.
- Create a relational index on a column or columns to order the rows in the sequence required by the query. Ordering is required not only in the ORDER BY clause, but also by other features, such as the DISTINCT and GROUP BY clauses.

  The following example uses the DISTINCT clause:

  ```
  SELECT DISTINCT WORKDEPT
    FROM EMPLOYEE
  ```

  The database manager can use an index defined for ascending or descending order on `WORKDEPT` to eliminate duplicate values. This same index could also be used to group values in the following example with a GROUP BY clause:

  ```
  SELECT WORKDEPT, AVERAGE(SALARY)
    FROM EMPLOYEE
  GROUP BY WORKDEPT
  ```
- Create a relational index with a compound key that names each column referenced in a statement. 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 SQL statement:

  ```
  SELECT LASTNAME
    FROM EMPLOYEE
    WHERE WORKDEPT IN ('A00','D11','D21')
  ```

  If a relational index is defined for the `WORKDEPT` and `LASTNAME` columns of the `EMPLOYEE` table, the statement might be processed more efficiently by scanning the index than by scanning the entire table. Note that since the predicate is on `WORKDEPT`, this column should be the first column of the relational index.

- Create a relational index with INCLUDE columns to improve the use of indexes on tables. Using the previous example, you could define a unique relational index as:

  ```
  CREATE UNIQUE INDEX x ON employee (workdept) INCLUDE (lastname)
  ```

  Specifying `lastname` as an INCLUDE column rather than as part of the index key means that `lastname` is stored only on the leaf pages of the index.

Relational index performance tips

Consider the following suggestions for using and managing relational indexes:

- **Specify a large utility heap**

  Write access by other users or applications to the underlying table is supported for both CREATE INDEX and REORG INDEXES. When you expect a lot of update activity on the underlying table for the relational index being created or reorganized, consider configuring a large utility heap. A large utility heap will speed up the index creation or index reorganization during the catch up phase. All writing activity on the index or indexes being created or reorganized is logged in the DB2 logs and in the internal memory buffer space. The internal memory buffer space is a designated memory area allocated on demand from the utility heap to store the changes to the index being created or reorganized. It is the use of this memory that allows the catch up phase to work quickly. The allocated memory is freed once the create or reorganization operations complete. Ensuring that there is enough utility heap to accommodate all or most of the changes to the indexes that are being created or reorganized can have a very positive performance impact on the catch up phase.

- **Increase sheapthres configuration parameter if running on an SMP machine**

  Each subagent will acquire the amount of memory specified by the**sortheap** configuration parameter in order to scan the table to avoid sort overflow. You should monitor the number of sort overflows and increase **sheapthres** accordingly.

- **Specify separate table spaces for relational indexes**

  Indexes can be stored in a different table space from the table data. This can allow for more efficient use of disk storage by reducing the movement of read/write heads during index access. You can also create index table spaces on faster physical devices. In addition, you can assign the index table space to a different buffer pool, which might keep the index pages in the buffer longer because they do not compete with table data pages.

  When you do not place indexes in separate table spaces, both data and index pages use the same extent size and prefetch quantity. If you use a different table space for indexes, you can select different values for all the characteristics of a table space. Because indexes are usually smaller than tables and are spread over fewer containers, indexes often have smaller extent sizes, such as 8 and 16 pages. The query optimizer considers the speed of the device for a table space when it chooses an access plan.

- **Ensure the degree of clustering**

  If your SQL statement requires ordering (for example, if it contains ORDER BY, GROUP BY, and DISTINCT SQL clauses), the optimizer might not choose the index even though it satisfies the ordering in the following cases:

  – Index clustering is poor. For information, examine the CLUSTERRATIO and CLUSTERFACTOR columns of SYSCAT.INDEXES.

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

– There are competing indexes for accessing the table.

After you create a clustering index, perform a REORG TABLE in classic mode, which creates a perfectly organized index. To recluster the table, you might perform a sort and LOAD instead. In general, a table can only be clustered on one index. Build additional indexes after you build the clustering index. A clustering index attempts to maintain a particular order of data, improving the CLUSTERRATIO or CLUSTERFACTOR statistics collected by the RUNSTATS utility.

To help maintain the clustering ratio, specify an appropriate PCTFREE when you alter a table before you load or reorganize that table. The free space on each page specified by PCTFREE provides space for inserts, so that these inserts can be clustered appropriately. If you do not specify PCTFREE for the table, reorganization eliminates all extra space.

Note: Clustering is not currently maintained during updates unless you are using range-clustered tables. That is, if you update a record so that its key value changes in the clustering index, the record is not necessarily moved to a new page to maintain the clustering order. To maintain clustering, use DELETE and then INSERT instead of UPDATE.

- **Keep table and index statistics up-to-date**

  After you create a new relational index, run the RUNSTATS utility to collect index statistics. These statistics allow the optimizer to determine whether using the index can improve access performance.

- **Enable online index defragmentation**

  Online index defragmentation is enabled if the MINPCTUSED clause is set to greater than zero for the relational index. Online index defragmentation allows indexes to be compacted by merging leaf pages when the free space on a page falls at or below the specified level while the index remains available.

- **Reorganize relational indexes as necessary**

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

  To reorganize the index, either drop it and re-create it or use the REORG utility.

  To reduce the need for frequent reorganization, when you create a relational index specify an appropriate PCTFREE to leave a percentage of free space on each index leaf page as it is created. During future activity, records can be inserted into the index with less likelihood of causing index page splits. Page splits cause index pages not to be contiguous or sequential, which in turn results in decreased efficiency of index page prefetching.

  Note: The PCTFREE specified when you create the relational index is retained when the index is reorganized.

  Dropping and re-creating or reorganizing the relational index also creates a new set of pages that are roughly contiguous and sequential and improves index page prefetch. Although more costly in time and resources, the REORG TABLE utility also ensures clustering of the data pages. Clustering has greater benefit for index scans that access a significant number of data pages.

  In a symmetric multi-processor (SMP) environment, the “classic” REORG TABLE mode, which uses a shadow table for fast table reorganization, can use multiple processors to rebuild the indexes.

- **Analyze EXPLAIN information about relational index usage**

Periodically, run EXPLAIN on your most frequently used queries and verify that each of your relational indexes is used at least once. If an index is not used in any query, consider dropping that index.

EXPLAIN information also lets you see if table scans on large tables are processed as the inner table of nested loop joins. If they are, an index on the join-predicate column is either missing or considered ineffective for applying the join predicate.

- **Use volatile tables for tables that vary widely in size**

  A *volatile* table is a table that might vary in size at run time from empty to very large. For this kind of table, in which the cardinality varies greatly, the optimizer might generate an access plan that favors a table scan instead of an index scan.

  Declaring a table "volatile" using the ALTER TABLE...VOLATILE statement allows the optimizer to use an index scan on the volatile table. The optimizer will use an index scan instead of a table scan regardless of the statistics in the following circumstances:

  – All columns referenced are in the index

  – The index can apply a predicate in the index scan.

  If the table is a typed table, using the ALTER TABLE...VOLATILE statement is supported only on the root table of the typed table hierarchy.

Index cleanup and maintenance

After you create indexes, performance degrades unless you keep the index compact and organized. Consider the following suggestions to 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 get X locks on tables, either explicitly or by lock escalation, if frequent COMMITS are not possible.

  Index keys marked deleted can be physically removed from the table after the COMMIT. X locks on tables allow the deleted key to be physically removed when it is marked deleted, as explained below.

- Use REORGCHK to help determine when to reorganize indexes or tables, or both, and when to use the REORG INDEXES with the CLEANUP ONLY option.

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

Note: In DB2 Version 8.1 and later, all new indexes are created as type-2 indexes. The one exception is when you add an index on a table that already has type-1 indexes. In this case only, the new index will also be a type-1 index. To find out what type of index exists for a table, execute the INSPECT command. To convert type-1 indexes to type-2 indexes, execute the REORG INDEXES command.

The primary advantages of type-2 indexes are as follows:

- An index can be created on columns whose length is greater than 255 bytes.
- The use of next-key locking is reduced to a minimum, which improves concurrency. Most next-key locking is eliminated because a key is marked deleted instead of being physically removed from the index page. For information about key locking, refer to topics that discuss the performance implications of locks.

Index keys that are marked deleted are cleaned up in the following circumstances:

- During subsequent insert, update, or delete activity

  During key insertion, keys that are marked deleted and are known to be committed are cleaned up if such a cleanup 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 this 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 CLEANUP options

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

  The CLEANUP ONLY ALL option frees not only index pages on which all keys are marked deleted and known to be committed, but it also removes RIDs marked deleted and known to be committed on pages that contain some undeleted 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 on the merged leaf page. The PCTFREE value is the percent of free space defined for the index when it is created. The default PCTFREE is ten percent. If two pages can be merged, one of the pages will be freed.

  For partitioned tables, you are encouraged to execute RUNSTATS after an asynchronous index cleanup has completed in order to generate accurate index statistics in the presence of detached data partitions. To determine whether or not there are detached data partitions in the table, you can check the status field in the SYSDATAPARTITIONS table and look for the value ″I″ (index cleanup) or ″D″ (detached with dependant MQT).
- Any rebuild of an index

  Utilities that rebuild indexes include the following:
  - REORG INDEXES when not using one of the CLEANUP options
  - REORG TABLE when not using the INPLACE option
  - IMPORT with the REPLACE option
  - LOAD with the INDEXING MODE REBUILD option

Understanding index behavior on partitioned tables

Indexes on partitioned tables are similar to indexes for ordinary tables, that is each index contains pointers to rows in all the data partitions of the table. One important difference however is that each index on a partitioned table is an independent object. In partitioned database environments, the index is distributed across the database partitions in the same manner as the table. Because an index on a partitioned table can act independently of other indexes, some special considerations are needed with respect to which table space is used when creating an index on a partitioned table.

An index on a partitioned table is created in a single table space even if the table's data partitions span multiple table spaces. Both DMS and SMS table spaces support the use of indexes in a different location than the table. All table spaces specified must be in the same database partition group. Each 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.

Additional benefits of an index on a partitioned table include:

- Improved performance of drop index and online index create.
- The ability to use different values for any of the table space characteristics between each index on the table (for example, different page sizes for each index may be appropriate to ensure better space utilization).
- Reduced IO contention providing more efficient concurrent access to the index data for the table.
- When individual indexes are dropped space will immediately become available to the system without the need for an index reorganization.
- If you choose to perform index reorganization, an individual index can be reorganized.

Figure 16 shows a non-partitioned index on a partitioned table residing in a single table space.

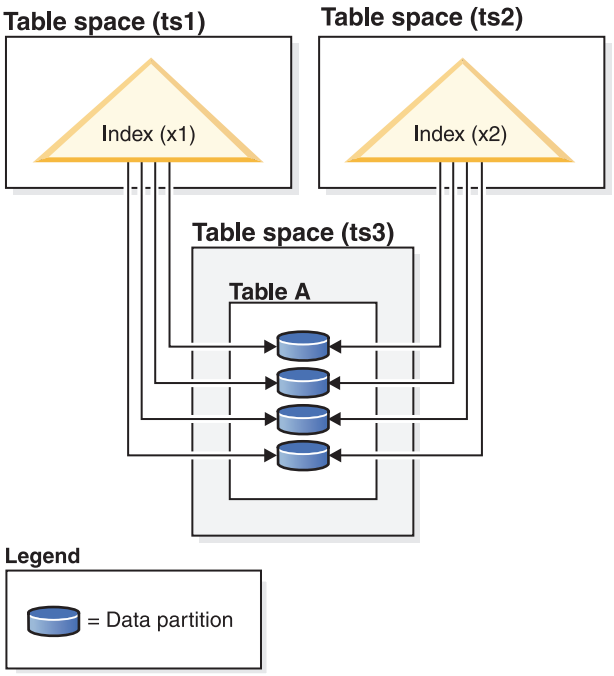


Figure 16. Index behavior on a partitioned table

Figure 17 on page 109 shows index behavior on a partitioned table that is also distributed across multiple database partitions.

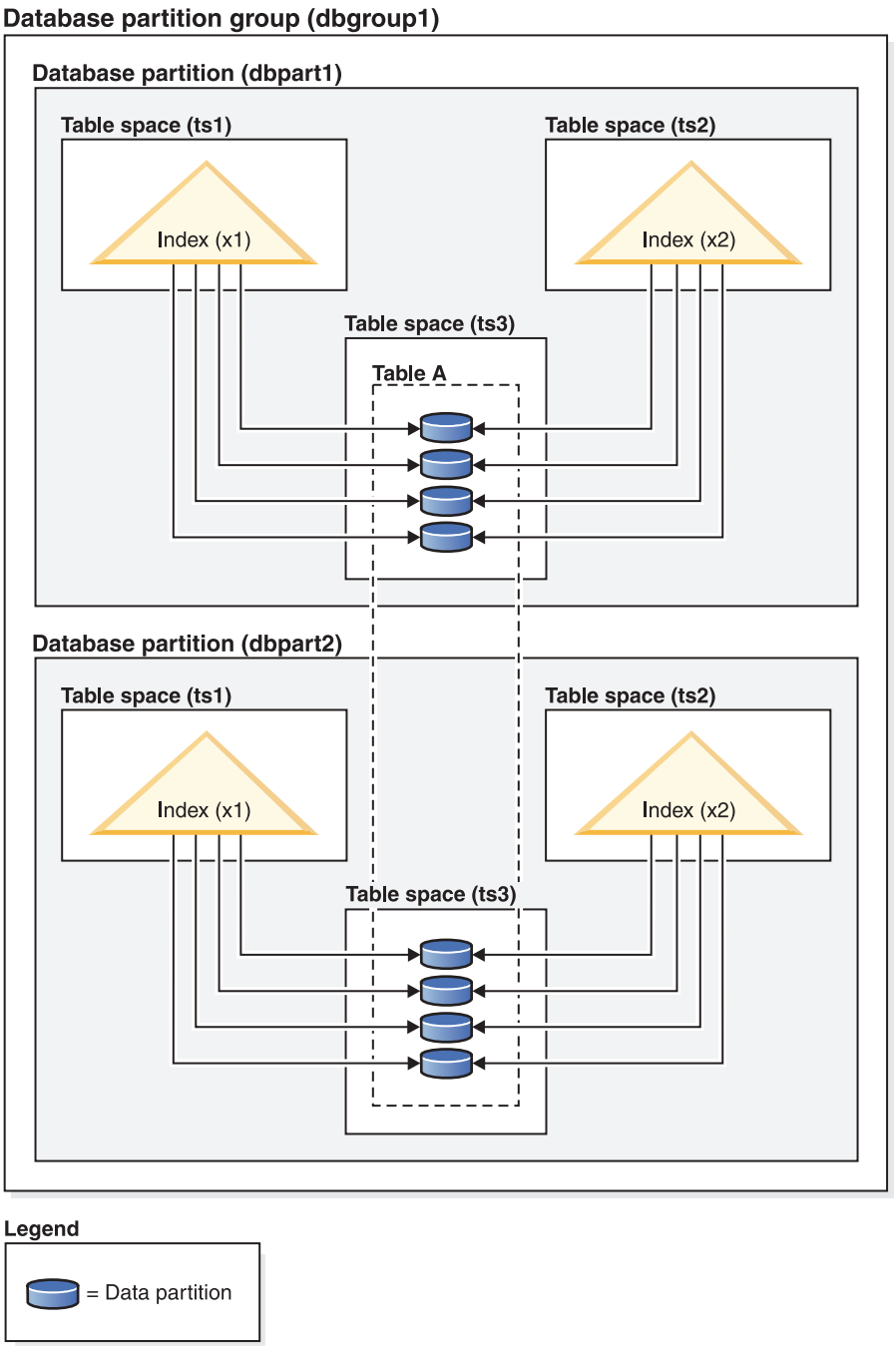


Figure 17. Index behaviour on a table that is both distributed and partitioned.

You can specify an index table space for a partitioned table in the CREATE INDEX ...IN <tbspace1> statement, which can be different from the index table space specified in the CREATE TABLE .. INDEX IN <tbspace2> statement.

For partitioned tables only, you can override the index location by using the IN clause on the CREATE INDEX statement, which allows you to specify a table space location for the index. This approach allows you to place different indexes on a partitioned table in a different table space as required. When you create a partitioned table without specifying where to place its non-partitioned indexes, and you create an index using the CREATE INDEX statement which does not specify a specific 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 created. The evaluation stops when there is a match to one of the cases:

```
Case 1:
When an index table space is specified in CREATE INDEX ... IN
<tbspace1> statement the table space specified in <tbspace1>
is used for this index.
Case 2:
When an index table space is specified in the CREATE TABLE .. INDEX IN
<tbspace2> statement the table space specified in <tbspace2>
is used for this index.
Case 3:
When no table space is specified, use the table space used by the first
attached or visible data partition.
```

Where the index is created depends on what is done during the CREATE TABLE statement. For non-partitioned tables, if you do not specify any INDEX IN clause, the database fills it in for you and is the same as your data table space. For partitioned tables, if you leave it blank, it remains as blank, and case 3 applies.

Example 1: This example assumes the existence of a partitioned table sales (a int, b int, c int), and creates a unique index 'a_idx' in the table space 'ts1'.

```
CREATE UNIQUE INDEX a_idx ON sales ( a ) IN ts1
```

Example 2: This example assumes the existence of a partitioned table sales (a int, b int, c int), and creates an index 'b_idx' in the table space 'ts2'.

```
CREATE INDEX b_idx ONsales ( b ) IN ts2
```

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 row identifiers (RIDs) or block identifiers (BIDs). Either way, these entries are removed by the index cleaners, which operate asynchronously in the background.

AIC accelerates the detach of a data partition from a partitioned table. If the partitioned table contains one or more non-partitioned indexes, AIC is initiated. In this case, AIC removes all non-partitioned index entries that refer to the detached data partition and any pseudo-deleted entries. After all of the indexes have been cleaned, the identifier associated with the detached data partition is removed from the system catalog.

Note: If the partitioned table has dependent materialized query tables (MQTs) defined, AIC is not initiated until after a SET INTEGRITY operation is performed.

While AIC is in progress, normal table access is maintained. Queries accessing the indexes ignore any non-valid entries that have not yet been cleaned.

In most cases, one cleaner is started for each non-partitioned index associated with the partitioned table. An internal task distribution daemon is responsible for distributing the AIC tasks to the appropriate database partitions and assigning database agents.

Both the distribution daemon and cleaner agents are internal system applications. They appear in the LIST APPLICATION 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 the cleaning is complete. If the database is deactivated while cleaning is in progress, AIC resumes when you reactivate the database.

Performance

AIC incurs minimal performance impact.

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

Each cleaner acquires a minimal table space lock (IX) and table lock (IS). These locks are released if the cleaner determines that other applications are waiting for the 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

You can monitor AIC with the LIST UTILITIES command. Each index cleaner appears in the monitor as a separate utility.

The following example demonstrates AIC activity in the WSDB database at the current database partition using the Command Line Processor (CLP) interface:

```
$ db2 list utilities show detail

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; 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 the cleaner.Normally, the state is Executing. The cleaner might be in the Waiting state if the cleaner is waiting to be assigned to an available database agent or if the cleaner is temporarily suspended due to lock contention.

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

Online index defragmentation

Online index defragmentation is enabled by the user-definable threshold for the minimum amount of used space on an index leaf page. When an index key is deleted from a leaf page and the threshold is exceeded, the neighboring index leaf pages are checked to determine if two leaf pages can be merged. If there is sufficient space on a page for a merge of two neighboring pages to take place, the merge occurs immediately in the background.

Online defragmentation of the index is only possible with indexes created in Version 6 and later. If existing indexes require the ability to be merged online, they must be dropped and then re-created with the `MINPCTUSED` clause. Set the `MINPCTUSED` value to less than one hundred (100). The recommended value for `MINPCTUSED` is less than 50 because the goal is to merge two neighboring index leaf pages. A value of zero for `MINPCTUSED`, which is also the default, disables online defragmentation.

Pages in the index are freed when the last index key on the page is removed. The exception to this rule occurs when you specify `MINPCTUSED` clause in the CREATE INDEX statement. The `MINPCTUSED` clause specifies a percent of space on an index leaf page. When an index key is deleted, if the percent of filled space on the page is at or below the specified value, then the database manager tries to merge the remaining keys with keys on an adjacent page. If there is sufficient space on an adjacent page, the merge is performed and an index leaf page is deleted.

Index non-leaf pages are not merged during an online index defragmentation. However, empty non-leaf pages are deleted and made available for re-use by other indexes on the same table. To free these non-leaf pages for other objects in a DMS storage model or to free disk space in an SMS storage model, perform a full reorganization of the table or indexes. Full reorganization of the table and indexes can make the index as small as possible. Index non-leaf pages are not merged during an online index defragmentation, but are deleted and freed for re-use if they become empty. The number of levels in the index and the number of leaf and non-leaf pages might be reduced.

For type-2 indexes, keys are removed from a page during key deletion only when there is an X lock on the table. During such an operation, online index defragmentation will be effective. However, if there is not an X lock on the table during key deletion, keys are marked deleted but are not physically removed from the index page. As a result, no defragmentation is attempted.

To defragment type-2 indexes in which keys are marked deleted but remain in the physical index page, execute the REORG INDEXES command with the CLEANUP

ONLY ALL option. The CLEANUP ONLY ALL option defragments the index, regardless of the value of MINPCTUSED. If you execute REORG INDEXES with the CLEANUP ONLY ALL, two neighbouring leaf pages are merged if such a merge can leave at least PCTFREE free space on the merged page. PCTFREE is specified at index creation time and defaults to ten percent.

Understanding clustering index behavior on partitioned tables

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

You can create clustering indexes 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, a lookup is done in the index to look for a suitable row identifier (RID). This RID is used as a starting point when looking for space in the table to insert the record. To achieve well-maintained 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 by 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 are grouped together by partition IDs to achieve good clustering. For example, if a table is partitioned on quarter, and a clustering index is defined on date, because the relation between quarter and date exists, optimal clustering of the data with good performance can be achieved because all keys of any data partition are grouped together within the index.

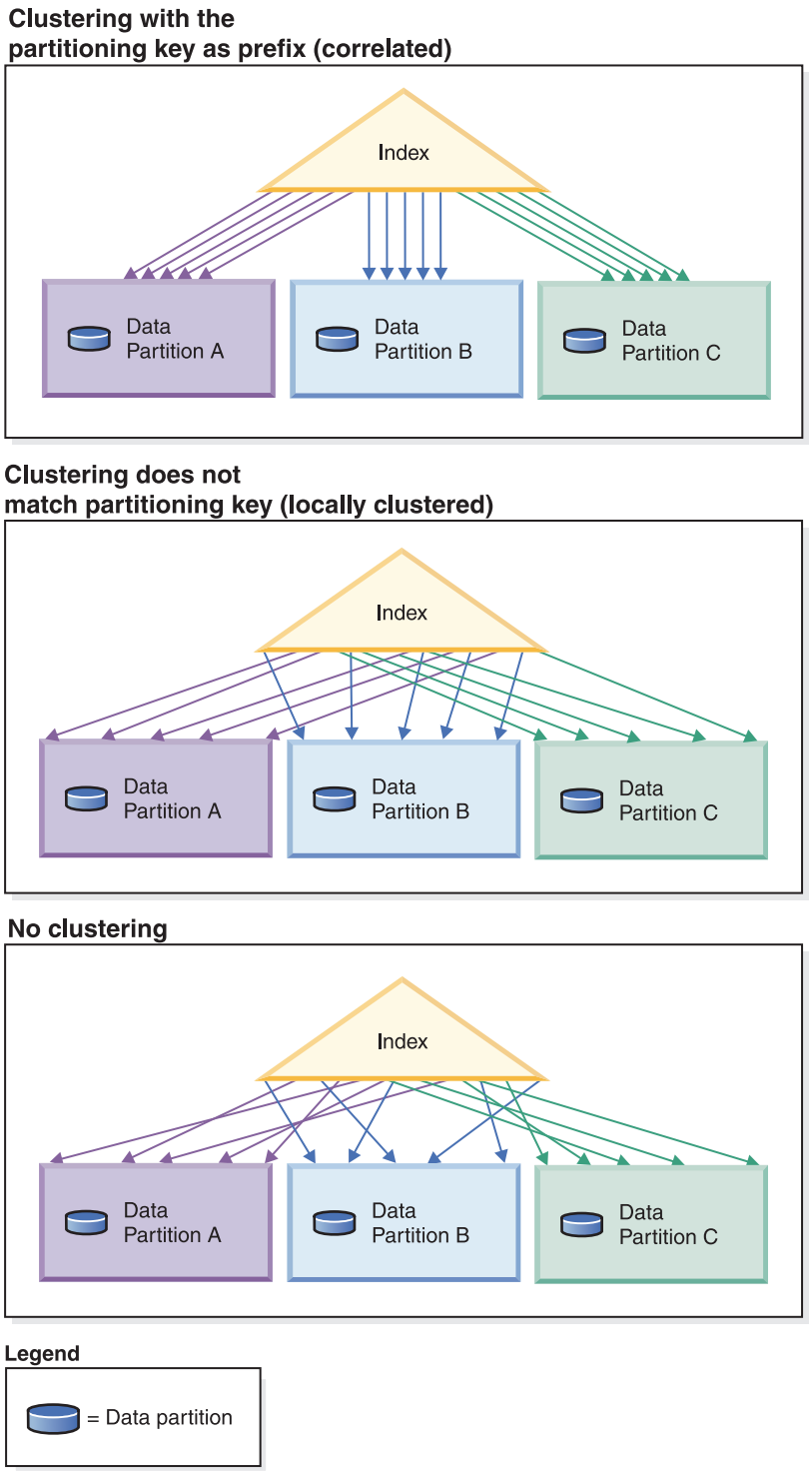


Figure 18. The possible effects of a clustered index on a partitioned table. In the first figure, data is both globally and locally clustered.

As Figure 18 shows, given the layout of the index and data in each example, optimal scan performance is achieved when the clustering correlates to the table partitioning key. When clustering is not correlated to the table partitioning key it is unlikely that the index will be locally clustered. Because a correlation between the table partitioning columns and index columns is expected, a perfect locally clustered scenario is highly unlikely.

Benefits of clustering include:

- Within each data partition, rows are in key order.
- Clustering indexes improve the performance of scans that traverse the table in the order of the keys. This is because the scan fetches the first row of the first page, then each row in that same page until it has fetched all of the rows for that page and moves on to the next. 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, then each row fetched is likely to be from a different page. Unless there is room in the buffer pool to hold the entire table, this will result in each page being fetched more than once, greatly slowing the scan.

  For partitioned tables, the ideal case of fetching each page only once during the scan can be guaranteed only if the table partitioning key is a prefix of the clustering key (see first figure in Figure 18 on page 114). However, if the clustering key is not correlated to the table partitioning key as described previously, and the data is locally clustered, you can still achieve the full benefit of the clustered index if there is enough space to hold one page of each data partition in the buffer pool. This is because each row fetched for a given data partition is near the previous row fetched for that same data partition. (see the second figure of Figure 18 on page 114). As previously mentioned, the clustering may not be well maintained in the case where the clustering key is unrelated to the table partitioning key, but if you do not expect a high level of insert, update and delete activity on your table this approach should be beneficial.

  Even if there is not sufficient space for a page of every data partition to be held in the buffer pool, there is still some benefit to be gained from defining a clustered index.

Database agents

For each database that an application accesses, various processes or threads start to perform the various application tasks. These tasks include logging, communication, and prefetching.

Database agents are threads withing 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 to an application as they remain attached to it until the application disconnects. The only exception to this rule is when engine concentrator is enabled in which case a coordinator agent may terminate that association at transaction boundaries (transaction COMMIT, ROLLBACK)

There are four types of worker agents:

- Idle agents
- Active coordinator agents
- Subagents

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 a database. After the coordinator agent is created, it performs all database requests on behalf of its application, and communicates to other agents using inter-process 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 may become an inactive agent.

When a client disconnects from a database or detaches from an instance its coordinating agent will be:

- An active agent. If other connections are waiting, the worker agent becomes an active coordinator agent.
- 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 agents 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 on the database. In DB2 Version 9.5, subagents may also exist in non-partitioned environments and in environments where intra-query-parallelism is not enabled.

Agents that are not performing work for any applications 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 for client programs or for subagents operating for existing coordinator agents. The number of available agents depends on the database manager configuration parameter *num_poolagents*.

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. However, it may be that your work environment is such that you require a many-to-one relationship between the number of connected applications and the number of application requests that can be processed.

The ability to control these factors separately is provided by two database manager configuration parameters:

- The *max_connections* parameter, which specifies the number of connected applications allowed
- The *max_coordagents* parameter, which specifies the 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 coordinator agent requires global resource overhead, the greater the number of these agents the greater the chance that the upper limits of available database global resources will be reached. To prevent reaching the upper limits of available database global resources, you might set the value of *max_connections* higher than the value of *max_coordagents*.

When setting the values of *max_connections* and *max_coordagents*, you can also specify AUTOMATIC. There are two specific scenario's where using automatic will be advantageous:

- If you are confident that your system can handle all the connections that might be needed, but you want to limit the amount of global resources used (by limiting the number of coordinating agents), then you should specify AUTOMATIC for only the *max_connections* parameter. When *max_connections* is set as AUTOMATIC with a value greater than *max_coordagents* it means connection concentrator is enabled where any number of connections will be allowed (as long as there are enough system resources), however, the maximum number of coordinating agents will remain limited. This can be used to control both memory and disk constraints by limiting the number of concurrently executing applications.
- If you do not want to place artificial limits on your system whereby you limit the maximum number of connections and coordinating agents, but you know that your system requires or would benefit from a many-to-one relationship (between connected applications and application requests processed), then you should enable the connection concentrator and set both parameters to AUTOMATIC. When both parameters are set to AUTOMATIC, the database manager uses the values you specify as a ratio representing the ideal number of coordinating agents to connections.

Example

Consider the following setup:

- The *max_connections* parameter is set to AUTOMATIC with a value of 300
- The *max_coordagents* parameter is set to AUTOMATIC with a value of 100

The ratio of *max_connections* to *max_coordagents* is 300 to 100. The database manager will favor new coordinating agents as connections come in, therefore concentration is only applied when needed. The settings above translate into:

- Connections 1 to 100 will create new coordinating agents
- Connections 101 to 300 will not create new coordinating agents, they share the 100 agents already created
- Connections 301 to 400 will create new coordinating agents
- Connections 401 to 600 will not create new coordinating agents, they will share the 200 agents already created
- and so forth...

Note: This example assumes 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 any work, the coordinating agents will become inactive and might be terminated.

If the number of connections are reduced, but the amount of work being driven by the remaining connections is high, the number of coordinating agents may 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 number of connections to coordinating agents may not exactly represent the ratio that you specified. Agents might return to the agent pool to be reused before they are terminated.

If a finer granularity of control is needed, a simplified ratio is recommended. For example, the ratio of 300 to 100 above can be reduced to a 3 to 1 ratio. If *max_connections* is set to 3 (AUTOMATIC) and *max_coordagents* is set to 1 (AUTOMATIC) then for every 3 connections, 1 coordinating agent is allowed to be created.

Connection-concentrator improvements for client connections

For Internet applications with many relatively transient connections, or similar kinds of applications, the connection concentrator improves performance by allowing many more client connections to be processed efficiently. It also reduces memory use for each connection and decreases the number of context switches.

Note: The connection concentrator is enabled when the value of *max_connections* is greater than the value of *max_coordagents*.

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 formerly found only in DB2 Connect™ connection pooling. Both connection pooling and the connection concentrator are described in the DB2 Connect User's Guide. After the first connection, the connection concentrator reduces the connect time to a host. When a disconnection from a host is requested, the inbound connection is dropped, but the outbound connection to the host is kept in a pool. When a new request is made to connect to the host, the database manager tries to reuse an existing outbound connection from the pool.

Note: When applications use connection pooling or the connection concentrator, for best performance tune the parameters that control the size of the block of data that is cached. For more information, refer to the DB2 Connect User's Guide.

When using connection pooling, DB2 Connect is restricted to inbound TCP/IP and to outbound TCP/IP connections.

Usage examples

Example 1:

Consider an ESE environment with a single database partition in which, on average, 1000 users are connected to the database. At times, the number of users connected may be higher, therefore, a restriction of 1000 should not be set. The number of concurrent transactions is as high as 200, but never higher than 250. Transactions are short.

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

- *max_coordagents* is set to 250 to support the maximum number of concurrent transactions
- *max_connections* is set as AUTOMATIC with a value of 1000 to ensure support for any number of connections (in this example, any value greater than 250 will suffice to ensure that the connection concentrator is turned on)
- *num_poolagents* is set as the default, which should ensure database agents are available to service incoming client requests with little overhead of creating new ones

Example 2:

In a system similar to that of Example 1, where a seemingly unrestricted number of connections is allowed, it is possible to have the number coordinating agents to also increase based on the number of connections. For this example, assume that on average 1000 users are connected, with about 250 concurrent transactions running, however, the peak number of users is sometimes unpredictable. On occasion there could be 2000 users connected, for which probably 500 on average are expected to be executing work. 500 coordinating agents should not be allowed for most of the time, since normally only 1000 users are connected, and 250 coordinating agents is normally sufficient for them.

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

```
db2 update dbm cfg using MAX_COORDAGENTS 250 AUTOMATIC
db2 update dbm cfg using MAX_CONNECTIONS 1000 AUTOMATIC
```

This means that as the number of connections increase beyond 1000, additional coordinating agents will be created as needed, with a maximum determined by the total number of connections. With both parameters specifying a value and both set as AUTOMATIC, as the workload increases, the database manager maintains an approximate ratio of coordinating agents to connections.

Example 3:

In a system for which you do not want to enable the connection concentrator but you do want to limit the number of connected users, for example you want to allow for only 250 connected users at one time, set the database manager configuration parameters as follows:

- *max_connections* is set to 250.
- *max_coordagents* is set to 250.

Example 4:

In a system for which you do not want to enable the connection concentrator and you do not want to limit the number of connected users, set the database manager configuration parameters as follows:

```
db2 update dbm cfg using MAX_COORDAGENTS AUTOMATIC
db2 update dbm cfg using MAX_CONNECTIONS AUTOMATIC
```

Agents in a partitioned database

For partitioned database environments and environments with intra-partition parallelism enabled, each database partition (that is, each database server or node) 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 is

finished its work. The subagents can remain as associated agents in the pool and be used by the database manager for new requests from the application they are associated with or from new applications.

For partitioned database environments and environments with intra-partition parallelism enabled, the impact to performance and memory costs 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 both agents and subagents that can be kept associated with applications on a database partition, which is also called a node. 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 re-associated to applications, performance suffers.

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

  In addition, if the value of *num_poolagents* is too small, one application may fill the pool with associated subagents. Then when another application requires a new subagent and has no subagents in its associated agent pool, it will recycle inactive subagents from the agent pools of other applications. This behavior ensures that resources are fully utilized.
- Weigh concerns about having too few agents against the resource costs of allowing too many agents to be active at any given time.

  For example, if the value of *num_poolagents* is too large, associated subagents may sit unused in the pool for long periods of time, using database manager resources that are not available for other tasks.

  Note: When the connection concentrator is enabled, the number of agents specified by *num_poolagents* does not necessarily reflect the exact number of agents that may sit idle in the pool at any one time. The number of agents may be exceeded temporarily to handle occurrences of higher workload activity.

Other asynchronous processes and threads

In addition to the 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.

The database system monitor information

The DB2 database manager maintains data about its operation, its performance, and the applications using it. This data is maintained as the database manager runs, and can provide important performance and troubleshooting information. For example, you can find out:

- The number of applications connected to a database, their status, and which SQL and XQuery statements each application is executing, if any.

- Information that shows how well the database manager and database are configured, and helps you to tune them.
- When deadlocks occurred for a specified database, which applications were involved, and which locks were in contention.
- The list of locks held by an application or a database. If the application cannot proceed because it is waiting for a lock, there is additional information on the lock, including which application is holding it.

Because collecting some of this data introduces overhead on the operation of DB2, **monitor switches** are available to control which information is collected. To set monitor switches explicitly, use the UPDATE MONITOR SWITCHES command or the sqlmon() API. (You must have SYSADM, SYSCTRL, SYSMAINT, or SYSMON authority.)

You can access the data that the database manager maintains either by taking a snapshot or by using an event monitor.

Taking a snapshot

You can take a snapshot in one of the following three ways:
- Use the GET SNAPSHOT command from the command line.
- Write your own application, using the db2GetSnapshot() API call.
- Use snapshot table functions to return monitor data about a specific area of the database system.

Using an event monitor

An event monitor captures system monitor information after particular events have occurred, such as the end of a transaction, the end of a statement, or the detection of a deadlock. This information can be written to files or to a named pipe.

To use an event monitor:
1. Create its definition with the Control Center or the SQL statement CREATE EVENT MONITOR. This statement stores the definition in database system catalogs.
2. Activate the event monitor through the Control Center, or with the SQL statement:

   ```
   SET EVENT MONITOR evname STATE 1
   ```

   If writing to a named pipe, start the application reading from the named pipe before activating the event monitor. You can either write your own application to do this, or use db2evmon. Once the event monitor is active and starts writing events to the pipe, db2evmon will read them as they are being generated and write them to standard output.
3. Read the trace. If using a file event monitor, you can view the binary trace that it creates in either of the following ways:
   - Use the db2evmon tool to format the trace to standard output.
   - Click on the **Event Analyzer** icon in the Control Center on a Windows-based operating system to use a graphical interface to view the trace, search for keywords, and filter out unwanted data.

     Note: If the database system that you are monitoring is not running on the same machine as the Control Center, you must copy the event monitor file to

the same machine as the Control Center before you can view the trace. An alternative method is to place the file in a shared file system accessible to both machines.

Performance impact when using a deadlock event monitor

When a deadlock event monitor is active with the HISTORY option enabled, the general performance of a DB2 database system is impacted in the following ways:

- Memory used in the package cache for cached dynamic SQL and XQuery statements that are listed in the the statement history is not released until that particular statement history is no longer needed (that is, the current unit of work ends). This can cause the package cache size to grow due to the increased use of space in the cache that cannot be freed.
- There is a minor impact in system performance, resulting from the copying of statement information to the statement history list.
- There is an increased use of the DB2 system monitor heap, at each database partition, in order to keep a statement history list for each active application at that database partition. The amount of the increase depends on the number of statements executed in each unit of work by each application. A suggested computation for monitor heap follows:

```
If an event monitor is of type DEADLOCK
   and the WITH DETAILS HISTORY option is running,
add X*100 bytes times the maximum number of concurrent applications
   you expect to be running,
where X is the expected maximum number of statements in your
   application's unit of work.
If the event monitor is of type DEADLOCK
   and the WITH DETAILS HISTORY VALUES option is running,
also add X*Y bytes times the maximum number of concurrent applications
   you expect to be running,
where Y is the sum of the expected maximum size of parameter values being
   bound into your SQL and XQuery statements.
```

When a deadlock event monitor is active with the VALUES option enabled, the general performance of a DB2 database system is impacted in the following ways (in addition to the ones listed previously for the HISTORY option):

- There is a very minor impact in system performance resulting from the copying of statement information to the statement history list.
- There is an increased use of the DB2 system monitor heap at each database partition in order to keep a statement history list for each active application at that database partition. The increase depends on the number of data values used per statement as well as the number of statements executed in each unit of work by each application.
- The database manager maintains an extra copy of the data values which, depending on the size and number of the variables, may have an impact on performance.

The memory impact on the DB2 system monitor heap can become substantial when both the HISTORY and the VALUES options are specified for the deadlock event monitor. To decrease the impact, use these options only when they are needed. Another way to decrease the impact is to increase the configured size of the DB2 System Monitor heap at all database partitions prior to enabling the event monitor.

When an actual deadlock occurs and there is a deadlock event monitor active, system performance is impacted by the generation of the event monitor records.

The degree of impact and its duration depends on the number of applications and database partitions involved in the deadlock as well as the number of statements and data values in the relevant statement history lists.

Efficient SELECT statements

Because SQL is a flexible high-level language, you can write several different SELECT statements to retrieve the same data. However, the performance might vary for the different forms of the statement as well as for the different classes of optimization.

Consider the following guidelines for SELECT statements:

- Specify only columns that you need. Although it is simpler to specify all columns with an asterisk (*), unnecessary processing and return of unneeded columns results.
- Use predicates that restrict the answer set to only those rows that you require
- When the number of rows you need is significantly less than the total number of rows that might be returned, specify the OPTIMIZE FOR clause. This clause affects both the choice of access plans and the number of rows that are blocked in the communication buffer.
- When the number of rows to be retrieved is small, specify only the OPTIMIZE FOR *k* ROWS clause. You do not need the FETCH FIRST n ROWS ONLY clause. However, if *n* is large and you want the first *k* rows quickly with a possible delay for the subsequent *k* rows, specify both clauses. The size of the communication buffers is based on the lesser of *n* and *k*. The following example shows both clauses:

  ```
  SELECT EMPNAME, SALARY FROM EMPLOYEE
     ORDER BY SALARY DESC
     FETCH FIRST 100 ROWS ONLY
     OPTIMIZE FOR 20 ROWS
  ```

- To take advantage of row blocking, specify the FOR READ ONLY or FOR FETCH ONLY clause to improve performance. In addition, concurrency improves because exclusive locks are never held on the rows retrieved. Additional query rewrites can also occur. Specifying the FOR READ ONLY or FOR FETCH ONLY clause as well as the BLOCKING ALL BIND option can improve the performance of queries against nicknames in a federated system in a similar way.
- For cursors that will be updated with positioned updates, specify the FOR UPDATE OF clause to allow the database manager to choose more appropriate locking levels initially and avoid potential deadlocks. Note that FOR UPDATE cursors cannot take advantage of row blocking.
- For cursors that will be updated with searched updates, you can avoid deadlocks and still allow row blocking by forcing U locks on affected rows with the FOR READ ONLY and the USE AND KEEP UPDATE LOCKS clauses.
- Avoid numeric data type conversions whenever possible. When comparing values, it might be more efficient to use items that have the same data type. If conversions are necessary, inaccuracies due to limited precision and performance costs due to run-time conversions might result.

  If possible, use the following data types:

  – Character instead of varying character for short columns
  – Integer instead of float or decimal
  – Datetime instead of character
  – Numeric instead of character

- To decrease the possibility that a sort operation will occur, omit clauses or operations such as DISTINCT or ORDER BY if such operations are not required.
- To check for existence of rows in a table, select a single row. Either open a cursor and fetch one row or perform a a single-row (SELECT INTO) selection. 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  TABLENAME
  ```

  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 as predicates.
- Consider using an IN list if the same column appears in multiple predicate clauses. For large IN lists 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 the join predicate to allow the join to be processed more efficiently. Indexes also benefit UPDATE and DELETE statements that contain SELECT statements that access several tables.
- If possible, avoid using expressions or OR clauses with join predicates because the database manager cannot use some join techniques. As a result, the most efficient join method may not be chosen.
- In a partitioned database environment, if possible ensure that both of the tables joined are partitioned on the join column.

Chapter 15. The Governor utility

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

A governor instance consists of a front-end utility and one or more daemons. Each instance of the governor that you start 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.

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.

Each governor daemon collects information about the applications that run against the database. It then checks this information against the rules that you specify in the governor configuration file for this database.

The governor manages application transactions as specified by the rules in the configuration file. For example, applying a rule might indicate that an application is using too much of a particular resource. The rule would specify the action to take, such as to change the priority of the application or force 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. To review the actions, you query the log files.

Starting and stopping the governor

The governor utility monitors applications that connect to a database and changes their behavior according to rules that you specify in a governor configuration file for that database.

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

To start or stop the governor, you must have *sysadm* or *sysctrl* authorization.

To start or stop the governor:

1. To start the governor, execute the *db2gov* command at the DB2 command line. Enter the following required parameters:
   - START *database_name*

     The database name that you specify must match the name of the database in the configuration file that you specify. An error is returned if the names are

not the same. Note that if a governor is running for more than one database, daemons will be started for each database.

- *config_file_name*

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

- *log_file_name*

  The base name of the log file for this governor. On a partitioned database, the database partition number is added for each database partition where a daemon runs for this instance of the governor.

To start the governor on a single database partition for a partitioned database, add the *nodenum* option.

For example, to start the governor for a database called *sales* on only node 3 of a partitioned database with a configuration file called *salescfg* and a log file called *saleslog*, enter the following command:

```
db2gov START sales nodenum 3 salescfg saleslog
```

To start the governor on all database partitions of the *sales* database, enter the following command:

```
db2gov START sales  salescfg saleslog
```

2. To stop the governor, enter the *db2gov* command with the STOP option.

   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
   ```

The Governor daemon

When the governor daemon starts, either when you execute by the `db2gov` utility or when it wakes up, it runs the following task loop.

1. It 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. It requests snapshot information about resource-use statistics for each application and agent that is working on the database.

   Note: On some platforms, the CPU statistics are not available from the DB2 Monitor. In this case, the account rule and the CPU limit are not available.
3. It checks the statistics for each application against the rules in the governor configuration file. If a rule applies to an application, the governor performs the specified action.

   Note: The governor compares accumulated information with the values defined in the configuration file. This means that if the configuration file is updated with new values that an application may have already breached, the governor rules concerning that breach are applied immediately to the application at the next governor interval.
4. It writes a record in the governor log file for any action that it takes.

Note: The governor cannot be used to adjust agent priorities if the *agentpri* database manager configuration parameter is anything other than the system default.

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

When the governor encounters an error or stop signal, it does cleanup processing before it ends. Using a list of applications whose priorities have been set, the 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 nondefault priorities after the governor ends. If an error occurs, the governor writes a message to the administration notification log to indicate that it ended abnormally.

Note: 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.

Configuring the Governor

To configure the Governor, you create a configuration file that determines the database that an instance of the Governor monitors and how it manages queries.

The configuration file consists of a set of rules. The first three rules specify the database to monitor, the interval at which to write log records, and the interval at which to wake up for monitoring. The remaining rules specify how to monitor the database server and what actions to take in specific circumstances.

To create a Governor configuration file:

1. In a directory that is mounted or available from all database partitions, create an ASCII file with a descriptive name. For example, the configuration file for a governor instance that monitors the **sales** database might be called *govcfgsales*.
2. Open the file in any text editor and enter configuration information and action conditions.

   End each rule with a semicolon (;). The following configuration information is recommended:

   - dbname: The name or alias of the database to be monitored.
   - account: The number of minutes after which the governor instance writes CPU usage statistics to its log file.
   - interval: The number of seconds after which the governor daemon wakes up to monitor activity. If you do not specify an interval, the default value of 120 seconds is used.

   For example, the first three rules in the configuration file might look like this:

   ```
   { Wake up once a second, the database name is sales,
    do accounting every 30 minutes.  }
   interval 1; dbname sales; account 30;
   ```

   Add rules that specify the conditions to monitor and the action to take if the rule evaluates to true. For example, you might add a rule that limits to an hour the amount of time that a unit of work (UOW) can run before being forced to disconnect from the database, as follows:

```
setlimit uowtime 3600 action force;
```

3. Save the file.

The governor configuration file

When you start the governor, you specify the configuration file that contains the rules that govern applications running against the database. The governor evaluates each rule and acts as specified when the rule evaluates to true.

If your rule requirements change, you edit the configuration file without stopping the governor. Each governor daemon detects that the file has changed, and rereads it.

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

The configuration file consists of three required rules that identify the database to be monitored, the interval at which log records are written, and the sleep interval of the governor daemons. Following these parameters, the configuration file contains a set of optional application-monitoring rules and actions. The following comments apply to all rules:

- Delimit comments inside { } braces.
- Most entries can be specified in uppercase, lowercase, or mixed case characters. The exception is the application name, specified as an argument to the `applname` rule, which is case sensitive.
- Each rule ends with a semicolon (;).

Required rules

The following rules specify the database to be monitored and the interval at which the daemon wakes up after each loop of activities. Each of these rules is specified only once in the file.

dbname
The name or alias of the database to be monitored.

account *nnn*
Account records are written containing CPU usage statistics for each connection at the specified number of minutes.

Note: This option is not available in the Windows® environment.

If a short connect 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 then restarted, CPU usage may be reflected in two log records; these can be identified through the application IDs in the log records.

interval
The interval, in seconds, at which the daemon wakes up. If you do not specify an interval, the default value, 120 seconds, is used.

Rules that govern actions

Following the required rules, you can add rules that specify how to govern the applications. These rules are made of smaller components called rule clauses. If used, the clauses must be entered in a specific order in the rule statement, as follows:

1. **desc** (optional): a comment about the rule, enclosed in quotation marks
2. **time** (optional): the time during the day when the rule is evaluated
3. **authid** (optional): one or more authorization IDs under which the application executes statements
4. **applname** (optional): the name of the executable or object file that connects to the database. This name is case sensitive. The application name must be surrounded by double quotes if the application contains spaces.
5. **setlimit**: the limits that the governor checks. These can be one of several, for example, CPU time, number of rows returned, or idle time..
6. **action** (optional): the action to take if 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 against the application can include reducing its agent priority, forcing it to disconnect from the database, or setting scheduling options for its operations.

You combine the rule clauses to form a rule, using each clause only once in each rule, and end the rule with a semicolon, as shown in the following examples:

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

desc "Slow down the use of db2 CLP by the novice user"
authid novice
applname db2bp.exe
setlimit cpu 5 locks 100 rowssel 250;
```

If more than one rule applies to an application, all are applied. Usually, the action associated with the rule limit encountered first is the action that is applied first. An exception occurs you specify if -1 for a clause in a rule. In this case, the value specified for the clause in the subsequent rule can only override the value previously specified for the *same* clause: other clauses in the previous rule are still operative. For example, one rule uses the `rowssel 100000 uowtime 3600` clauses to specify that the priority of an application is decreased either 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` is still valid.

Order of rule application

The governor processes rules in the configuration file from the top of the file to the bottom. However, if a later rule's setlimit clause is more relaxed than a preceding rule, the more restrictive rule still applies. For example, in the following configuration file, `admin` will be limited to 5000 rows despite the later rule because the first rule is more restrictive.

```
desc "Force anyone selecting 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 rule that occurs earlier in the file, you can specify the -1 option 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 selecting 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;
```

Governor rule elements

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 clauses must be specified in the order shown. In the clause descriptions, [ ] indicates an optional clause.

Optional beginning elements

[desc] Specifies a text 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 valid 24 hours a day.

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

[applname]
Specifies the name of the executable (or object file) that makes the connection to the database.

Multiple application names must be separated by a comma (,), for example, `applname db2bp, batch, geneprog`. If this clause does not appear in a rule, 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 you want to govern is uniquely identified by the first 20 characters of its application name; otherwise, an unintended application may be governed.

   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 can only

be -1 or greater than 0 (for example, `cpu -1 locks 1000 rowssel 10000`). At least one of the limits (cpu, locks, rowsread, uowtime) must be specified, and any limit not specified by the rule is not limited by that particular rule. The governor can check the following limits:

cpu *nnn*
Specifies the number of CPU seconds that can be consumed by an application. If you specify `-1`, the governor does not limit the application's CPU usage.

locks *nnn*
Specifies the number of locks that an application can hold. If you specify `-1`, the governor does not limit the number of locks held by the application.

rowssel *nnn*
Specifies the number of rows that are returned to the application. This value will only be non-zero at the coordinator node. If you specify `-1`, the governor does not limit the number of rows that can be selected. The maximum value that can be specified for *nnn* is 4 294 967 298.

uowtime *nnn*
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` (Database System Monitor Switch) 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 you turn off the switches in the database manager configuration file, then it is turned off for the entire instance, and governor will no longer receive this information.

idle *nnn*
Specifies the number of idle seconds allowed for a connection before a specified action is taken. 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 EDUs not visible to the governor. These database connections will appear to be idle and might exceed the idle time limit. To prevent the governor from taking action against these utilities, you can specify -1 for them through the authorization id which invoked them. For example, to prevent the governor from acting against utilities run by authorization id DB2SYS, specify "authid DB2SYS setlimit idle -1".

rowsread *nnn*
Specifies the number of rows an application can select. If you specify `-1`, there is no limit on the number of rows the application can select. The maximum value that can be specified for *nnn* is 4 294 967 298.

Note: This limit is not the same as rowssel. The difference is that rowsread is the count of the number of rows that had to be read in

order to return the result set. The number of rows read includes reads of the catalog tables by the engine and may be diminished when indices are used.

Action clauses

[action]

Specifies the action to take if one or more of the specified limits is exceeded. You can specify the following actions.

Note: 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.

nice *nnn*

Specifies a change to the relative priority of agents working for the application. Valid values are from -20 to +20.

For this parameter to be effective:

- On UNIX -based platforms, the *agentpri* database manager parameter must be set to the default value; otherwise, it overrides the priority clause.
- On Windows platforms, the *agentpri* database manager parameter and *priority* action may 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 will disassociate itself from its outbound correlator (if it is associated with one) and set 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.

force Specifies to force the agent that is servicing the application. (Issues a FORCE APPLICATION to terminate the coordinator agent.)

Note: In multi-partition database environments, the force action is only carried out when the governor daemon is running on the application's coordinating database partition. Therefore if a governor daemon is running on database partition A and a limit is exceeded for some application whose coordinating database partition is database partition B, then the force action is skipped.

schedule [class]

Scheduling improves the priorities of the agents working on the applications with the goal of minimizing the average response times while maintaining fairness across all applications.

The governor chooses the top applications for scheduling based on the following three criteria:

- The application holding the most locks

  This choice is an attempt to reduce the number of lockwaits.
- The oldest application

- The application with the shortest estimated remaining running time.

  This choice is an attempt to allow as many short-lived statements as possible to complete during the interval.

The top three applications in each criteria 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 applications are given the second highest priority and the third-ranked applications are given the third highest priority. If a single application is ranked in the top three in more than one of the criteria, 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 running time, it is given the highest priority for the first criterion, and the fourth ranked application with the shortest estimated remaining running time is given the third highest priority for that criterion.

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

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` (Database System Monitor Switch) 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, then it is turned off for the entire instance, and governor will no longer receive this information. The schedule action can:

- Ensure that applications in different groups each get time without all applications splitting time evenly.

  For instance, if 14 applications (three short, five medium, and six long) are running at the same time, they may 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 instance, 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, 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 included with the action, 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 DB2 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, no *applname* or *authid* clauses are supplied, and the *setlimit* clause causes no restrictions.

Note: If a limit is exceeded and the action clause is not specified, the governor reduces the priority of agents working for the application.

Example of a Governor configuration file

The following example shows a governor configuration file that sets several rules with actions:

```
 Wake up once a second, the database name is ibmsampl,
 do accounting every 30 minutes.  }
interval 1; dbname ibmsampl; account 30;

desc "CPU restrictions apply 24 hours a day to everyone"
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 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 dept"
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 db2 CLP by the novice user"
authid novice
applname db2bp.exe
setlimit cpu 5 locks 100 rowssel 250;

desc "During day hours 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 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 "Some people should not be limited -- database administrator
  and a few others.  As 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;

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;
```

Governor log files

Whenever a governor daemon performs an action, it writes a record to its log file. Actions include the following:

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

Each governor daemon has a separate log file. Separate log files prevent file-locking bottlenecks that might result when many governor daemons write to the same file at the same time. To merge the log files together and query them, use the `db2govlg` utility.

The log files are stored in the `log` subdirectory of the `sqllib` directory, except on Windows Operating systems, where the `log` subdirectory is under the instance directory. You provide the base name for the log file when you start the governor with the `db2gov` command. Make sure that the log file name contains the database

name to distinguish log files on each database partition that is governed. To ensure that the filename is unique for each governor in a partitioned database environment, the database partition number where 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 NodeNum RecType Message
```

Note: 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 *NodeNum* field indicates the number of the database partition on which the governor is running.

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

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

Some of these values are described in more detail below.

START
The START record is written when the governor is started. It has the following format:

```
Database = <database_name>
```

STOP The STOP record is written when the governor is stopped. It has the following format:

```
Database = <database_name>
```

FORCE
The FORCE record is written out whenever the governor determines that an application is to be forced as required by a rule 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 coordinating 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 exceeded. This can contain the following values:

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

NICE The NICE record is written when the priority of an application is changed via a priority action specified 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 exceeded. This can contain the following values:

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

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

WARNING
The WARNING record is written to the governor log 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 (″SQL1661 No data was returned″).
- A snapshot call returned a negative SQLCODE that was not -1224 (″SQL 1224N A database agent could not be started to service a request, or was terminated as a result of a database system shutdown or a force command″) or -1032 (″SQL1032N No start database manager command was issued″). These return codes occur when a previously active instance has been brought down.

- In a UNIX environment, an attempt was made to install a signal handler and the attempt failed.

ACCOUNT

The ACCOUNT record is written to the governor log in the following situations:

- The value of agent_usr_cpu or agent_sys_cpu for this application has changed since the last ACCOUNT record was written for this application.
- An application is found to be 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>

SCHEDGRP

The SCHEDGRP record is written whenever:

- An application is added to a scheduling group
- 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 exceeded. This can contain the following values:

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

Because standard values are written, you can query the log files for different types of actions. The *Message* field provides other nonstandard information that varies according to the value under the *RecType* field. For instance, a `FORCE` or `NICE` record indicates application information in the *Message* field, while an `ERROR` record includes an error message.

An example log file is as follows:

```
1995-12-11 14.54.52    0 START      Database = TQTEST
1995-12-11 14.54.52    0 READCFG    Config = /u/db2instance/sqllib/tqtest.cfg
1995-12-11 14.54.53    0 ERROR      SQLMON Error: SQLCode = -1032
1995-12-11 14.54.54    0 ERROR      SQLMONSZ Error: SQLCode = -1032
```

Governor log file queries

Each governor daemon writes to its own log file. You can use `db2govlg` utility to query the log file. You can list the log files for a single database partition, or for all database partitions, sorted by date and time. You can also query on the basis of the

RecType log field. The syntax for `db2govlg` is as follows:

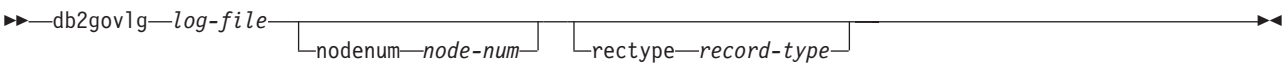


Figure 19. Syntax for db2govlg

The parameters are as follows:

log-file The base name of the log file (or files) that you want to query.

nodenum *node-num*
The node number of the database partition on which the governor is running.

rectype *record-type*
The type of record that you want to query. The record types are:

- START
- READCFG
- STOP
- FORCE
- NICE
- ERROR
- WARNING
- ACCOUNT

There are no authorization restrictions for using this utility. This allows all users to query whether the governor has affected their application. If you want to restrict access to this utility, you can change the group permissions for the `db2govlg` file.

Chapter 16. 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), and should be performed against your application in order to determine current performance and improve it. If the application code has been written as efficiently as possible, additional performance gains might be realized from tuning the database and database manager configuration parameters. You can even tune application parameters to meet the requirements of the application better.

You run different types of benchmark tests to discover specific kinds of information:

- A *transaction per second* benchmark determines the throughput capabilities of the database manager under certain limited laboratory conditions.
- An *application* benchmark tests the same throughput capabilities under conditions that are closer production conditions.

Benchmarking tuning configuration parameters is based upon these "real-world" conditions, and requires repeatedly running SQL taken from your application with varying parameter values until the application runs as efficiently as possible.

The benchmarking methods described here are oriented toward tuning configuration parameters. However, the same basic technique can be used for tuning other factors that affect performance, such as:

- SQL statements
- Indexes
- Table space configuration
- Application code
- Hardware configuration.

Benchmarking is helpful in understanding how the database manager responds under varying 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 product.

Benchmark testing methods

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 benchmarking 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 when necessary.

Characteristics of good benchmarks or measurements include:

- Tests are repeatable.
- Each iteration of a test starts in the same system state.

- No other functions or applications are active in the system unless the scenario includes some amount of other activity going on in the system.

  Note: Started applications use memory even when they are minimized or idle. This increases the probability that paging will skew the results of the benchmark and violates the repeatability rule.
- The hardware and software used for benchmarking match your production environment.

For benchmarking, you create a scenario and then applications in this scenario several times, capturing key information during each run. Capturing key information after each run is of primary importance in determining the changes that might improve performance of both the application and the database.

Benchmark preparation

Complete the logical design of the database against which the application runs before you start performance benchmarking. Set up and populate tables, views, and indexes. Normalize tables, bind application packages, and populate tables with realistic data.

You should also have determined the final physical design of the database. Place database manager objects in their final disk locations, size log files, determining the location of work files and backup, and test backup procedures. In addition, check packages to make sure that performance options such as row blocking are enabled when possible.

You should have reached a point in application programming and testing phases that will enable you to create your benchmark programs. Although the practical limits of an application might be revealed during the benchmark testing, the purpose of the benchmark described here is to measure performance, not to detect defects or abends.

Your benchmarking test program will need to run in as accurate a representation of the final production environment as possible. Ideally, it should run on the same model of server with the same memory and disk configurations. This is especially important when the application will ultimately involve large numbers of users and large amounts of data. The operating system itself and any communications or file-serving facilities used directly by the benchmark should also have been tuned.

Make sure that you run benchmark tests with a production-size database. An individual SQL statement should return as much data and require as much sorting as in production. This rule ensures that the application will test representative memory requirements.

SQL statements to be benchmarked should be either *representative* or *worst-case*, as described below:

Representative SQL
: Representative SQL includes those statements that are executed during typical operations of the application being benchmarked. The statements that are selected will depend on the nature of the application. For example, a data-entry application might test an INSERT statement, while a banking transaction might test a FETCH, an UPDATE, and several INSERTs. Consider the frequency of execution and volume of data processed by the

statements chosen average. If the volumes are excessive, consider the statements under the *worst-case* category, even if they are typical SQL statements.

Worst-case SQL

Statements falling in this category include:

- Statements that are executed frequently.
- Statements that have high volumes of data being processed.
- Statements that are time-critical.

  For example, an application that is run when a telephone call is received from a customer and the statements must be run to retrieve and update the customer's information while the customer is waiting.
- Statements with the largest number of tables being joined or with the most complex SQL in the application.

  For example, a banking application that produces combined customer statements of monthly activity for all their different types of accounts. A common table may list customer address and account numbers; however, several other tables must be joined to process and integrate all of the necessary account transaction information. Multiply the work necessary for one account by the several thousand accounts that must be processed during the same period, and the potential time savings drives the performance requirements.
- Statements that have a poor access path, such as one that is not executed very often and is not supported by the indexes that have been created for the tables involved.
- Statements that have a long elapsed time.
- A statement that is only executed at application initialization but has disproportionate resource requirements.

  For example, an application that generates a list of account work that must be processed during the day. When the application is started, the first major SQL statement causes a 7-way join, which creates a very large list of all the accounts for which this application user is responsible. The statement might only be run a few times per day, but takes several minutes to run when it has not been tuned properly.

Benchmark test creation

Consider a variety of factors when you design and implement a benchmark program. Because the main purpose of the program is to simulate a user application, the overall structure of the program varies. You might use the entire application as the benchmark and simply introduce a means for timing the SQL statements 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 might include these statements alone in the benchmark program along with the necessary CONNECT, PREPARE, OPEN, and other statements and 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 time interval. The ratio of the number of statements executed and this time interval would give the throughput for the application. Another option is simply to determine the time required to execute individual SQL statements.

For all benchmark testing, you need an efficient timing system to calculate the elapsed time, whether for individual SQL statements or the application as a whole. To simulate applications in which individual SQL statements are executed in isolation, it might be important to track times for CONNECT, PREPARE, and COMMIT statements. However, for programs that process many different statements, perhaps only a single CONNECT or COMMIT is necessary, and focusing on just the execution time for an individual statement might be the priority.

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 is not using the CPU to its full capacity. A benchmark program should allow 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 whole answer set might be input for another program, so that none of the rows from the first application are sent as output. Formatting data for screen output usually has high CPU cost and might not reflect user need. To provide an accurate simulation, a benchmark program should reflect the row handling of the specific application. If rows are sent to an output device, inefficient formatting could consume the majority of CPU processing time and misrepresent the actual performance of the SQL statement itself.

The db2batch Benchmark Tool: A benchmark tool (`db2batch`) is provided in the `bin` subdirectory of your instance `sqllib` directory. This tool uses many of guidelines for creating a benchmark program. This tool can read SQL statements from either a flat file or standard input, dynamically describe and prepare the statements, and return an answer set. It also allows you to control the size of the answer set, as well as the number of rows that are sent from this answer set to an output device.

You can specify the level of performance-related information supplied, including the elapsed time, CPU and 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. For syntax and options, type `db2batch -h` on a command line.

Benchmark test execution

For one type of database benchmark, 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 executing the application through several iterations (for example, 20 or 30 times) with the same parameter value to get an average timing, which shows the effect of parameter changes more clearly.

When you run the benchmark, the first iteration, which is called a warm-up run, should be considered a separate case from the subsequent iterations, which are called normal runs. Because the warm-up run includes some start-up activities, such as initializing the buffer pool, and consequently, takes somewhat longer than normal runs. Although the information from the warm-up run might be realistically valid, it is not statistically valid. When you calculate the average timing or CPU for a specific set of parameter values, use only the results from normal runs.

You might consider using the Configuration Advisor to create the warm-up run of the benchmark. The questions that the Configuration Advisor asks can provide insight into some things to consider when you adjust the configuration of your environment for the normal runs during your benchmark activity. You can start the Configuration Advisor from the Control Center or by executing the db2 autoconfigure command with appropriate options.

If benchmarking uses individual queries, ensure that you minimize the potential effects of previous queries by flushing the buffer pool. To flush the buffer pool, read a number of pages that irrelevant to your query and to fill the buffer pool.

After you complete the iterations for a single set of parameter values, you can change a single parameter. However, 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, make sure that the same values for the statistics are used for every iteration.
- The data used in the tests must be consistent if it is updated by the tests. 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 allows you to restore only the data that has been affected. REORG and RUNSTATS utilities should be run against the tables and indexes that contain this data.
- To return the application to its original state, re-bind it to the database.

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

Step 1 Leave the database and database manager tuning parameters at their **default** values except for:

- Those parameters significant to the workload and the objectives of the test. (You rarely have enough time to perform benchmark testing to tune all of the parameters, so you may want to start by using your best guess for some of the parameters and tune from that point.)
- Log sizes, which should be determined during unit and system testing of your application.
- Any parameters that must be changed to enable your application to run (that is, the changes needed to prevent negative SQL return codes from such events as running out of memory for the statement heap).

Run your set of iterations for this initial case and calculate the average timing or CPU.

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

Step 3 Run another set of iterations and calculate the average timing or CPU.

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.

Note: If you were to graph the performance results, you would be looking for the point where the curve begins to plateau or decline.

You can write a driver program to help you with your benchmark testing. This driver program could be written using a language such as REXX or, for Linux and UNIX platforms, using shell scripts.

This driver program would execute the benchmark program, pass it 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/consolidate the test results. These driver programs can be flexible enough that they could be used to run the entire set of benchmark tests, analyze the results, and provide a report of the final and best parameter values for the given test.

Benchmark test analysis example

Output from the benchmark program should include an identifier for each test, the iteration of the program execution, the statement number, and the timing for the execution.

A summary of benchmarking results after a series of measurements 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 20. Benchmark Sample Results

Note: The data in the above report is shown for illustration purposes only. It does **not** represent measured results.

Analysis shows that the CONNECT (statement 01) took 1.34 seconds, the OPEN CURSOR (statement 10) took 2 minutes and 8.15 seconds, the FETCHES (statement 15) returned seven rows with the longest delay being .28 seconds, the CLOSE CURSOR (statement 20) took .84 seconds, and the CONNECT RESET (statement 99) took .03 seconds.

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

Sample output for a benchmark report might be:

```
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 21. Benchmark Sample Timings Report

Note: The data in the above report is shown for illustration purposes only. It does **not** represent any measured results.

Chapter 17. 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, MQTs, clustering dimensions, or database partitions to create for a complex workload can be quite daunting. The Design Advisor identifies all of the objects needed to improve the performance of your workload. Given a set of SQL statements in a workload, the Design Advisor will generate recommendations for:

- New indexes
- New materialized query tables (MQTs)
- Determine the addition of clustering indexes
- Conversion to multidimensional clustering (MDC) tables
- Redistribution of tables
- Deletion of indexes and MQTs unused by the specified workload (through the GUI tool)

You can have the Design Advisor implement some or all of these recommendations immediately or schedule them for a later time.

Using either the Design Advisor GUI or the command-line tool, the Design Advisor can help simplify the following tasks:

Planning for or setting up a new database

While designing your database use the Design Advisor to:

- Generate design alternatives in a test environment of a partitioned database environment, and of indexes, MQTs, and MDC tables.
- For partitioned database environments, you can use the Design Advisor to:
  - Determine the database partitioning strategy before loading data into a database.
  - Assist in migrating from a single-partition DB2 database to a multiple-partition DB2 database.
  - Assist in migrating from another database product to a multiple-partition DB2 database.
- Evaluate indexes, MQTs, MDC tables, or database partitioning strategies that have been generated manually.

Workload performance tuning

After your database is set up, you can use the Design Advisor to:

- Improve performance of a particular statement or workload.
- Improve general database performance, using the performance of a sample workload as a gauge.
- Improve performance of the most frequently executed queries, for example, as identified by the Activity Monitor.
- Determine how to optimize the performance of a new key query.
- Respond to Health Center recommendations regarding shared memory utility or sort heap problems in a sort-intensive workload.
- Find objects that are not used in a workload.

Design Advisor output

If you use the Design Advisor GUI, you can view, save, or implement the recommendations from within the Design Advisor. If you run the Design Advisor from the command line, the output is printed to stdout by default, and saved in the ADVISE_TABLE and ADVISE_INDEX tables:

- The ADVISE_TABLE contains USE_TABLE='Y' for MQT, MDC tables, and database partitioning strategy recommendations.

  The MQT recommendations can also be found in the ADVISE_MQT table; the MDC recommendations can also be found in the ADVISE_TABLE table; and the database partitioning strategy recommendations can also be found in the ADVISE_PARTITION table. The RUN_ID value in these tables corresponds to the START_TIME value in the ADVISE_INSTANCE table for each execution of the Design Advisor.
- The ADVISE_INDEX table contains USE_INDEX='Y' or 'R' for index recommendations.

The ADVISE_INSTANCE table is also updated with one row each time that the Design Advisor runs:

- The START_TIME field and the END_TIME field show the start and stop times of the utility, respectively.
- The STATUS field will contain 'COMPLETED' if the utility ended successfully.
- The MODE field indicates whether the -m option was used.
- The COMPRESSION field indicates the type of compression used.

You can save the Design Advisor recommendations to a file using the -o option. The saved Design Advisor output consists of the following elements:

- CREATE STATEMENTS given for new indexes, MQTs, database partitioning strategies, and MDC tables.
- REFRESH statements for MQTs.
- RUNSTATS commands for new objects.
- Existing MQTs and indexes will appear in the recommended script if they were and are used to execute the workload.

  Note: The COLSTATS column of the ADVISE_MQT table contains the column statistics for an MQT. The statistics are in 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>
```

  The base table on which an index is defined is also added. Ranking of indexes and MQTs can be done using the benefit value. You can also rank MQTs using

benefit - 0.5*overhead and indexes using benefit-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, as well as the list of tables, indexes and MQTs that were used by the statement.

Note: The original spacing in the SQL text is preserved in this output but the SQL text is split into 80 character commented lines for readability.
MDC and partitioning is not explicitly shown in this XML output. 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>TPCD </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 tpcd.customer, tpcd.orders,
-- tpcd.lineitem where o_orderkey in( select
-- l_orderkey from tpcd.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>TPCD </schema>
--</identifier>
--<identifier>
--<name>CUSTOMER</name>
--<schema>TPCD </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>
```

Note that the XML structure can contain more than one column. For each column, the column cardinality (that is, the number of values in the column) is shown, and optionally, the high2 and low2 keys.

When MQT, database partitions, or MDC recommendations are provided, the relevant ALTER TABLE stored procedure call command is placed in the ALTER_COMMAND column of the ADVISE_TABLE.

Note: The ALTER TABLE stored procedure call command may not succeed due to restrictions on the table for the ALTOBJ stored procedure.

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 command statements into a single RUNSTATS invocation on the new or modified objects.
- Providing more usable object names than the system-generated IDs.
- Removing or commenting out any DDL for objects that you do not want to implement immediately.

Using the Design Advisor

You can run the Design Advisor from the command line or by using the Design Advisor GUI in the Control Center.

- **Control Center method:**
  1. "Defining a workload for the Design Advisor" on page 153.
  2. Open the Control Center.
  3. Run the Design Advisor from the Control Center by right-clicking on the appropriate database or on a particular index in the database, and selecting Design Advisor from the pop-up menu.
  4. Once the Design Advisor has finished generating recommendations, you can choose to save these recommendations in a report, or you can have the Design Advisor implement some or all of the recommendations for you.
- **Command line method:**
  1. "Defining a workload for the Design Advisor" on page 153.
  2. Run the db2advis command on the workload.

Note: If the statistics on your database are not current, the generated recommendations will be less reliable.

3. Interpret the output from db2advis and make any necessary modifications.
4. Implement the selected Design Advisor recommendations.

Defining a workload for the Design Advisor

A *workload* is a set of SQL statements that the database manager has to process during a given period of time. For example, during one month your database manager may have to process 1000 INSERTs, 10000 UPDATEs, 10000 SELECTs, and 1000 DELETEs. The Design Advisor analyzes a specified workload and considers factors such as the type of workload statements, the frequency with which a particular statement occurs, and characteristics of your database to generate recommendations that minimize the total cost to run the workload.

From the Design Advisor GUI workload page, you can create a new workload file, or modify a previously existing workload file. You can import statements into the file from several sources:

- A delimited text file
- An Event Monitor table
- Query Patroller historical data tables by using the -qp option from the command line
- Explained statements in the EXPLAINED_STATEMENT table
- Recent SQL statements that have been 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 your SQL statements, you can add, change, modify, or remove statements and modify their frequency.

From the command line, run the Design Advisor using:

- A single SQL statement that you enter in-line with the command
- A set of dynamic SQL statements captured in a DB2 snapshot
- A set of SQL statements contained in a workload file.
- To run the Design Advisor on dynamic SQL statements:
  1. Reset the database monitor with the following command:
     ```
     db2 reset monitor for database database-name
     ```
  2. Wait for an appropriate interval of time for the execution of dynamic SQL statements against the database.
  3. Issue the db2advis command with the -g option. If you want to save the dynamic SQL statements in the ADVISE_WORKLOAD table for later reference, use the -p option as well.
- To run the Design Advisor on a set of SQL statements contained 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 above.
  2. Set the frequency of the statements in the workload. Every statement in the workload file is assigned a frequency of 1 by default. The frequency of an

SQL statement represents the number of times 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, while another SELECT statement occurs 10 times. To represent the relative frequency of these two statements, you could assign the first SELECT statement a frequency of 10, while the second select statement has a frequency of 1. You can manually change the frequency or weight that a particular statement has 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. Run the db2advis command using the -i option followed by the name of the workload file.

- To run the Design Advisor on a workload contained in the ADVISE_WORKLOAD table, run db2advis with the -w option followed by the workload name.

Using the Design Advisor to migrate from a single-partition to a multiple-partition database

You can use the Design Advisor to help you migrate from a single-partition to a multiple-partition database. The Design Advisor can provide you with recommendations for distributing your data and, at the same time, provide you with recommendations for new indexes, materialized query tables (MQTs), and multi-dimensional clustering (MDC) tables.

1. Registering a DB2 product or feature license key using the db2licm command
2. Create at least one table space in a multiple-partition database partition group.

   Note: Because the Design Advisor can only recommend data redistribution to existing table spaces, the table spaces that you want the Design Advisor to consider must exist in the partitioned database before the Design Advisor is run.
3. Run the Design Advisor, with the database partitioning feature selected in the Design Advisor GUI, or with the partitioning option specified for the db2advis command.
4. If you are using the Design Advisor in the Control Center, you can implement the database partitioning recommendations automatically. If you are using the db2advis command you will need to modify the db2advis output file slightly before running the DDL statements generated by the Design Advisor.

Design Advisor limitations and restrictions

1. **Restrictions on index recommendations**
   - Indexes recommended on materialized query tables (MQTs) are to improve the workload performance as opposed to REFRESH TABLE performance. Also if updates, inserts, or deletes are not included in the workload, the performance of changing (for example, updating) the MQT would not be included for IMMEDIATE MQTs.
   - The clustering RID index is only recommended when multidimensional clustering is to be selected. The advisor will include RID clustering indexes as an option instead of creating an MDC structure for a table.
2. **Restrictions on MQT recommendations**

- The Design Advisor will not recommend incremental MQTs. If you want to create incremental MQTs, you can take REFRESH IMMEDIATE MQTs and convert these to incrementals with your choice of staging tables.
- Indexes recommended for MQTs are designed to improve workload performance and not MQT refresh performance.
- If updates, inserts, or deletes are not included in the specified workload, the performance impact of updating a recommended REFRESH IMMEDIATE MQT is not considered. It is recommended 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.

3. **Restrictions on MDC recommendations**
   - Existing tables must be populated with a representative set of data, otherwise the Design Advisor will not consider MDC for the table. A minimum of two to three dozen megabytes of data is recommended. Tables smaller than 12 extents will always be excluded.
   - MDC recommendations for new MQTs will not be considered unless the sampling option, -r, is used with the command, or MQT sampling is selected in the GUI tool.
   - The Design Advisor does not make MDC recommendations for typed or temporary tables.
   - The Design Advisor does not make MDC recommendations for federated tables.
   - Storage space (approximately 1% of the table data for large tables) must exist for the sampling data used during the execution of the Design Advisor, otherwise the sampled table will be examined only for base columns under the uncorrelated assumption. A warning message will be generated in this case.
   - Tables without statistics collected will be skipped for consideration.
   - The Design Advisor does not make recommendations for multicolumn dimensions.
   - Existing tables must have data in them for sampling to work in MDC selection.
4. **Restrictions on database partitioning recommendations**

   The Design Advisor can only recommend database partitioning on DB2 Enterprise Server Edition. If the database partitioning options are specified with the db2advis command, an error is returned. In the Design Advisor GUI, the database partitioning feature is not selectable in a single-partition database environment.
5. **Additional restrictions**

   Simulation catalog tables are created during the execution of the Design Advisor. These tables are dropped when the Design Advisor execution completes. Incomplete Design Advisor execution may 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 from the command line. 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 generate the DROP statements that are required to remove the tables.

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 userid as the schema name.

You should create a separate table space for storing these simulated catalog tables and set DROP TABLE RECOVERY to ″OFF″. This allows for easier cleanup and for faster Design Advisor execution. The separate table space for the simulation catalog tables must only be defined on the catalog node.

Part 5. Tuning database application performance

Chapter 18. Application considerations

Concurrency issues

Because many users access and change data in a relational database, the database manager must be able both to allow users to make these changes and to ensure 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 from the database and both calculate new values for one of its columns based on the data these applications read. If A updates the row with its new value and B then also updates the row, the update performed by A is lost.
- **Access to uncommitted data**. Application A might update a value in the database, and application B might read that value before it was committed. Then, if the value of A is not later committed, but backed out, the calculations performed by B are based on uncommitted (and presumably invalid) data.
- **Nonrepeatable reads**. Some applications involve the following sequence of events: application A reads a row from the database, then goes on to process other requests. In the meantime, application B either modifies or deletes the row and commits the change. Later, if application A attempts to read the original row again, it receives the modified row or discovers that the original row has been deleted.
- **Phantom Read Phenomenon**. The phantom read phenomenon occurs when:
  1. Your application executes a query that reads a set of rows based on some search criterion.
  2. Another application inserts new data or updates existing data that would satisfy your application's query.
  3. Your application repeats the query from step 1 (within the same unit of work).

     Some additional ("phantom") rows are returned as part of the result set that were not returned when the query was initially executed (step 1).

Note: Declared temporary tables have no concurrency issues because they are available only to the application that declared them. This type of table only exists from the time that the application declares it until the application completes or disconnects.

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) or databases in a single statement. To reference the data sources, which consist of a DBMS and data, DB2 uses *nicknames*. Nicknames are aliases for objects in other database managers. In a federated system, DB2 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, with location transparency if information about tables and views is moved, references to that information through nicknames can be updated without changing applications that request the information. When an application accesses

data through nicknames, DB2 relies on the concurrency control protocols of data-source database managers to ensure isolation levels. Although DB2 tries to match the requested level of isolation at the data source with a logical equivalent, results may vary depending on data source capabilities.

Isolation levels and performance

An *isolation level* determines how data is locked or isolated from other processes while the data is being accessed. The isolation level will be in effect for the duration of the unit of work. Applications that use a cursor declared with a DECLARE CURSOR statement using the WITH HOLD clause will keep the chosen isolation level for the duration of the unit of work in which the OPEN CURSOR was performed. DB2 supports the following isolation levels:

- Repeatable Read
- Read Stability
- Cursor Stability
- Uncommitted Read.

Note: Some host database servers support the *no commit* isolation level. On other databases, this isolation level behaves like the uncommitted read isolation level.

Detailed explanations for each of the isolation levels follows in decreasing order of performance impact, but in increasing order of care required when accessing and updating data.

Repeatable Read

Repeatable Read (RR) locks all the rows an application references within a unit of work. Using Repeatable Read, a SELECT statement issued by an application twice within the same unit of work in which the cursor was opened returns the same result each time. With Repeatable Read, lost updates, access to uncommitted data, and phantom rows are not possible.

The Repeatable Read application can retrieve and operate on the rows as many times as needed until the unit of work completes. However, no other applications can update, delete, or insert a row that would affect the result table, until the unit of work completes. Repeatable Read applications cannot see uncommitted changes of other applications.

With Repeatable Read, every row that is referenced is locked, not just the rows that are retrieved. Appropriate locking is performed so that 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 re-executed. This prevents phantom rows from occurring. For example, if you scan 10 000 rows and apply predicates to them, locks are held on all 10 000 rows, even though only 10 rows qualify.

Note: The Repeatable Read isolation level ensures that all returned data remains unchanged until the time the application sees the data, even when temporary tables or row blocking are used.

Since Repeatable Read may acquire and hold a considerable number of locks, these locks may exceed the number of locks available as a result of the *locklist* and *maxlocks* configuration parameters. In order to avoid lock escalation, the optimizer may elect to acquire a single table-level lock immediately for an index scan, if it believes that lock escalation is very likely to occur. This functions as though the

database manager has issued a LOCK TABLE statement on your behalf. If you do not want a table-level lock to be obtained ensure that enough locks are available to the transaction or use the Read Stability isolation level.

When evaluating referential constraints, there are a few situations where DB2 will internally upgrade the isolation level used on the scan on the child table to Repeatable Read (RR), regardless of the isolation level set by the user. This will result in additional locks being held until commit, which increases the likelihood of a deadlock or lock timeout occurring. To avoid this, it is recommended that you create an index that only contains the column or columns of the foreign key, allowing the RI scan to use this index instead.

Read Stability

Read Stability (RS) locks only those rows that an application retrieves within a unit of work. It ensures that any qualifying row read during a unit of work is not changed by other application processes until the unit of work completes, and that any row changed by another application process is not read until the change is committed by that process. That is, "nonrepeatable read" behavior is **not** possible.

Unlike repeatable read, with Read Stability, if your application issues the same query more than once, you may see additional *phantom* rows (the *phantom read phenomenon*). Recalling the example of scanning 10 000 rows, Read Stability only locks the rows that qualify. Thus, with Read Stability, only 10 rows are retrieved, and a lock is held only on those ten rows. Contrast this with Repeatable Read, where in this example, locks would be held on all 10 000 rows. The locks that are held can be share, next share, update, or exclusive locks.

Note: The Read Stability isolation level ensures that all returned data remains unchanged until the time the application *sees* the data, even when temporary tables or row blocking are used.

One of the objectives of the Read Stability isolation level is to provide both a high degree of concurrency as well as a stable view of the data. To assist in achieving this objective, the optimizer ensures that table level locks are not obtained until lock escalation occurs.

The Read Stability isolation level is best for applications that include all of the following:

- Operate in a concurrent environment
- Require qualifying rows to remain stable for the duration of the unit of work
- Do not issue the same query more than once within the unit of work, or do not require that the query get the same answer when issued more than once in the same unit of work.

Cursor Stability

Cursor Stability (CS) locks any row accessed by a transaction of an application while the cursor is positioned on the row. This lock remains in effect until the next row is fetched or the transaction is terminated. However, if any data on a row is changed, the lock must be held until the change is committed to the database.

No other applications can update or delete a row that a Cursor Stability application has retrieved while any updatable cursor is positioned on the row. Cursor Stability applications cannot see uncommitted changes of other applications.

Recalling the example of scanning 10 000 rows, if you use Cursor Stability, you will only have a lock on the row under your current cursor position. The lock is removed when the cursor moves off that row (unless you update that row).

With Cursor Stability, both nonrepeatable read and the phantom read phenomenon are possible. Cursor Stability is the default isolation level and should be used when you want the maximum concurrency while seeing only committed rows from other applications.

Uncommitted Read

Uncommitted Read (UR) allows an application to access uncommitted changes of other transactions. The application also does not lock other applications out of the row it is reading, unless the other application attempts to drop or alter the table. Uncommitted Read works differently for read-only and updatable cursors.

Read-only cursors can access most uncommitted changes of other transactions. However, 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.

Note: Cursors that are updatable operating under the Uncommitted Read isolation level will behave as if the isolation level was cursor stability.

When it runs a program using isolation level UR, an application can use isolation level CS. This happens because the cursors used in the application program are ambiguous. The ambiguous cursors can be escalated to isolation level CS because of a BLOCKING option. The default for the BLOCKING option is UNAMBIG. This means that ambiguous cursors are treated as updatable and the escalation of the isolation level to CS occurs. To prevent this escalation, you have the following two choices:

- Modify the cursors in the application program so that they are unambiguous. Change the SELECT statements to include the FOR READ ONLY clause.
- Leave cursors ambiguous in the application program, but precompile the program or bind it with the BLOCKING ALL and STATICREADONLY YES options to allow any ambiguous cursors to be treated as read-only when the program is run.

As in the example given for Repeatable Read, of scanning 10 000 rows, if you use Uncommitted Read, you do not acquire any row locks.

With Uncommitted Read, both nonrepeatable read behavior and the phantom read phenomenon are possible. The Uncommitted Read isolation level is most commonly used for queries on read-only tables, or if you are executing only select statements and you do not care whether you see uncommitted data from other applications.

Summary of isolation levels

The following table summarizes the different isolation levels in terms of their undesirable effects.

Table 4. Summary of isolation levels

Isolation Level	Access to uncommitted data	Nonrepeatable reads	Phantom read phenomenon
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 table below provides a simple heuristic to help you choose an initial isolation level for your applications. Consider this table a starting point, and refer to the previous discussions of the various levels for factors that might make another isolation level more appropriate.

Table 5. 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

Choosing the appropriate isolation level for an application is very important to avoid the phenomena that are intolerable for that application. The isolation level affects not only the degree of isolation among applications but also the performance characteristics of an individual application since the CPU and memory resources that are required to obtain and free locks vary with the isolation level. The potential for deadlock situations also varies with the isolation level.

Specifying the isolation level

Because the isolation level determines how data is locked and 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. The isolation level that you specify is in effect for the duration of the unit of work.

Note: Isolation levels cannot be specified for XQuery statements at the statement level.

The isolation level can be specified in several different ways. The following heuristics are used in determining which isolation level will be used in compiling an SQL or XQuery statement:

Static SQL:

- If an isolation clause is specified in the statement, then the value of that clause is used.
- If no isolation clause is specified in the statement, then the isolation level used is the one specified for the package at the time when the package was bound to the database.

Dynamic SQL:

- If an isolation clause is specified in the statement, then the value of that clause is used.
- If no isolation clause is specified in the statement, and a SET CURRENT ISOLATION statement has been issued within the current session, then the value of the CURRENT ISOLATION special register is used.
- If no isolation clause is specified in the statement, and no SET CURRENT ISOLATION statement has been issued within the current session, then the isolation level used is the one specified for the package at the time when the package was bound to the database.

Static or dynamic XQuery statements:

- The isolation level of the environment determines the isolation level 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.

To specify the isolation level:

- **At precompile or bind time:**

  For an application written in a supported compiled language, use the ISOLATION option of the command line processor PREP or BIND commands. You can also use the PREP or BIND APIs 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 used during precompilation.
  – If you do not specify an isolation level, the default of cursor stability is used.

  Note: To determine the isolation level of a package, execute the following query:

  ```
  SELECT ISOLATION FROM SYSCAT.PACKAGES
    WHERE PKGNAME = 'XXXXXXXX'
    AND PKGSCHEMA = 'YYYYYYYY'
  ```

  where XXXXXXXX is the name of the package and YYYYYYYY is the schema name of the package. Both of these names must be in all capital letters.

- **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 packages are also bound to the database when a database is created.

  REXX and the command line processor connect to a database using a default isolation level of cursor stability. Changing to a different isolation level does not change the connection state. It must be executed in the CONNECTABLE AND UNCONNECTED state or in the IMPLICITLY CONNECTABLE state.

  To verify the isolation level in use by a REXX application, check the value of the SQLISL REXX variable. The value is updated every time the CHANGE SQLISL command is executed.

- **At the statement level:**

  Use the WITH clause. The statement-level isolation level overrides the isolation level specified for the package in which the statement appears.

  You can specify an isolation level for the following SQL statements:

  – SELECT
  – SELECT INTO
  – Searched DELETE

- INSERT
- Searched UPDATE
- DECLARE CURSOR

The following conditions apply to isolation levels specified for statements:

- The WITH clause cannot be used on subqueries
- The WITH UR option applies only to read-only operations. In other cases, the statement is automatically changed from UR to CS.

- **From CLI or ODBC at runtime:**

  Use the CHANGE ISOLATION LEVEL command. For DB2 Call Level Interface (DB2 Call Level Interface), you can change the isolation level as part of the DB2 Call Level Interface configuration. At runtime, 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*. You can also use the TXNISOLATION keyword in the db2cli.ini file .

- **When working with JDBC or SQLJ at run time:**

  Note: JDBC and SQLJ are implemented with CLI on DB2, which means the db2cli.ini settings might affect what is written and run using JDBC and SQLJ.

  In SQLJ, you use the SQLJ profile customizer (db2sqljcustomize command) to create a package. The options that you can specify for this package include its isolation level.

- **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 level of isolation for any dynamic SQL statements issued within the current session. Once set, the CURRENT ISOLATION special register provides the isolation level for any subsequent dynamic SQL statement compiled within the session, regardless of the package issuing the statement. This isolation level will apply until the session is ended or until a SET CURRENT ISOLATION statement is issued with the RESET option.

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 used when the table is accessed. The choices are either ROW for row locks, TABLE for table locks, or BLOCKINSERT for block locks on MDC tables only. When the BLOCKINSERT clause is used on an MDC table, row-level locking is performed except on an INSERT operation where block-level locking is done instead. Use the ALTER TABLE ... LOCKSIZE BLOCKINSERT statement for MDC tables when transactions perform large inserts into disjoint cells. Use the ALTER TABLE ... LOCKSIZE TABLE statement for read-only tables. This reduces the number of locks required by database activity. For partitioned tables, table locks are first acquired and then data partition locks are acquired, as dictated by the data accessed.
- The amount of memory devoted to locking. The amount of memory devoted to locking is controlled by the `locklist` database configuration parameter. If the lock list fills, performance can degrade due to lock escalations and reduced concurrency on shared objects in the database. If lock escalations occur frequently, increase the value of either `locklist` or `maxlocks`, or both. Also, to reduce number of locks held at one time, ensure that transactions COMMIT frequently to free held locks.

Although most locking occurs on tables, when a buffer pool is created, altered, or dropped, a buffer pool lock is set. The mode used with this lock is EXCLUSIVE (X). You may encounter this type of lock when collecting system monitoring data. When viewing the snapshot, you will see that the lock name used is the identifier (ID) of the buffer pool itself.

In general, row-level locking is used unless one of the following is the case:

- The isolation level chosen is uncommitted read (UR).
- The isolation level chosen is repeatable read (RR) and the access plan requires a scan with no predicates.
- The table LOCKSIZE attribute is “TABLE”.
- The lock list fills, causing escalation.
- There is an explicit table lock acquired via the LOCK TABLE statement. The LOCK TABLE statement prevents concurrent application processes from either changing a table or using a table.

If this is an MDC table, there are several other cases where block-level locking is used instead of row-level locking, including:

- The table LOCKSIZE attribute is “BLOCKINSERT”
- The isolation level chosen is repeatable read (RR) and the access plan involves predicates
- A searched update or delete operation that involves only predicates on dimension columns

Lock escalations reduce concurrency. Conditions that might cause lock escalations should be avoided.

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 only held while the cursor is positioned on the row.
- RS scans: Only qualifying row locks are held for the duration of the transaction.

- RR scans: All row locks are held for the duration of the transaction.

Lock attributes

Database manager locks have the following basic attributes:

Mode The type of access allowed for the lock owner as well as the type of access permitted 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 imposes locks on other types of resources, such as rows, tables, and table spaces. For multidimensional clustering (MDC) tables, block locks can also be imposed; for partitioned tables, data partition locks can be imposed. The object being locked determines the *granularity* of the lock.

Duration
The length of time a lock is held. The isolation level in which the query runs affects the lock duration.

The following table shows the modes and their effects in order of increasing control over resources. For detailed information about locks at various levels, refer to the lock-mode reference tables.

Table 6. 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.
NS (Next Key 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. NS lock mode is not used for next-key locking. It is used instead of S mode during CS and RS scans to minimize the impact of next-key locking on these scans.
S (Share)	Rows, blocks, tables, data partitions	The lock owner and all concurrent applications can read, but not update, the locked data.
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.
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 attempt to update it.
NW (Next Key Weak Exclusive)	Rows	When a row is inserted into an index, an NW lock is acquired on the next row. For type 2 indexes, 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 W and NS locks.

Table 6. Lock Mode Summary (continued)

Lock Mode	Applicable Object Type	Description
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 applications can access the locked object.
W (Weak Exclusive)	Rows	This lock is acquired on the row when a row is inserted into a table that does not have type-2 indexes defined. The lock owner can change the locked row. To determine if a duplicate value has been committed when a duplicate value is found, this lock is also used during insertion into a unique index. This lock is similar to an X lock except that it is compatible with the NW lock. Only uncommitted read applications can access the locked row.
Z (Super Exclusive)	Table spaces, tables, data partitions	This lock is acquired on a table in 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.

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 scope (granularity) of locks at the row, data partition, block, or table level. By default, row locks are used. Only S (Shared) and X (Exclusive) locks are requested by these defined table locks. The ALTER TABLE statement LOCKSIZE ROW clause does not prevent normal lock escalation from occurring.

A permanent table lock defined by the ALTER TABLE statement might be preferable to a single-transaction table lock using LOCK TABLE statement in the following cases:

- 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 runs, the 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 an MDC table, you can specify BLOCKINSERT for the LOCKSIZE clause in order to use block-level locking during INSERT operations only. When this 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 for next-key locking if RR scans are encountered in the indexes as they are being updated. BLOCKINSERT locking might be beneficial in the following cases:

- There are multiple transactions doing mass insertions into separate cells.

- Concurrent insertions to 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.

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.

Lock waits and timeouts

Lock timeout detection is a database manager feature that prevents applications from waiting indefinitely for a lock to be released in an abnormal situation. For example, a transaction might be waiting for a lock held by another user's application, but the other user has left the workstation without allowing the application to commit the transaction that would release the lock. To avoid stalling an application in such a case, set the **locktimeout** configuration parameter to the maximum time that any application should wait to obtain a lock.

Setting this parameter helps avoid global deadlocks, especially in distributed unit of work (DUOW) applications. If the time that the lock request is pending is greater than the **locktimeout** value, the requesting application receives an error and its transaction is rolled back. For example, if APPL1 tries to acquire a lock which is already held by APPL2, APPL1 returns SQLCODE -911 (SQLSTATE 40001) with reason code 68 if the timeout period expires. The default value for **locktimeout** is -1, which turns off lock timeout detection.

Note: For table, row, data partition and MDC block locks, an application can override the database level **locktimeout** setting by using SET CURRENT LOCK TIMEOUT.

To generate a report file about the lock timeout, set the **DB2_CAPTURE_LOCKTIMEOUT** registry variable to ON. The lock timeout report includes information about the key applications involved in the lock contention that resulted in the lock timeout, as well as details about the lock itself, such as lock name, lock type, row ID, table space ID, and table ID.

To log more information about lock-request timeouts in the db2diag.log file, set the database manager configuration parameter **diaglevel** to four. The logged information includes the locked object, the lock mode, and the application 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** four.

You can get information about lock waits and lock timeouts from the lock wait information system monitor elements, or from the **db.apps_waiting_locks** health indicator.

Lock timeout reporting

The lock timeout reporting function captures information about lock timeout events, including information about the key applications involved in the lock contention that resulted in the lock timeout. This function is controlled by the **DB2_CAPTURE_LOCKTIMEOUT** registry variable.

Information is captured for both the lock requestor (the application that received the lock timeout error) and the current lock owner, and the report is stored in a directory determined by a database configuration parameter.

When a lock timeout occurs and the lock timeout reporting function is active, the following information is captured and placed in a lock timeout report file.

Lock in contention

- Lock name and type
- Lock specifics, including row ID, table space ID, and table ID. Use this information to query the SYSCAT.TABLES system catalog view to identify the name of the table.

Lock Requestor

- Application identification information, such as application name and coordinator partition
- Lock timeout value
- Lock mode requested
- Request SQL context, if applicable
- Package ID
- Section entry number
- SQL information, such as whether the statement is dynamic or static, type of statement
- Effective lock isolation
- Relevant statement text, if available (Refer to "Function limitations" on page 172 for details.)
- Application status
- Current operation
- Lock escalation

Lock owner or representative

There can be more than one lock owner. For example, when a lock is held in share mode by more than one application, only information about the first lock owner encountered is reported. The first lock owner is the representative for other lock owners.

- Application identification information, such as application name and coordinator partition
- Lock mode held
- List of currently active SQL statements at this partition
  1. Package ID
  2. Section entry number
  3. SQL information, such as whether the statement is dynamic or static, type of statement
  4. Effective lock isolation
  5. Relevant statement text, when available
- List of inactive SQL statements from the current unit of work at this database partition (available only if a deadlock event monitor with statement history is active)
  1. Package ID
  2. Section entry number

3. SQL information, such as whether the statement is dynamic or static, type of statement
4. Effective lock isolation
5. Relevant statement text, when available

To collect additional information, such as the operating system or other related environment information, use the customized db2cos script.

Function usage

The **DB2_CAPTURE_LOCKTIMEOUT** registry variable controls the lock timeout reporting function.

When this registry variable is set to ON, the function captures basic information about each lock timeout that happens within the DB2 instance, and a lock timeout report is created. If the registry variable is set to empty, the function is disabled.

To enable lock timeout reporting, issue the following command:

```
db2set DB2_CAPTURE_LOCKTIMEOUT=ON
```

To disable lock timeout reporting, issue the following command:

```
db2set DB2_CAPTURE_LOCKTIMEOUT=
```

In some cases, you may want to capture all statements that have executed in the lock owner's unit of work prior to the occurrence of the lock timeout. For example, this is useful if the lock timeout is being caused by a previously executed SQL statement. To have the past history of a unit of work included in the lock timeout information for the current lock owner, activate a deadlock event monitor using the statement history clause. For example, use one of the following statements:

```
create event monitor testit for deadlocks with details history write to file path global
create event monitor testit for deadlocks with details history write to table
```

The CREATE EVENT MONITOR statement has additional options, such as the ability to specify the name of the tablespace and table into which data will be written. For details, see the CREATE EVENT MONITOR statement description.

The event monitor with statement history capability affects all applications and increases the monitor heap usage by the DB2 database manager. Only use it in cases where the cause of the lock contention is not the current statement being executed by the applications involved or if the goal of the report is more than simple identification of the applications involved in the lock timeout.

For example, application APPL1 executes the following statements:

```
INSERT INTO T1 VALUES (1)
SELECT * FROM T5
```

Subsequently, APPL2 executes the following statement:

```
SELECT * FROM T1
```

In this example, the SELECT statement executed by APPL2 is waiting for the row lock acquired in the previous INSERT statement executed by APPL1. In this case, seeing only the SELECT statement currently executed by APPL1 would not help identify the specific cause of the lock contention. The statement history for the lock owner would reveal the previous INSERT statement and the cause of the lock contention.

Creating an event monitor with statement history capability would also help if the application holding the lock is not running an SQL statement at the time of the lock timeout. If a deadlock event monitor with history is active, the application's previous SQL statements in the unit of work will be written to the report.

Function limitations

The lock timeout report might not always capture the required details about lock contention and lock timeout events. In some cases, such as those listed below, not all information is available to the lock timeout reporting function:

- The SQL statement text may not be available in all cases, such as when static SQL statements are involved in the lock timeout.
- DB2 utilities and internal functions can acquire locks without running SQL. For example, online backup acquires table space and table locks while processing these objects during online backup. An application could time out waiting for a lock while a table is being backed up. If the backup command was executed from CLP, the application name would be reported as "db2bp" (CLP), and there would be no active SQL statement for the lock owner. A DB2 utility or an internal function could also timeout waiting for an application to free a lock, and there would be no active SQL statement for the lock requestor.
- Subagents acting on an application's behalf at remote partitions may have incomplete information about the application's SQL statements and other information. In particular, if the lock owner is a remote subagent, it will not have the complete statement history for the application available to it.
- Internal catalog table processing performed during regular DB2 operation does not use SQL. The function will simply identify the table involved but not the component of DB2 that is either causing or encountering the conflict.
- Some locks are not acquired by specific SQL statements. They can represent non-table objects (such as packages) or are used for other internal DB2 processing.

Lock timeout report files

When the lock timeout reporting function is active and a lock timeout occurs, the agent receiving the lock timeout error generates a report file. This file is placed in the diagnostic data directory path, at the database partition where the lock timeout was encountered.

The diagnostic data directory path is defined by the database manager configuration parameter **diagpath**. If this parameter is null, see the **diagpath** parameter description for the report file location.

In a partitioned database environment, an application can be executing work and acquiring locks at one or more database partitions at the same time. The lock timeout reporting function helps isolate the specific database partition at which each lock timeout problem occurs by generating a report at that database partition. The report includes relevant information for the unique context of the lock timeout problem at that partition.

The report is stored in a file using the following name format: db2locktimeout.*par*.*AGENTID*.*yyyy-mm-dd-hh-mm-ss*, where

- *par* is the database partition number. In non-partitioned database environments, *par* is set to `0`.
- *AGENTID* is the Agent ID.

- *yyyy-mm-dd-hh-mm-ss* is the timestamp consisting of the year, month, day, hour, minute and second.

An example of a lock timeout report file name is /home/juntang/sqllib/db2dump/db2locktimeout.000.4944050.2006-08-11-11-09-43.

Note: When the lock timeout report files are no longer needed, delete them from the directory. Since the report files are in the same location as other diagnostics logs, the DB2 system could shut down if the directory is allowed to get full. If you need to keep some lock timeout report files for a long period of time, move them to a different directory or folder.

Lock conversion

Changing the mode of a lock already held is called a *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 time, although it can request a lock many times on the same data object indirectly through a query.

Some lock modes apply only to tables, others only to rows or blocks. For rows or blocks, conversion usually occurs if an X is needed and an S or U (Update) lock is held.

IX (Intent Exclusive) and S (Shared) locks are special cases with regard to lock conversion, however. Neither S nor IX is considered to be more restrictive than the other, so if one of these is held and the other is required, the resulting conversion is to a SIX (Share with Intent Exclusive) lock. All other conversions result in the requested lock mode becoming the mode of the lock held 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 an 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 to an X.

Lock conversion usually takes place implicitly as a query is executed. Understanding the kinds of locks obtained for different queries and table and index combinations can assist you in designing and tuning your application.

The system monitor elements *lock_current_mode* and *lock_mode* can provide information about lock conversions occurring in your database.

Preventing lock-related performance issues

Consider the following guidelines when you tune locking for concurrency and data integrity:

- Create small units of work with frequent COMMIT statements to promote concurrent access of data by many users.

  Include COMMIT statements when your application is logically consistent, that is, when the data you have changed is consistent. When a COMMIT is issued, locks are released except for table locks associated with cursors declared WITH HOLD.
- Close a CURSOR WITH HOLD before issuing a COMMIT statement

In some situations, locks remain after the result set is closed and the transaction is committed. Closing a CURSOR WITH HOLD before issuing a COMMIT statement, ensures locks are released.

- Execute INSERT statements as separate units of work

  In some situations, locks remain after the result set is closed and the transaction is committed. Executing INSERT statements as separate units of work, ensures locks are released.

- Specify an appropriate isolation level.

  Locks are acquired even if your application merely reads rows, so it is still important to commit read-only units of work. This is because shared locks are acquired by repeatable read, read stability, and cursor stability isolation levels in read-only applications. With repeatable read and read stability, all locks are held until a COMMIT is issued, preventing other processes from updating the locked data, unless you close your cursor using the WITH RELEASE clause. In addition, catalog locks are acquired even in uncommitted read applications using dynamic SQL or XQuery statements.

  The database manager ensures that your application does not retrieve uncommitted data (rows that have been updated by other applications but are not yet committed) unless you are using the uncommitted read isolation level.

- Use the LOCK TABLE statement appropriately.

  The statement locks an entire table. Only the table specified in the LOCK TABLE statement is locked. Parent and dependent tables of the specified table are not locked. You must determine whether locking other tables that can be accessed is necessary to achieve the desired result in terms of concurrency and performance. The lock is not released until the unit of work is committed or rolled back.

  LOCK TABLE IN SHARE MODE

  You want to access data that is *consistent in time*; that is, data current for a table at a specific point in time. If the table experiences frequent activity, the only way to ensure that the entire table remains stable is to lock it. For example, your application wants to take a snapshot of a table. However, during the time your application needs to process some rows of a table, other applications are updating rows you have not yet processed. This is allowed with repeatable read, but this action is not what you want.

  As an alternative, your application can issue the LOCK TABLE IN SHARE MODE statement: no rows can be changed, regardless of whether you have retrieved them or not. You can then retrieve as many rows as you need, knowing that the rows you have retrieved have not been changed just before you retrieved them.

  With LOCK TABLE IN SHARE MODE, other users can retrieve data from the table, but they cannot update, delete, or insert rows into the table.

  LOCK TABLE IN EXCLUSIVE MODE

  You want to update a large part of the table. It is less expensive and more efficient to prevent all other users from accessing the table than it is to lock each row as it is updated, and then unlock the row later when all changes are committed.

  With LOCK TABLE IN EXCLUSIVE MODE, all other users are locked out; no other applications can access the table unless they are uncommitted read applications.

- Use ALTER TABLE statements in applications.

The ALTER TABLE statement with the LOCKSIZE parameter is an alternative to the LOCK TABLE statement. The LOCKSIZE parameter lets you specify a lock granularity of either ROW locks or TABLE locks for the next table access. For MDC tables, it also lets you specify a lock granularity of the BLOCKINSERT clause.

The selection of ROW locks is no different from selecting the default lock size when a table is created. The selection of TABLE locks may improve query performance by limiting the number of locks that need to be acquired. However, concurrency might be reduced because all locks are on the complete table. For MDC tables, the selection of the BLOCKINSERT clause may improve the performance of INSERT operations by locking at the block level and avoiding row locks for insertions. Row-level locking is still performed for all other operations and is performed on key insertions to protect Repeatable Read (RR) scanners. The BLOCKINSERT option is useful for large insertions into cells by individual transactions. None of the LOCKSIZE choices prevent normal lock escalation.

- Close cursors to release the locks that they hold.

  When you close a cursor with the CLOSE CURSOR statement that includes the WITH RELEASE clause, the database manager attempts to release all read locks that have been held for the cursor. Table read locks are IS, S, and U table locks. Row-read locks are S, NS, and U row locks. Block-read locks are IS, S, and U block locks.

  The WITH RELEASE clause has no effect on cursors that are operating under the CS or UR isolation levels. When specified for cursors that are operating under the RS or RR isolation levels, the WITH RELEASE clause ends some of the guarantees of those isolation levels. Specifically, a RS cursor may experience the *nonrepeatable read* phenomenon, and a RR cursor may experience either the *nonrepeatable read* or *phantom read* phenomenon.

  If a cursor that is originally RR or RS is reopened after being closed using the WITH RELEASE clause, then new read locks are acquired.

  In CLI applications, the DB2 CLI connection attribute SQL_ATTR_CLOSE_BEHAVIOR can be used to achieve the same results as CLOSE CURSOR WITH RELEASE.
- In a partitioned database environment, when you change the configuration parameters that affecting locking, ensure that the changes are made to all of the database partitions.

Correcting lock escalation problems

The database manager can automatically escalate locks from row or block level to table level. For partitioned tables, the database manager can automatically escalate locks from row or block level to data partition level. The *maxlocks* database configuration parameter specifies when lock escalation is triggered. The table that acquires the lock that triggers lock escalation might not be affected. Locks are first escalated for the table with the most locks, beginning with tables for which long object (LOBs) and long VARCHAR descriptors are locked, then the table with the next highest number of locks, and so on, until the number of locks held is decreased to about half of the value specified by *maxlocks*.

In a well designed database, lock escalation rarely occurs. If lock escalation reduces concurrency to an unacceptable level (indicated by the *lock_escalation* monitor element or the *db.lock_escal_rate* health indicator) you need to analyze the problem and decide how to solve it.

Ensure that lock escalation information is recorded. Set the database manager configuration parameter *notifylevel* to 3, which is the default, or to 4. At *notifylevel* of 2, only the error SQLCODE is reported. At *notifylevel* of 3 or 4, when lock escalation fails, information is recorded for the error SQLCODE and the table for which the escalation failed. The current query statement is logged only if it is a currently executing, dynamic query statement and *notifylevel*is set to 4.

Follow these general steps to diagnose the cause of unacceptable lock escalations and apply a remedy:

1. Analyze in the administration notification log on all tables for which locks are escalated. This log file includes the following information:
   - The number of locks currently held.
   - The number of locks needed before lock escalation is completed.
   - The table identifier information and table name of each table being escalated.
   - The number of non-table locks currently held.
   - The new table level lock to be acquired as part of the escalation. Usually, an "S," or Share lock, or an "X," or eXclusive lock is acquired.
   - The internal return code of the result of the acquisition of the new table lock level.
2. Use the information in administration notification log to decide how to resolve the escalation problem. Consider the following possibilities:
   - Increase the number of locks allowed globally by increasing the value of the *maxlocks* or the *locklist* parameters, or both, in the database configuration file. In a partitioned database, make this change on all database partitions.

     You might choose this method if concurrent access to the table by other processes is most important. However, the overhead of obtaining record level locks can induce more delay to other processes than is saved by concurrent access to a table.
   - Adjust the process or processes that caused the escalation. For these processes, you might issue LOCK TABLE statements explicitly.
   - Change the degree of isolation. Note that this may lead to decreased concurrency, however.
   - Increase the frequency of commits to reduce the number of locks held at a given time.
   - Consider frequent COMMIT statements for transactions that require long VARCHAR or various kinds of long object (LOB) data. Although this kind of data is not retrieved from disk until the result set is materialized, the descriptor is locked when the data is first referenced. As a result, many more locks might be held than for rows that contain more ordinary kinds of data.

Evaluate uncommitted data via lock deferral

To improve concurrency, DB2 now permits the deferral of row locks for CS or RS isolation scans in some situations until a record is known to satisfy the predicates of a query. By default, when row-locking is performed during a table or index scan, DB2 locks each row that is scanned before determining whether the row qualifies for the query. To improve the concurrency of scans, it may be possible to defer row locking until after it is determined that a row qualifies for a query.

To take advantage of this feature, enable the DB2_EVALUNCOMMITTED registry variable.

With this variable enabled, predicate evaluation can occur on uncommitted data. This means that a row that contains an uncommitted update may not satisfy the query, whereas if the predicate evaluation waited until the updated transaction completed, the row may satisfy the query. Additionally, uncommitted deleted rows are skipped during table scans. DB2 will skip deleted keys in type-2 index scans if the DB2_SKIPDELETED registry variable is enabled.

These registry variable settings apply 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 runtime, the lock avoidance strategy is not employed unless DB2_EVALUNCOMMITTED was enabled at bind time. If the registry variable is enabled at bind time but not enabled at runtime, the lock avoidance strategy is still in effect. For static SQL or XQuery statements, if a package is rebound, the registry variable setting 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.

Applicability of evaluate uncommitted for different access plans

Table 7. RID Index Only Access

Predicates	Evaluate Uncommitted
None	No
SARGable	Yes

Table 8. Data Only Access (relational or deferred RID list)

Predicates	Evaluate Uncommitted
None	No
SARGable	Yes

Table 9. 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 10. 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 of the default locking behavior and the new evaluate uncommitted behavior.

The table below 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 are acting on this table, with the default Cursor Stability (CS) isolation level.

Table 11. Transactions on the ORG table with 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 in Session 1 holds an exclusive record lock on the first row in the table, prohibiting the SELECT query in Session 2 from returning even though the row being updated in Session 1 does not currently satisfy the query in Session 2. This is because the CS isolation level dictates that any row 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.

When scanning the table, the lock-wait in Session 2 can be avoided using the evaluate uncommitted feature which first evaluates the predicate and then locks the row for a true predicate evaluation. 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 would also mean 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 evaluate uncommitted. Under default locking behavior, Session 2 would first acquire a row lock prohibiting the searched UPDATE in Session 1 from executing even though the UPDATE in Session 1 would not change the row locked by the query of Session 2. If the searched UPDATE in Session 1 first attempted to examine rows and then only lock them if they qualified, the query in Session 1 would be non-blocking.

Restrictions

The following external restrictions apply to this new functionality:

- The registry variable DB2_EVALUNCOMMITTED must be enabled.

- The isolation level must be CS or RS.
- Row locking is to occur.
- SARGable evaluation predicates exist.
- Evaluation uncommitted is not applicable to scans on the catalog tables.
- For MDC tables, block locking can be deferred for an index scan; however, block locking will not be deferred for table scans.
- Deferred locking will not occur on a table which is executing an inplace table reorg.
- Deferred locking will not occur for an index scan where the index is type-1.
- For Iscan-Fetch plans, row locking is not deferred to the data access but rather the row is locked during index access before moving to the row in the table.
- Deleted rows are unconditionally skipped for table scans while deleted type-2 index keys are only skipped if the registry variable DB2_SKIPDELETED is enabled.

Option to disregard uncommitted insertions

The DB2_SKIPINSERTED registry variable controls whether uncommitted insertions can be ignored for cursors using the Cursor Stability (CS) or Read Stability (RS) isolation levels. The DB2 database system can handle uncommitted insertions in the following ways:

- The DB2 database system can wait until the INSERT transaction completes (commits or rolls back) and process data accordingly. This is the default option, OFF.

  The following examples show instances when the default option, OFF, is preferred:

  – 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 in the table such that if the next row to be read is being inserted by the first application, the second application must wait until the insert is committed. In such cases, the default value for DB2_SKIPINSERTED should be used.

  – Suppose that an application modifies data by deleting the data and inserting the new image of the data. In such cases that avoid UPDATE statements, the default value for DB2_SKIPINSERTED should be used.

- The DB2 database system can ignore uncommitted insertions, which in many cases can improve concurrency. If you want this behavior, the registry variable must be specified as ON.

  In general, ON produces greater concurrency and is preferred for most applications. When the registry variable is enabled, uncommitted inserted rows are treated as if they had not yet been inserted.

Lock type compatibility

Lock compatibility becomes an issue when one application currently has 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 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.

The following table displays information about the circumstances in which a lock request can be granted when another process holds or is requesting a lock on the same resource in a given state. A **no** indicates that the requestor must wait until all incompatible locks are released by other processes. Note that a timeout can occur when a requestor is waiting for a lock. A **yes** indicates that the lock is granted unless an earlier requestor is waiting for the resource.

Table 12. Lock Type Compatibility

	State of Held Resource											
State Being Requested	**none**	**IN**	**IS**	**NS**	**S**	**IX**	**SIX**	**U**	**X**	**Z**	**NW**	**W**
none	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
IN	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	yes
IS	yes	yes	yes	yes	yes	yes	yes	yes	no	no	no	no
NS	yes	yes	yes	yes	yes	no	no	yes	no	no	yes	no
S	yes	yes	yes	yes	yes	no	no	yes	no	no	no	no
IX	yes	yes	yes	no	no	yes	no	no	no	no	no	no
SIX	yes	yes	yes	no	no	no	no	no	no	no	no	no
U	yes	yes	yes	yes	yes	no	no	no	no	no	no	no
X	yes	yes	no	no	no	no	no	no	no	no	no	no
Z	yes	no	no	no	no	no	no	no	no	no	no	no
NW	yes	yes	no	yes	no	no	no	no	no	no	no	yes
W	yes	yes	no	no	no	no	no	no	no	no	yes	no

Note:

I	Intent
N	None
NS	Next Key Share
S	Share
X	Exclusive
U	Update
Z	Super Exclusive
NW	Next Key Weak Exclusive
W	Weak Exclusive

Note:

- yes - **grant** lock requested immediately
- no - **wait** for held lock to be released or timeout to occur

Lock modes and access paths for standard tables

This topic includes reference information about locking methods for standard tables for different data-access plans.

The following tables list the types of locks obtained for standard tables at each level for different access plans. Each entry is made up of two parts: table lock and row lock. A dash indicates that a particular level of locking is not done.

Note:

1. In a multi-dimensional clustering (MDC) environment, an additional lock level, BLOCK, is used.
2. Lock modes can be changed explicitly with the lock-request-clause of a select statement.

Table 13. 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 14. 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: At UR isolation level with IN lock for type-1 indexes or if there are predicates on include columns in the index, the isolation level is upgraded to CS and the locks to an IS table lock and NS row locks.

Table 15. 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 16. 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

Table 16. Lock Modes for RID Index Scans with a Single Qualifying Row (continued)

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan	Update or Delete
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 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 18. 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

The following tables shows the lock modes for cases in which reading of the data pages is deferred to allow the list of rows to be:

- Further qualified using multiple indexes
- Sorted for efficient prefetching

Table 19. 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 20. 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 21. 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 22. 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 23. 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

Table 24. Lock modes for index scans used for deferred data page access, after a RID index scan with 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	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

Lock modes for table and RID index scans of MDC tables

In a multi-dimensional clustering (MDC) environment, an additional lock level, BLOCK, is used. The following tables list the types of locks obtained at each level for different access plans. Each entry is made up of three parts: table lock, block lock, and row lock. A dash indicates that a particular level of locking is not used.

Note: Lock modes can be changed explicitly with the lock-request-clause of a select statement.

Table 25. 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 or delete	Update
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 26. 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 or delete	Update
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 27. 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 or delete	Update
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

Table 27. Lock Modes for Table Scans with Other Predicates (sargs, resids) (continued)

Isolation Level	Read-only and ambiguous scans	Cursored operation		Searched update or delete	
		Scan	Where current of	Scan or delete	Update
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X

The following two tables show lock modes for RID indexes on MDC tables.

Table 28. 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 Delete	Update
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 29. Lock Modes for RID Index Scans with Single Qualifying Row

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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 30. 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 Delete	Update
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
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 31. Lock Modes for RID Index Scans with Index Predicates Only

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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

Table 31. Lock Modes for RID Index Scans with Index Predicates Only (continued)

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X

Table 32. Lock Modes for RID Index Scans with Other Predicates (sargs, resids)

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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

Note: In the following tables, which shows lock modes for RID index scans used for deferred data-page access, at UR isolation level with IN lock for type-1 indexes or 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, and NS row locks.

Table 33. Lock modes for RID 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 Delete	Update
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 34. Lock modes for RID index scans used for deferred data-page access: 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 Delete	Update
RR	IN/IN/-	IX/IX/S	IX/IX/X	X/-/-	X/-/-
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 35. Lock modes for RID 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 Delete	Update
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 36. Lock modes for RID index scans used for deferred data-page access: 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 Delete	Update
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 37. Lock modes for RID 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 Delete	Update
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 38. Lock modes for RID index scans used for deferred data-page access: 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 Delete	Update
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

Locking for block index scans for MDC tables

The following tables list the types of locks obtained at each level for different access plans. Each entry is made up of three parts: table lock, block lock, and row lock. A dash indicates that a particular level of locking is not done.

Note: Lock modes can be changed explicitly with the lock-request-clause of a select statement.

Table 39. Lock Modes for Index Scans with No Predicates

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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 40. Lock Modes for Index Scans with Dimension Predicates Only

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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 41. Lock Modes for 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 Delete	Update
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 42. Lock Modes for Index Scans with Predicates

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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

Table 42. Lock Modes for Index Scans with Predicates (continued)

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
UR	IN/IN/-	IX/IX/U	IX/IX/X	IX/IX/U	IX/IX/X

The following table lists lock modes for block index scans used for deferred data-page access:

Table 43. Lock modes for block index scans used for deferred data-page access: Block index scan with no predicates

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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 44. Lock modes for block index scans used for deferred data-page access: Deferred data-page access after a block index scan with no predicates

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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 45. Lock modes for block index scans used for deferred data-page access: Block index scan with dimension predicates only

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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 46. Lock modes for block index scans used for deferred data-page access: Deferred data-page access after a block index scan with dimension predicates only

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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 47. Lock modes for block 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 operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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 48. Lock modes for block index scans used for deferred data-page access: Deferred data-page access after a block 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 Delete	Update
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 49. Lock modes for block index scans used for deferred data-page access: Block index scan other predicates (sargs, resids)

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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 50. Lock modes for block index scans used for deferred data-page access: Deferred data-page access after a block index scan with other predicates (sargs, resids)

Isolation Level	Read-only and ambiguous scans	Cursored operations		Searched update or delete	
		Scan	Where current of	Scan Delete	Update
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 non-partitioned table. The new data partition lock is identified in the output of the db2pd command, event monitors, administrative views, and table functions.

When accessing a table, locking behavior obtains the table lock first, and then acquires data partition locks as dictated by the data accessed. Access methods and isolation levels might require locking of data partitions not in the result set. Once 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 costs of re-acquiring the data partition lock if that data partition is referenced in subsequent keys. The data partition lock also carries the cost of ensuring access to the table spaces. For non-partitioned tables, table space access is handled by the table lock. Therefore, data partition locking occurs even if there is an exclusive or share lock at the table level for a partitioned table.

Finer granularity allows one transaction to have exclusive access to a given data partition and avoid row locking while other transactions are able to access other data partitions. This can be a result of the plan chosen for a mass update or due to 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. 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 may be accessed, then a non-intent might be chosen at the table level to prevent deadlocks between data partition locks from concurrent transactions.

Locking for SQL LOCK TABLE statements

For partitioned tables, the only lock acquired for the LOCK TABLE statement is at the table level; no data partition locks are acquired. This ensures no row locking to the table for subsequent DML statements as well as avoiding deadlocks at the row, block, or data partition level. Using LOCK TABLE IN EXCLUSIVE MODE can also be used to guarantee exclusive access when updating indexes, which is useful in limiting the growth of type 2 indexes during a large update.

Effect of the LOCKSIZE TABLE parameter of the ALTER TABLE statement

The ALTER TABLE statement has an option for setting LOCKSIZE TABLE, which ensures that the table is locked share or exclusive with no intent locks. For a partitioned table this lock strategy is applied to both the table lock and to the data partition locks for any data partitions accessed.

Row and block lock escalation

For partitioned tables, the unit of escalation of row and block locks is to the data partition level. This again means that a table is more accessible to other transactions even if a data partition is escalated to share, exclusive, or super exclusive, leaving other non-escalated data partitions are unaffected. The transaction might continue to row lock on other data partitions after escalation for a given data partition. The notification log message for escalations includes the data partition escalated as well as the partitioned table's name. Therefore, exclusive access to an index cannot be ensured by lock escalation. Either the statement must use an exclusive table level lock, an explicit LOCK TABLE IN EXCLUSIVE MODE statement must be issued, or the table must use the LOCKSIZE TABLE attribute. The overall table lock for the access method is chosen by the optimizer, and depends upon data partition elimination. A large update to a table might choose an exclusive table lock if there is no data partition elimination occurring.

Interpreting lock information

The following example from the SNAPLOCK administrative view can help you interpret lock information returned for a partitioned table.

Example 1:

This 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 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  LOCK_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 and concurrency to each data partition.

There are additional lock objects of type TABLE_LOCK captured in this output (TAB_FILE_ID 4 through 16) which 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.

Factors that affect locking

The following factors affect the mode and granularity of database manager locks:

- The type of processing that the application performs
- The data access method
- Whether indexes are type-2 or type-1
- 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

  This type includes all select statements that are intrinsically read-only, have an explicit FOR READ ONLY clause, or are ambiguous but which the query compiler assumes to be read-only because of the value of the BLOCKING option that the PREP or BIND command specifies. This processing type requires only Share locks (S, NS, or IS).

- Intent to change

  This type includes all select statements with the FOR UPDATE clause, with the USE AND KEEP UPDATE LOCKS clause, with the USE AND KEEP EXCLUSIVE LOCKS clause, or for which the query compiler interprets an ambiguous statement to imply that change is intended. This type uses Share and Update locks (S, U, and X for rows; IX, U, X, and S for blocks; IX, U, and X for tables).

- Change

  This type includes UPDATE, INSERT, and DELETE, but not UPDATE WHERE CURRENT OF or DELETE WHERE CURRENT OF. This type requires Exclusive locks (X or IX).

- Cursor controlled

  This type includes UPDATE WHERE CURRENT OF and DELETE WHERE CURRENT OF. It also requires Exclusive locks (X or IX).

A statement that inserts, updates or deletes data in a target table, based on the result from a sub-select statement, does two types of processing. The rules for read-only processing determine the locks for the tables returned in the sub-select 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. For example, when an index scan is used to locate a specific row, the optimizer will probably choose row-level locking (IS) for the table. For example, if the EMPLOYEE table that has an index on employee number (EMPNO), access through an index might be used to select information for a single employee with a statement that contains the following SELECT clause:

```
SELECT *
  FROM EMPLOYEE
  WHERE EMPNO = '000310';
```

If an index is not used, the entire table must be scanned in sequence to find the selected rows, and may thus acquire 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 with a a statement that contains the following SELECT clause:

```
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 a cursor, an exclusive lock is always obtained to perform the update or delete.

Locking in range-clustered tables works slightly differently from standard key or next-key locking. In 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 or next key locking, only rows with existing records are locked.

Reference tables provide detailed information about which locks are obtained for what kind of access plan.

Deferred access of the data pages implies that access to the row occurs in two steps, which results in more complex locking scenarios. The timing of lock acquisition and the persistence of the locks depend on the isolation level. Because the Repeatable Read isolation level retains all locks until the end of the transaction, the locks acquired in the first step are held and there is no need to acquire further locks in the second step. For the Read Stability and Cursor Stability isolation levels, locks must be acquired during the second step. To maximize concurrency, locks are not acquired during the first step and rely on the reapplication of all predicates to ensure that only qualifying rows are returned.

Index types and next-key locking

As transactions cause changes to type-1 indexes, some next-key locking occurs. For type-2 indexes, minimal next-key locking occurs.

- Next-key locking for type 2 indexes

  Next-key locking occurs when a key is inserted into an index.

  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 an RR index scan. The lock mode used for the next-key lock is NW. This next-key lock is released before the key insertion is actually performed. Key insertion occurs when a row is inserted into a table.

When updates to a row result in a change to the value of the index key for that row, key insertion also occurs 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. If the key that follows the end of the scan range is marked deleted, the scan continues to lock the corresponding rows until it finds a key that is not marked deleted, when it locks the corresponding row for that key, or until the end of the index is locked.

- Next-key locking for type-1 indexes:

  Next-key locks occur during deletes and inserts to indexes and during index scans. When a row is updated in, deleted from, or inserted into a table, an X lock is obtained on that row. For insertions this might be downgraded to a W lock.

  When the key is deleted from the table index or inserted into it, the table row that corresponds to the key that follows the deleted or inserted key in the index is locked. For updates that affect the value of the key, the original key value is first deleted and the new value is inserted, so two next-key locks are acquired. The duration of these locks is determined as follows:

  – During index key deletion, the lock mode on the next key is X and the lock is held until commit time.
  – During index key insertion, the lock mode on the next key is NW. This lock is acquired only if there is contention for the lock, in which case the lock is released before the key is actually inserted into the index.
  – During RR scans, the table row that corresponds to the key just beyond the end of the index scan range is locked in S mode and is held until commit time.
  – During CS/RS scans, the row corresponding to the key just beyond the end of the index scan range is locked in NS mode if there is contention for the lock. This lock is released once the end of the scan range is verified.

  The next-key locking for type-1 indexes during key insertions and key deletion might result in deadlocks. The following example shows how two transactions could deadlock. With type 2 indexes, such deadlocks do not occur.

  Consider the following example of an index that contains 6 rows with the following values: `1 5 6 7 8 12`.

  1. Transaction 1 deletes the row with key value 8. The row with value 8 is locked in X mode. When the corresponding key from the index is deleted, the row with value 12 is locked in X mode.
  2. Transaction 2 deletes the row with key value 5. The row with value 5 is locked in X mode. When the corresponding key from the index is deleted, the row with value 6 is locked in X mode.
  3. Transaction 1 inserts a row with key value 4. This row is locked in W mode. When inserting the new key into the index is attempted, the row with value 6 is locked in NW mode. This lock attempt will wait on the X lock that transaction 2 has on this row.
  4. Transaction 2 inserts a row with key value 9. This row is locked in W mode. When inserting the new key into the index is attempted, the row with key value 12 is locked in NW mode. This lock attempt will wait on the X lock that transaction 1 has on this row.

When type-1 indexes are used, this scenario will result in a deadlock and one of these transactions will be rolled back.

Specifying a lock wait mode strategy

An individual session can now 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 new SET CURRENT LOCK TIMEOUT statement, which changes the value of the CURRENT LOCK TIMEOUT special register. The CURRENT LOCK TIMEOUT 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.

Whereas the value of the *locktimeout* parameter applies to all locks, the lock types that are impacted by this new function include row, table, index key, and multidimensional clustering (MDC) block locks.

Tuning applications

Guidelines for restricting select statements

The optimizer assumes that an application must retrieve all of the rows specified by SELECT statement. This assumption is most appropriate in OLTP and batch environments. However, in "browse" applications, queries often define a large potential answer set but they retrieve only first few rows, usually only as many rows as are required to fill the screen.

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/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 in 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.

Note: Row blocking is affected if you use the FOR UPDATE, FETCH FIRST *n* ROWS ONLY, or the OPTIMIZE FOR *n* ROWS clause or if you declare your cursor as SCROLLing.

The following sections describe the performance advantages of each method.

FOR UPDATE Clause

The FOR UPDATE clause limits the result set by including only the 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 FOR UPDATE clause in the following circumstances:

- If the cursor associated with the SELECT statement cannot be deleted.
- If at least one of the selected columns is a column that cannot be updated in a catalog table and has not been excluded in the FOR UPDATE clause.

Use the DB2 CLI connection attribute SQL_ATTR_ACCESS_MODE in CLI applications for the same purposes.

FOR READ or FETCH ONLY Clause

The FOR READ ONLY clause or FOR FETCH ONLY clause ensures that read-only results are returned. Because the result table from a SELECT on a view defined as read-only is also read only, this clause is permitted but has no effect.

For result tables where updates and deletes are allowed, specifying FOR READ ONLY may improve the performance of FETCH operations if the database manager can retrieve blocks of data instead of exclusive locks. Do not use the FOR READ ONLY clause for queries that are used in positioned UPDATE or DELETE statements.

The DB2 CLI connection attribute SQL_ATTR_ACCESS_MODE can be used in CLI applications for the same purposes.

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 prefer 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 bounded by the value of “n” in the OPTIMIZE FOR clause. Thus the OPTIMIZE FOR clause affects both how the server retrieves the qualifying rows from the database by the server, and how it returns the qualifying rows to the client.

For example, suppose you are querying the employee table for the employees with the highest salary on a regular basis.

```
SELECT LASTNAME,FIRSTNAME,EMPNO,SALARY
FROM EMPLOYEE
ORDER BY SALARY DESC
```

You have defined a descending index on the SALARY column. However, since employees are ordered by employee number, the salary index is likely to be very

poorly clustered. To avoid many random synchronous I/Os, the optimizer would probably choose to use 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,FIRSTNAME,EMPNO,SALARY
FROM EMPLOYEE
ORDER BY SALARY DESC
OPTIMIZE FOR 20 ROWS
```

In this case, the optimizer probably chooses to use the SALARY index directly because only the twenty 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 interrupt the flow of rows, such as sorts. You are most likely to influence an access path by using OPTIMIZE FOR 1 ROW. 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 to retrieve a few rows.
- An index that matches the ORDER BY clause is more likely because no sort is required for the ORDER BY.
- List prefetch is less likely because this access method requires a sort.
- Sequential prefetch is less likely because of the understanding that only a small number of rows is required.
- In a join query, the table with the columns in the ORDER BY clause is likely to be picked as the outer table if an index on the outer table provides the ordering 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 below 3 use Greedy join enumeration method. This method sometimes results in access plans for multi-table joins that do not lend themselves to quick retrieval of the first few rows.

The OPTIMIZE FOR clause does not prevent you from retrieving all the qualifying rows. If you do retrieve all qualifying rows, the total elapsed time might be significantly greater than if the optimizer had optimized for the entire answer set.

If a packaged application uses the call level interface (DB2 CLI or ODBC), you can use the OPTIMIZEFORNROWS keyword in the `db2cli.ini` configuration file to have DB2 CLI automatically append an OPTIMIZE FOR clause to the end of each query statement.

When data is selected from nicknames, results may vary depending on data source support. If the data source referenced by the nickname supports the OPTIMIZE FOR clause and the DB2 optimizer pushes down the entire query to the data source, then the clause is generated in the remote SQL sent to the data source. If the data source does not support this clause or if the optimizer decides that the least-cost 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 available to the

optimizer for generating plans are slightly limited and performance gains from the OPTIMIZE FOR clause may be negligible.

If both the FETCH FIRST clause and the OPTIMIZE FOR clause are specified, the lower of the two 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 you specify both the FETCH FIRST clause and the OPTIMIZE FOR clause, the lower of the two values affects the communications buffer size. For optimization purposes the two values are independent of each other.

DECLARE CURSOR WITH HOLD Statement

When you declare a cursor with the DECLARE CURSOR statement that includes the WITH HOLD clause, open cursors remain open when the transaction is committed and all locks are released, except locks that protect the current cursor position of open WITH HOLD cursors.

If the transaction is rolled back, all open cursors are closed and all locks are released and LOB locators are freed.

The DB2 CLI connection attribute SQL_ATTR_CURSOR_HOLD can be used in CLI applications to achieve the same results. If a packaged application that uses the call level interface (DB2 CLI or ODBC), use the CURSORHOLD keyword in the `db2cli.ini` configuration file to have DB2 CLI automatically assume the WITH HOLD clause for every declared cursor.

Specifying row blocking to reduce overhead

Row blocking is supported for all statements and data types, including LOB data types. Row blocking reduces database manager overhead for cursors by retrieving a *block* of rows in a single operation.

Note: The block of rows that you specify is a number of pages in memory. It is not a multi-dimensional (MDC) table block, which is physically mapped to an extent on disk.

Row blocking is specified by the following arguments to the BIND or PREP commands:

UNAMBIG
For cursors that are specified with the FOR READ ONLY clause, blocking occurs.

Cursors that are not declared with the FOR READ ONLY or FOR UPDATE clause which are not *ambiguous* and are *read-only* will be blocked. *Ambiguous* cursors will not be blocked

ALL For cursors that are specified with the FOR READ ONLY clause or are not specified as FOR UPDATE, blocking occurs.

NO Blocking does not occur for any cursors.

For the definition of a read-only cursor and an ambiguous cursor, refer to DECLARE CURSOR statement.

Note: If you use the FETCH FIRST *n* ROWS ONLY clause or the OPTIMIZE FOR *n* ROWS clause in a SELECT statement, the number of rows per block will be the minimum of the following:

- The value calculated in the above formula
- The value of *n* in the FETCH FIRST clause
- The value of *n* in the OPTIMIZE FOR clause

Two database manager configuration parameters must be set appropriately. Both values are set as a number of pages of memory. Note the values of these parameters for use in block-size calculations.

- The database manager configuration parameter *aslheapsz* specifies application support layer heap size for local applications.
- The database manager configuration parameter *rqrioblk* specifies the size of the communication buffer between remote applications and their database agents on the database server.

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 the 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 database configuration parameter *rqrioblk*.

To increase the amount of memory allocated to the heap, modify the *database_memory* database configuration parameter by:

- Setting the parameter to AUTOMATIC will tell the database manager to manage database memory automatically.
- Increasing the value by 256 pages if the parameter is currently set to a user-defined numeric value.

To increase the performance of existing embedded SQL applications that reference LOB values, the application can be rebound using the BIND command and specifying either the BLOCKING ALL clause or the BLOCKING UNAMBIGUOUS clause to request blocking. Embedded applications will retrieve the LOB values from the server a row at a time once a block of rows have been retrieved from the server. UDFs returning large LOB results may cause DB2 to revert to single-row retrieval of LOB data where large amounts of memory are consumed on the server.

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 4 096.
   - Use the following formula for remote applications:

     ```
     Rows per block = rqrioblk / orl
     ```

2. To enable row blocking, specify an appropriate argument to the BLOCKING option in the PREP or BIND commands.

   If you do not specify a BLOCKING option, the default row blocking type is UNAMBIG. For the command line processor and call level interface, the default row blocking type is ALL.

Query tuning guidelines

Follow the query-tuning guidelines to fine-tune the SQL and XQuery statements in an application program. The guidelines are intended to help you minimize the use of system resources and the time required to return results from large tables and complex queries.

Note: The optimization class that the optimizer uses might eliminate the need for some fine tuning because the query compiler can rewrite the SQL and XQuery code into more efficient forms.

Note that the optimizer choice of an access plan is also affected by other factors, including environmental considerations and system catalog statistics. If you conduct benchmark testing of the performance of your applications, you can find out what adjustments might improve the access plan.

Query optimization using the REOPT bind option

To enable query optimization (or reoptimization) of static and dynamic SQL and XQuery statements that have host variables, special registers, global variables or parameter markers, bind the package with the REOPT bind option. When this option is used, the access path for an SQL or XQuery statement that both belongs to the package and contains host variables, parameter markers, global variables or special registers, will be optimized using the values of these variables rather than the default estimates that are chosen by the compiler. The optimization takes place at query execution time when the values are available.

Improving performance by binding with REOPT

SQL or XQuery queries may perform poorly during execution if the values used for the input variables such as parameter markers, host variables, global variables and special registers, are outside the predictive range of default filter factor estimates. Default filter factors, used for scenarios where the actual data value is not known, are estimates of how many rows will actually be returned at runtime when the actual data value is used.

The REOPT bind option specifies whether or not to have DB2 optimize an access path using values for host variables, parameter markers, global variables and special registers. REOPT values are specified by the following arguments to the BIND, PREP and REBIND commands:

REOPT NONE
The access path for a given SQL or XQuery statement containing host variables, parameter markers, global variables or special registers will not be optimized using real values for these variables; The default estimates for the these variables are used instead. This plan is cached and will be used subsequently. This is the default behavior.

REOPT ONCE
The access path for a given SQL or XQuery statement will be optimized

using the real values of the host variables, parameter markers, global variables or special registers when the query is first executed. This plan is cached and used subsequently.

REOPT ALWAYS
The access path for a given SQL or XQuery statement will always be compiled and reoptimized using the values of the host variables, parameter markers, global variables or special registers known at each execution time.

Data sampling in SQL and XQuery queries

Databases are growing so large and queries on those databases so complex that it is often impractical and sometimes unnecessary to access all of the data relevant to a query. In some cases, a user is interested in finding overall trends or patterns, in which case approximate answers within some margin of error will suffice. One way to speed up such queries is to perform the query on a random sample of the database. DB2 allows you to do efficient sampling of data in SQL and XQuery queries, potentially improving performance of large queries by orders of magnitude while maintaining a high degree of accuracy.

The most common application of sampling is for aggregate queries such as AVG, SUM, and COUNT, where reasonably accurate answers of the aggregates can be obtained from a sample of the data. Sampling can also be used to obtain a random subset of the actual rows in a table for auditing purposes or to speed up data mining and analysis tasks.

DB2 provides two methods of sampling: row-level sampling and block-level sampling.

Row-level Bernoulli sampling

Row-level Bernoulli sampling gets 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 gets a valid, random sample regardless 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 applied to it. If there is no index then there are no I/O savings over executing a query without sampling. If an index is available, then performance using this type of sampling is improved because the sampling predicate is applied on the RIDS inside 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 are sampled and not rows. A page is included in the sample with a probability of P/100. If a page is included, all of the rows in that page are included.

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 disparity in accuracy is most pronounced

when there are many rows per block or when the columns referenced in the query exhibit a high degree of clustering within the pages.

The best sampling method for a particular task will be determined by a user's time constraints and the desired degree of accuracy.

Specifying the sampling method

To execute a query on a random sample of data from a table, you can use the TABLESAMPLE clause of the table-reference clause in a SQL statement. To specify the method of sampling, use the keywords BERNOULLI or SYSTEM.

The BERNOULLI keyword specifies that row-level Bernoulli sampling is performed.

The SYSTEM keyword specifies that system page-level sampling is performed unless the optimizer determines that it is more efficient to perform row-level Bernoulli sampling instead.

Parallel processing for applications

DB2 supports parallel environments primarily on symmetric multi-processor (SMP) machines, but also to a limited extent on uniprocessor machines. In SMP machines, more than one processor can access the database, allowing parallel execution of complex SQL requests to be divided among the processors.

To specify the degree of parallelism to implement when you compile an application, use the CURRENT DEGREE special register, or the DEGREE bind option. *Degree* refers to the number of parts of a query that execute concurrently. There is no strict relation between the number of processors and the value that you select for the degree of parallelism. You can specify more or less than the number of processors on the machine. Even for uniprocessor machines you can set a degree higher than one to improve performance in some ways. Note, however, that each degree of parallelism adds to the system memory and CPU overhead.

Some configuration parameters must be modified to optimize performance when you use parallel execution of queries. In particular, for 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 three configuration parameters control and manage intra-partition parallelism.

- The *intra_parallel* database manager configuration parameter enables or disables parallelism support.
- The *max_querydegree* database configuration parameter sets an upper limit for 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.

If a query is compiled with DEGREE = ANY, the database manager chooses the degree of intra-partition parallelism based on a number of factors including the number of processors and the characteristics of the query. The actual degree used at runtime may be lower than the number of processors depending on these factors and the amount of activity on the system. Parallelism may be lowered before query execution if the system is heavily utilized. This occurs because

intra-partition parallelism aggressively uses system resources to reduce the elapsed time of the query, which may adversely affect the performance of other database users.

To display information about the degree of parallelism chosen by the optimizer, use the SQL Explain Facility to display the access plan. Use the database system monitor to display information about the degree of parallelism actually used at runtime.

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 may benefit from declaring a degree of 2 or more. In this case, the processor might not have to wait for input or output tasks to complete before starting to process a new query. Declaring a degree of 2 or more does not directly control I/O parallelism on a uniprocessor machine, however. Utilities such as Load can control I/O parallelism independently of such a declaration. The keyword `ANY` can also be used to set the *dft_degree* database manager configuration parameter. The `ANY` keyword allows the optimizer to determine the degree of intra-partition parallelism.

Chapter 19. Environmental considerations

Table space impact on query optimization

Certain characteristics of your table spaces can affect the access plan chosen by the query compiler:

- Container characteristics

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

  – OVERHEAD, which provides an estimate in milliseconds of the time 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 may use the following formula to help you estimate the overhead cost:

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

    where:

    - 0.5 represents an average overhead of one half rotation
    - Rotational latency is calculated in milliseconds for each full rotation, as follows:

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

      where you:

      • Divide by rotations per minute to get minutes per rotation
      • Multiply by 60 seconds per minute
      • Multiply by 1000 milliseconds per second.

    As an example, let the rotations per minute for the disk be 7 200. Using the rotational-latency formula, this would produce:

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

    which can then be used in the calculation of the OVERHEAD estimate with an assumed average seek time of 11 milliseconds:

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

    giving an estimated OVERHEAD value of about 15 milliseconds.

  – TRANSFERRATE, which provides an estimate in milliseconds of the time required to read one page of data into memory.

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

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

    where:

- spec_rate represents the disk specification for the transfer rate, in MB per second
- Divide by spec_rate to get seconds per MB
- Multiply by 1000 milliseconds per second
- Divide by 1 024 000 bytes per MB
- Multiply by the page size in bytes (for example, 4 096 bytes for a 4 KB page)

As an example, suppose the specification rate for the disk is 3 MB per second. This would produce the following calculation

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

giving an estimated TRANSFERRATE value of about 1.3 milliseconds per page.

If the table space containers are not single physical disks but are arrays of disks (such as RAID), then you must take additional considerations into account when you attempt to determine the TRANSFERRATE to use. If the array is relatively small then you can multiply the spec_rate by the number of disks, assuming that the bottleneck is at the disk level.

However, if the number of disks in the array making up the container is large, then the bottleneck may 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 capability is the product of the spec_rate and the number of disks. Instead, you must measure the actual I/O rate in MBs during a sequential scan. For example, a sequential scan could be `select count(*) from big_table` and will be MBs in size. Divide the result by the number of containers that make up the table space in which big_table resides. Use the result as a substitute for spec_rate in the formula given above. For example, a measured sequential I/O rate of 100 MBs while scanning a table in a four container table space would imply 25 MBs per container, or a TRANSFERRATE of (1/25) * 1000 / 1024000 * 4096 = 0.16 milliseconds per page.

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

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

Experimentation becomes especially important in the environment mentioned above where you have a disk array as a container. You should create a simple query that moves data and use it in conjunction with a platform-specific measuring utility. You can then re-run the query with different container configurations within your table space. You can use the CREATE and ALTER TABLESPACE statements to change how data is transferred in your environment.

The I/O cost information provided through these two values could influence the optimizer in a number of ways, including whether or not to use an index to access the data, and which table to select for the inner and outer tables in a join.

- Prefetching

  When considering the I/O cost of accessing data from a table space, the optimizer also considers the potential impact that prefetching data and index

pages from disk can have on the query performance. Prefetching data and index pages can reduce the overhead and wait time associated with reading the data into the buffer pool.

The optimizer uses the information from the PREFETCHSIZE and EXTENTSIZE columns in SYSCAT.TABLESPACES to estimate the amount of prefetching that will occur for a table space.

- EXTENTSIZE can only be set when creating a table space (for example using the CREATE TABLESPACE statement). The default extent size is 32 pages (of 4 KB each) and is usually sufficient.
- PREFETCHSIZE can be set when you create a table space and or use the ALTER TABLESPACE statement. 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 to improve the data movement.

The following shows an example of the syntax to change the characteristics of the RESOURCE table space:

```
ALTER TABLESPACE RESOURCE
  PREFETCHSIZE 64
  OVERHEAD     19.3
  TRANSFERRATE 0.9
```

After making any changes to your table spaces, consider rebinding your applications and executing the RUNSTATS utility to collect the latest statistics about the indexes to ensure that the best access plans are used.

Server options affecting federated databases

A federated system is composed of a DB2 DBMS (the federated database) and one or more data sources. You identify the data sources to the federated database when you issue CREATE SERVER statements. When you issue these statements, you can include server options that refine and control aspects of federated system operations involving DB2 and the specified data source. To change server options later, use ALTER SERVER statements.

Note: You must install the distributed join installation option and set the database manager parameter *federated* to YES before you can create servers and specify server options.

The server option values that you specify affect query pushdown analysis, global optimization and other aspects of federated database operations. For example, in the CREATE SERVER statement, you can specify performance statistics as server option values, such as the *cpu_ratio* option, which specifies the relative speeds of the CPUs at the data source and the federated server. You might also set the *io_ratio* option to a value that indicates the relative rates of the data I/O divides at the source and the federated server. When you execute the CREATE SERVER statement, this data is added to the catalog view SYSCAT.SERVEROPTIONS, and the optimizer uses it in developing its access plan for the data source. If a statistic changes (as might happen, for instance, if the data source CPU is upgraded), use the ALTER SERVER statement to update SYSCAT.SERVEROPTIONS with this change. The optimizer then uses the new information the next time it chooses an access plan for the data source.

Chapter 20. Catalog statistics

When the query compiler optimizes the query plans, its decisions are heavily influenced by statistical information about the size of the database tables, indexes and statistical views. 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 join tables. The optimizer uses this information to estimate the costs of alternative access plans for each query.

In addition to table size and data distribution information, you can also collect 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. You use this information to decide when to reorganize tables and indexes.

When you collect statistics for a table in a partitioned database environment, statistics are only collected for that portion of the table that resides on the database partition where the utility is executed, or for the first database partition in the database partition group that contains the table. When you collect statistics for a statistical view, statistics are collected for all database partitions.

When you execute the RUNSTATS utility for a table, statistical view, or for a table and its associated indexes, the following kinds of statistical information are stored in the system catalog tables:

For a table and index:

- The number of pages in use
- The number of pages that contain rows
- The number of rows that overflow
- The number of rows in the table (cardinality)
- For MDC tables, the number of blocks that contain data
- For partitioned tables, the degree of data clustering within a single data partition

For each column in the table or statistical view and the first column in the index key:

- The cardinality of the column
- The average length of the column
- The second highest value in the columns
- The second lowest value in the column
- The number of NULLs in the column

For each XML column, the following statistical information are collected. Each row in an XML column stores an XML document. The node count of a given path or path-value pair refers to the number of nodes reachable by the path or path-value pair. The document count of a given path or path-value pair refers to the number of documents that contains the given path or path-value pair.

- The number of NULL XML documents
- The number of non-NULL XML documents
- The number of distinct paths

- The sum of the node count of each distinct path
- The sum of the document count of 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 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

For groups of columns that you specify:
- A timestamp based name for the column group
- The cardinality of the column group

For indexes only:
- The number of index entries (index cardinality)
- The number of leaf pages
- The number of index levels
- The degree of clustering of the table data to this index
- The degree of clustering of the index keys with regard to data partitions
- The ratio of leaf pages 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 number of pages on which all RIDs are marked deleted
- The number of RIDs marked deleted on pages on which not all RIDs are marked deleted

If you request detailed statistics for an index, you also store finer information about the degree of clustering of the table to the index and the page fetch estimates for different buffer sizes.

You can also collect the following kinds statistics about tables and indexes:
- Data distribution statistics

  The optimizer uses data distribution statistics to estimate efficient access plans for tables and statistical views in which data is not evenly distributed and columns have a significant number of duplicate values.
- Detailed index statistics

  The optimizer uses detailed index statistics to determine how efficient it is to access a table through an index.
- Sub-element statistics

The optimizer uses sub-element statistics for LIKE predicates, especially those that search for a pattern embedded within a string, such as `LIKE %disk%`.

Distribution statistics are not collected:

- When the *num_freqvalues* and *num_quantiles* configuration parameters are set to zero (0)
- When the distribution of data is known, such as when each data value is unique.
- When the column is a data type for which statistics are never collected. These data type are LONG, large object (LOB), or structured columns.
- 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

Note: You can perform a RUNSTATS on a declared temporary table, but the resulting statistics are not stored in the system catalogs because declared temporary tables do not have catalog entries. However, the statistics are stored in memory structures that represent the catalog information for declared temporary tables. In some cases, therefore, it might be useful to perform a RUNSTATS on these tables.

Automatic statistics collection

The DB2 optimizer uses catalog statistics to determine the most efficient access plan for any given query. Having out-of-date or incomplete statistics for a table or an index might lead the optimizer to select a plan that is not optimal, slowing down query execution. However, deciding which statistics to collect for a given workload is complex, and keeping these statistics up-to-date is time-consuming.

With automatic statistics collection, part of the DB2 automated table maintenance feature, you can let the DB2 database manager determine whether database statistics need to be updated. Automatic statistics collection can occur at statement compilation time using the real-time statistics feature or can be collected by running the RUNSTATS utility in the background. Background statistics collection can be enabled while real-time statistics collection is disabled. Background statistics collection must be enabled in order for real-time statistics collection to be enabled. Starting in DB2 Version 9, background automatic statistics collection is enabled by default when you create a new database. Starting in DB2 Version 9.5, both background automatic statistics collection and real-time statistics are enabled by default when you create a new database.

Understanding background and real-time statistics collection

Automatic statistics can be collected synchronously or asynchronously, by running the RUNSTATS utility. *Asynchronous* collection occurs in the background. When the real-time statistics feature is enabled, statistics can also be collected *synchronously*, at statement compilation time. When real-time statistics is enabled, statistics may also be fabricated using metadata maintained by the data and index manager. The query optimizer determines how the statistics should be collected, based on the needs of the query and the amount of table update activity. Table update activity is measured through the number of updates, deletes, and inserts.

Real-time statistics is determined by the needs of the SQL statement before it is optimized. This provides more timely statistics collection and more accurate statistics. Accurate statistics can result in better query execution plans and improved performance. When real-time statistics are not enabled, asynchronous statistics collection occurs at 2 hour intervals. This may not be frequent enough to provide accurate statistics for some applications.

When real-time statistics are enabled, asynchronous statistics collection checking will still occur at 2 hour intervals. Real-time statistics also initiates asynchronous collection requests when:

- The table activity isn't high enough to require synchronous collection, but is high enough to require asynchronous collection.
- The synchronous statistics collection used sampling because the table was large.
- The synchronous statistics were fabricated.
- The synchronous statistics collection failed because the collection time was exceeded.

At most, two asynchronous requests can be processed at the same time, but only for different tables. One request will have been initiated by real-time statistics, while the other will have been initiated by asynchronous statistics collection checking.

The performance impact of automatic statistics collection is minimized in several ways:

- Asynchronous statistics collection is performed using a throttled RUNSTATS utility. Throttling controls the amount of resources consumed by the RUNSTATS utility based on current database activity: as database activity increases, the RUNSTATS utility runs more slowly, reducing its resource demands.
- Synchronous statistics collection is limited to 5 seconds per query. This value can be controlled by the RTS optimization guideline. 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 later stored in the system catalogs by an asynchronous operation. This avoids the overhead and possible lock contention involved with updating the system catalogs. The statistics in the statistics cache are available to subsequent SQL compilation requests.
- Only one synchronous statistics collection will occur per table. Other agents requiring synchronous statistics collection will fabricate statistics, if possible, and continue with statement compilation. This behavior is also enforced in a DPF environment, where agents on different database partitions may require synchronous statistics.
- By default, statistics collected for synchronous and asynchronous operations are basic table statistics with distribution information and detailed index statistics using sampling. (The RUNSTATS command is issued with WITH DISTRIBUTION and SAMPLED DETAILED INDEXES ALL options.) You can customize the type of statistics collected by enabling statistics profiling, which uses information about previous database activity to determine which statistics are required by the database workload. You can also customize the type of statistics collected for a particular table, by creating your own statistics profile for that table.
- Only tables with missing statistics or high levels of activity (as measured through the number of updates, deletes, and inserts) are considered for statistics

collection. But even if the table meets the statistics collection criteria, synchronous statistics are not collected unless query optimization requires it. In some cases, the query optimizer can choose the access plan without statistics.

- For asynchronous statistics collection checking, large tables (consisting of more than 4000 pages) are sampled to determine whether the high table activity has indeed changed the statistics. Statistics for these large tables are collected only if warranted.
- For asynchronous statistics collection, the RUNSTATS utility is automatically scheduled to run during the optimal maintenance window specified in your maintenance policy definition. This policy also specifies the set of tables that are within the scope of the automatic statistics collection, further minimizing unnecessary resource consumption.
- Synchronous statistics collection and fabrication does not follow the maintenance window specified in your maintenance policy definition. This is because synchronous requests must occur immediately and have limited collection time. Synchronous statistics collection and fabrication follows the policy specifying the set of tables that are within the scope of the automatic statistics collection.
- While automated statistic collection is being performed, the affected tables are still available for regular database activity (updates, inserts, deletes) as if RUNSTATS command were not running on the table.
- For asynchronous statistics collection, the SYSPROC.NNSTAT stored procedure is run using the catalog-based collection method to refresh nicknames statistics automatically. Real-time statistics (synchronous or fabricated) are not collected for nicknames.

Real-time synchronous statistics collection is performed for regular tables, materialized query tables (MQTs), and Declared Global Temporary Tables (DGTTs).

Asynchronous statistics are not collected for DGTTs. This means that real-time statistics processing will not initiate asynchronous requests for DGTTs.

Automatic statistics collection (synchronous or asynchronous) does not occur for:

- statistical views
- tables that are marked VOLATILE (tables with VOLATILE field set in SYSCAT.TABLES)
- tables that have had their statistics manually updated, by issuing UPDATE statements directly against SYSSTAT catalog views

  If statistics have been modified manually, DB2 assumes that the user is maintaining statistics for a table. Therefore, DB2 will not maintain statistics for the table. In order to allow DB2 to maintain statistics for a table that has had its statistics manually updated, issue a manual RUNSTATS for the table. Migrated tables that have had statistics updated manually will have their statistics automatically maintained by DB2.

In a database partitioning feature (DPF) environment, statistics are collected on a single database partition and extrapolated. If a table exists on multiple database partitions and the table has no statistics, the DB2 database manager always collects statistics (both synchronous and asynchronous) on the first database partition of the database partition group. If the table has existing statistics, then statistics are collected on the database partition where statistics were last collected. This ensures consistent statistics, since they are always collected on the same database partition.

No real-time statistics collection activity will occur until at least 5 minutes after database activation.

When real-time statistics are enabled, you should schedule a defined maintenance window. By default, the maintenance window is undefined and includes all the time. If there is no defined time maintenance window, only synchronous statistics collection will occur. In this case, the collected statistics are only in-memory and typically collected using sampling, except for small tables.

Real-time statistics processing occurs for both static and dynamic SQL.

A table that has been truncated using the IMPORT command is automatically recognized as having stale statistics.

Automatic statistics collection, both synchronous and asynchronous, 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.

Real time statistics and explain processing

There is no real-time processing for a query which is just explained (not executed) using the explain facility. The following table summarizes the behavior for the different values of the CURRENT EXPLAIN MODE register.

Table 51. Real-time statistics gathering behavior for CURRENT EXPLAIN MODE register

CURRENT EXPLAIN MODE	Real-time statistics gathering considered
YES	YES
EXPLAIN	NO
NO	YES
REOPT	YES
RECOMMEND INDEXES	NO
EVALUATE INDEXES	NO

Automatic statistics collection and statistics cache

To make the synchronous collected statistics available to all queries, a statistics cache was introduced in DB2 Version 9.5. This cache is part of the catalog cache. In a database partitioning feature (DPF) environment, this cache resides only on the catalog database partition. The catalog cache can store multiple entries for the same SYSTABLES object, which will increase the size of the catalog cache size on all database partitions. Consider increasing the **CATALOGCACHE_SZ** database configuration parameter when real-time statistics is enabled.

Starting with DB2 Version 9, the Configuration Advisor is used to determine the initial database configuration for new databases. The Configuration Advisor will always recommend that the **AUTO_STMT_STATS** configuration parameter is set to ON.

Automatic statistics collection and statistical profiles

Synchronous and asynchronous statistics are collected according to a statistical profile that is in effect for a table, with the following exceptions:

- To minimize the overhead of synchronous statistics collection, the DB2 database management system may collect statistics using sampling. In this case, the sampling rate and method may be different from those specified in the statistical profile.
- Synchronous statistics collection may choose to fabricate statistics. It may not be possible to fabricate all statistics specified in the statistical profile. For example, column statistics such as COLCARD, HIGH2KEY and LOW2KEY cannot be fabricated unless the column is leading in some index.

If synchronous statistics collection cannot collect all statistics specified in the statistical profile, an asynchronous collection request will be submitted.

Although real-time statistics collection is designed to minimize statistics collection overhead, try it in a test environment first, to ensure there is no negative performance impact, which might be possible for some OLTP scenarios especially if there is an upper bound of how long a query should run.

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. You can optionally enhance this functionality in environments where a single database partition operates on a single processor (serial environment) by collecting query data and generating statistics profiles that help DB2 automatically collect the exact set of statistics that is required by your workload. This option is not available in MPP environments, certain federated environments, or environments in which intra-partition parallelism is enabled.

To enable automatic statistics collection:

1. Configure your database instance by using the Configure Automatic Maintenance wizard or the command line:
   - To use the Configure Automatic Maintenance wizard:
     a. Open the wizard either from the Control Center by right-clicking a database object or from the Health Center by right-clicking a database instance.
     b. Select **Configure Automatic Maintenance** from the pop-up window. Within this wizard, you can enable automatic statistics collection, specify the tables from which you want to automatically collect statistics, and specify a maintenance window for the execution of the RUNSTATS utility.
   - To use the command line, set each of the following configuration parameters to ON:
     – **AUTO_MAINT**
     – **AUTO_TBL_MAINT**
     – **AUTO_RUNSTATS**
2. Optional: To enable the automatic statistics profile generation, set the following two configuration parameters to ON:
   - **AUTO_STATS_PROF**
   - **AUTO_PROF_UPD**
3. Optional: To enable real-time statistics gathering, set the **AUTO_STMT_STATS** configuration parameter to ON. If this configuration parameter is set to ON, table statistics are automatically compiled at statement compilation time, whenever they are needed to optimize a query.

Storage used by automatic statistics collection and profiling

The automatic statistics collection and reorganization features store working data in tables in your database. These tables are created in the SYSTOOLSPACE table space. The SYSTOOLSPACE table space 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 should be calculated as approximately 1 KB per table. If this is a significant size for your database, you may want to drop and re-create the table space yourself and allocate storage appropriately. The automatic maintenance and health monitor tables in the table space are automatically re-created. Any history captured in those tables is lost when the table space is dropped.

Automatic statistics collection activity logging

In order to understand what statistics collection activities have occurred for a database, a statistics log has been introduced in DB2 Version 9.5. The statistics log records all statistics activities for a database, including automatic and manual statistics collection.

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. Statistics log behavior is controlled by the **DB2_OPTSTATS_LOG** registry variable.

The contents of the statistics log can be viewed directly or they can be queried using the SYSPROC.PD_GET_DIAG_HIST table function.

The SYSPROC.PD_GET_DIAG_HIST 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 52. Generic columns in the statistics log file

Column name	Data type	Description
OBJTYPE	VARCHAR(64)	The type of object the event applies to. For statistics logging, this is the type of statistics to be collected. It can also refer to a statistics collection background process when it is started or stopped. It can also refer to activities performed by automatics statistics collections such as a sampling test, initial sampling and table evaluation activity. Possible values for statistics collection actions are: **TABLE STATS** Table statistics will be collected. **INDEX STATS** Index statistics are to be collected. **TABLE AND INDEX STATS** Both table and index statistics will be collected. Possible values for automatic statistics operations are: **EVALUATION** The automatic statistics background collection process has begun an evaluation phase. During the evaluation phase, tables will be checked to determine if they need statistics and then 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 catalogs. This allows automatic statistics collection to quickly collect statistics for a table with no statistics. Subsequent statistics collections will collect unsampled statistics. 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 catalogs. The sampled statistics will be compared to the current catalog statistics to determine if full statistics should be gathered for this table and when they should be gathered. The sampling test is performed during the evaluation phase of automatic statistics collection. **STATS DAEMON** The statistics daemon is a background process used to process requests submitted by real-time statistics processing. This object type is logged when the background process is started and stopped.
OBJNAME	VARCHAR(255)	The name of the object the event applies to, if available. For the statistics log, this contains the table or index name. If the OBJTYPE is STATS DAEMON or EVALUATION, the OBJNAME is the database name and the OBJNAME_QUALIFIER is NULL.

Table 52. Generic columns in the statistics log file (continued)

Column name	Data type	Description
OBJNAME_QUALIFIER	VARCHAR(255)	For the statistics log, this contains the table or index schema.
EVENTTYPE	VARCHAR(24)	The event type is the action or verb 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** An attempt has been made to access a table for statistics collection purposes. This event type is used to log an unsuccessful access when the object is unavailable.
FIRST_EVENTQUALIFIERTYPE	VARCHAR(64)	The type of the first event qualifier. Event qualifiers are used to describe what was affected by the event. For the statistics log, the first event qualifier is the timestamp 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 when the statistics event occurred. The timestamp of the statistics event may 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 52. 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 using the RUNSTATS, LOAD, CREATE INDEX or REDISTRIBUTE commands. **Synchronous** Statistics collection was performed at SQL statement compilation time by DB2. The statistics are stored in the statistics cache but not the system catalogs. **Synchronous sampled** Statistics collection was performed using sampling at SQL statement compilation time by DB2. The statistics are stored in the statistics cache but not the system catalogs. **Fabricate** Statistics were fabricated at SQL statement compilation time using information maintained by the data and index manager. The statistics are stored in the statistics cache but not the system catalogs. **Fabricate partial** Only some statistics were fabricated at SQL statement compilation time using information maintained by the data and index manager. In particular, only the HIGH2KEY and LOW2KEY for certain columns were fabricated. The statistics are stored in the statistics cache but not the system catalogs. **Asynchronous** Statistics were collected by a background process by DB2 and have been stored in the system catalogs. 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 52. 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 may 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 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) event2,
substr(third_eventqualifiertype,1,6) event3_type,
substr(third_eventqualifier,1,15) 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_TYPE EVENT2               EVENT3_TYPE EVENT3         EVENTSTATE
----- ------------------ ---------- ------------------------- ------------------- ---------- -------------------------- ----------- -------------------- ----------- -------------- --------------------
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
```

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 to hold the log records and desired columns.

```
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 over 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 using the LOAD command and the load from cursor capability.

   ```
   load from c1 of cursor replace into db2user.stats_log;
   ```

4. Create indexes and 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;
   ```

Guidelines for collecting and updating statistics

The RUNSTATS command 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:

- When data has been loaded into a table and the appropriate indexes have been created.
- When you create a new index on a table. You need execute RUNSTATS for only the new index if the table has not been modified since you last ran RUNSTATS on it.
- When a table has been reorganized with the REORG utility.
- When the table and its indexes have been extensively updated, by data modifications, deletions, and insertions. (“Extensive” in this case might mean that 10 to 20 percent of the table and index data has been affected.)
- Before binding application programs whose performance is critical
- When you want to compare current and previous statistics. If you update statistics at regular intervals you can discover performance problems early.
- When the prefetch quantity is changed.
- When you have used the REDISTRIBUTE DATABASE PARTITION GROUP utility.

  Note: In previous versions of DB2, this command used the NODEGROUP keyword instead of the DATABASE PARTITION GROUP keywords.
- Use the RUNSTATS utility to collect statistics on XML columns. When RUNSTATS is used to collect statistics for XML columns only, existing statistics for non-XML columns that have been collected by LOAD or a previous execution of the RUNSTATS utility are retained. In the case where statistics on some XML columns have been collected previously, the previously collected statistics for an XML column will either be dropped if no statistics on that XML column are collected by the current command, or be replaced if statistics on that XML column are collected by the current command.

To improve RUNSTATS performance and save disk space used to store statistics, consider specifying only the columns for which data distribution statistics should be collected.

Ideally, you should rebind application programs after running statistics. The query optimizer might choose a different access plan if it has new statistics.

If you do not have enough time to collect all of the statistics at one time, you might run RUNSTATS to update statistics on only a few tables, indexes, or statistical views at a time, rotating through the set of objects. If inconsistencies occur as a result of activity on the table between the periods where you run RUNSTATS with a selective partial update, then a warning message (SQL0437W, reason code 6) is issued during query optimization. For example, this might occur if you execute RUNSTATS to gather table distribution statistics and, after some table activity, execute RUNSTATS again to gather index statistics on that table. If inconsistencies occur as a result of the activity on the table and these inconsistencies are detected during query optimization, the warning message is issued. When this happens, you should run RUNSTATS again to update the distribution statistics.

To ensure that the index statistics are synchronized with the table, execute RUNSTATS to collect both table and index statistics at the same time. Index statistics retain most of the table and column statistics collected from the last run of RUNSTATS. If the table has been modified extensively since the last time its table statistics were gathered, gathering only the index statistics for that table will leave the two sets of statistics out of synchronization on all nodes.

Invoking RUNSTATS on a production system might negatively impact the performance of the production workload. The RUNSTATS utility now supports a throttling option which 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 only collects statistics for tables on the database partition from which you execute it. The RUNSTATS results from this database partition are extrapolated to the other database partitions. If the database partition from which you execute RUNSTATS does not contain a portion of a particular table, the request is sent to the first database partition in the database partition group that contains a portion of the table.

When you collect statistics for a statistical view, statistics are collected for all database partitions containing base tables referenced by the view.

Consider these tips to improve the efficiency of RUNSTATS and the usefulness of the collected statistics:

- Collect statistics only for the columns used to join tables or in the WHERE, GROUP BY, and similar clauses of queries. If these columns are indexed, you can specify the columns with the ONLY ON KEY COLUMNS clause for the RUNSTATS command.
- Customize the values for *num_freqvalues* and *num_quantiles* for specific tables and specific columns in tables.
- Collect DETAILED index statistics with the SAMPLE DETAILED clause to reduce the amount of background calculation performed for detailed index statistics. The SAMPLE DETAILED clause reduces the time required to collect statistics, and produces adequate precision in most cases.

- When you create an index for a populated table, add 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, execute RUNSTATS again to update the statistics.
- Since RUNSTATS only collects statistics on a single database partition, the statistics will be less accurate if the data is not distributed consistently across all database partitions. If you suspect that there is skewed data distribution, you might want to redistribute the data across database partitions using the REDISTRIBUTE DATABASE PARTITION GROUP command before executing RUNSTATS.

Collecting catalog statistics

You collect catalog statistics on tables, indexes and statistical views to provide information that the optimizer uses to choose the best access plans for queries.

For tables and indexes, you must have one of the following:

- sysadm
- sysctrl
- sysmaint
- dbadm
- CONTROL privilege on the table
- LOAD authority

For statistical views, you must have one of the following:

- sysadm
- sysctrl
- sysmaint
- dbadm
- CONTROL privilege on the table

In addition, you must have privileges to access rows from the statistical view. Specifically, for each table, view or nickname referenced in the statistical view definition, you must have one of the following privileges:

- SYSADM or DBADM
- CONTROL
- SELECT

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. From the DB2 command line, execute the RUNSTATS command with appropriate options. These options allow you to tailor the statistics that are collected for the queries that run against the tables, indexes or statistical views.
3. When RUNSTATS is complete, issue a COMMIT statement to release locks.
4. Rebind packages that access tables, indexes or statistical views for which you have regenerated statistical information.

To use a graphical user interface to specify options and collect statistics, use the Control Center.

Note:

1. Because the RUNSTATS utility does not support use of nicknames, you update statistics differently for federated database queries. If queries access a federated database, execute RUNSTATS for the 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 collects statistics for tables on the database partition from which you execute it. The RUNSTATS results from this database partition are extrapolated to the other database partitions. If the database partition from which you execute RUNSTATS does not contain a portion of a particular table, the request is sent to the first database partition in the database partition group that contains a portion of the table.

   When you collect statistics for a statistical view, statistics are collected for all database partitions containing base tables referenced by the view.

Collecting distribution statistics for specific columns

For efficiency both of RUNSTATS and subsequent query-plan analysis, you might collect distribution statistics on only the columns that queries use in WHERE, GROUP BY, and similar clauses. You might 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 the group.

In the following steps, the database is assumed to be **sales** and to contain the table **customers**, with indexes **custidx1** and **custidx2**.

You must connect to the database that contains the tables and indexes and have one of the following authorization levels:

- sysadm
- sysctrl
- sysmaint
- dbadm
- CONTROL privilege on the table

Note: RUNSTATS only collects statistics for tables on the database partition from which you execute it. The RUNSTATS results from this database partition are extrapolated to the other database partitions. If the database partition from which you execute RUNSTATS does not contain a portion of a table, the request is sent to the first database partition in the database partition group that holds that portion of the table.

To collect statistics on specific columns:

1. Connect to the **sales** database.
2. Execute one of the following commands at 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 adjust the distribution defaults:

```
RUNSTATS ON TABLE sales.customers
     WITH DISTRIBUTION ON
     COLUMNS (zip, ytdtotal NUM_FREQVALUES 50 NUM_QUANTILES 75)
```

- To collect distribution statistics on the columns indexed in **custidx1** and **custidx2**:

```
RUNSTATS ON TABLE sales.customer
         ON KEY COLUMNS
```

- To collect column statistics on the table only for specific 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 statistics for non-XML columns have been collected using the LOAD command with the STATISTICS option. To complement the non-XML statistics with the statistics for the XML column **miscinfo**:

```
RUNSTATS ON TABLE sales.customers
   ON COLUMNS (miscinfo)
```

- To collect column statistics on the table for non-XML columns only (the EXCLUDING XML COLUMNS option takes precedence over all other clauses that may specify XML columns):

```
RUNSTATS ON TABLE sales.customers
   EXCLUDING XML COLUMNS
```

You can also use the Control Center to collect distribution statistics.

Collecting index statistics

Collect index statistics to allow the optimizer to evaluate whether an index should be used to resolve a query.

In the following steps, the database is assumed be **sales** and to contain the table **customers**, with indexes **custidx1** and **custidx2**.

You must connect to the database that contains the tables and indexes and have one of the following authorization levels:

- sysadm
- sysctrl
- sysmaint
- dbadm
- CONTROL privilege on the table

Executing RUNSTATS with the SAMPLED DETAILED option requires 2MB of the statistics heap. Allocate an additional 488 4K pages to the *stat_heap_sz* database configuration parameter setting for this additional memory requirement. If the heap appears to be too small, RUNSTATS returns an error before it attempts to collect statistics.

To collect detailed statistics for an index:

1. Connect to the **sales** database.
2. Execute one of the following commands at the DB2 command line, depending on your requirements:
   - To create detailed statistics on both **custidx1** and **custidx2**:

```
RUNSTATS ON TABLE sales.customers AND DETAILED INDEXES ALL
```

- To create detailed statistics on both indexes, but use sampling instead of performing detailed calculations for each index entry:

```
RUNSTATS ON TABLE sales.customers AND SAMPLED DETAILED INDEXES ALL
```

- To create detailed sampled statistics on 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
```

You can also use the Control Center to collect index and table statistics.

Collecting statistics on a sample of the table data

Table statistics are used by the query optimizer in selecting the best access plan for any given query, so it is important that statistics remain current to accurately reflect the state of a table at any given time. As the activity against a table increases, so should the frequency of statistics collection. With the increasing size of databases, it is becoming more important to find efficient ways to collect statistics. Random sampling of table data on which to collect statistics can improve RUNSTATS performance. For I/O bound or CPU bound systems, the performance benefits can be enormous. The smaller the sample, the faster RUNSTATS completes.

Starting in Version 8.2, the RUNSTATS command provides the option to collect statistics on a sample of the data in the table by using the TABLESAMPLE option. This feature can increase the efficiency of statistics collection since sampling uses only a subset of the data. At the same time, the sampling methods ensure a high level of accuracy.

There are two ways to specify how the sample is to be collected. The BERNOULLI method samples the data at the level of the row. During a full table scan of the data pages each row is considered in turn and is selected based on probability P as specified by the numeric parameter. It is only on these selected rows that statistics will be collected. In a similar manner, the SYSTEM method samples the data at the page-level. Thus, each page is selected on probability P and rejected with probability 1-P/100.

Performance of page-level sampling is excellent because only one I/O is required for each selected page. With row-level sampling, I/O costs are not reduced since every table page is retrieved in a full table scan. However, row-level sampling provides significant improvements, even if the amount of I/O is not reduced, because gathering statistics is CPU intensive.

Row-level sampling will provide a better sample than page-level sampling in situations where the data values are highly clustered. Compared to page sampling, the row-level sample set will likely be a better reflection of the data since it will include P percent rows from each data page. In page-level sampling all the rows of P percent pages will be in the sample set. If the rows are distributed randomly over the table, then the accuracy of row sampled statistics will be similar to the accuracy of page sampled statistics.

Each sample is randomly generated across RUNSTATS commands unless the REPEATABLE option is used. With the REPEATABLE clause, the same sample will be generated as in the last execution of the RUNSTATS command with the

TABLESAMPLE option. Users may find this beneficial in cases where the generation of consistent statistics is desired for tables of constant data.

Collecting statistics using a statistics profile

The RUNSTATS utility provides an option to register and use a statistics profile, which is a set of options that specify which statistics are to be collected on a particular table, for example, table statistics, index statistics, or distribution statistics.

This feature simplifies statistics collection by allowing you to store the options that you specify when you issue the RUNSTATS command so that you can collect the same statistics repeatedly on a table without having to re-type the command options.

You can register or update a statistics profile with or without actually collecting statistics. For example, to register a profile and collect statistics at the same time, issue the RUNSTATS command with the SET PROFILE option. To register a profile only, without actually collecting statistics, issue the RUNSTATS command with the SET PROFILE ONLY option.

To collect statistics using a statistics profile that you have already registered, issue the RUNSTATS command, specifying only the name of the table and the USE PROFILE option.

To see what options are currently specified in the statistics profile for a particular table, you can query the catalog tables with the following select statement, where `tablename` is the name of the table that you want the profile for:

```
SELECT STATISTICS_PROFILE FROM SYSIBM.SYSTABLES WHERE NAME = tablename
```

Automatic statistics profiling

Statistics profiles can also be generated automatically by the DB2 automatic statistics profiling feature. When this feature is enabled, information about database activity is collected and stored it in a query feedback warehouse. Based on this data, a statistics profile is generated. Enabling this feature can alleviate the problem of uncertainty about which statistics are relevant to a particular workload and permits the collection of the minimal set of statistics to provide optimal database workload performance.

This feature can be used with the automatic statistics collection feature, which automatically schedules statistics maintenance based on the information contained within the automatically generated statistics profile.

To enable this feature, you need to have already enabled automatic table maintenance by setting the appropriate configuration parameters . The AUTO_STATS_PROF configuration parameter activates the collection of query feedback data, and the AUTO_PROF_UPD configuration parameter activates the generation of a statistics profile for use by automatic statistics collection.

Note: Automatic statistics profile generation can only be activated in DB2 serial mode, and is blocked for queries in certain federated environments, multi-partition MPP environments, and where intra-partition parallelism is enabled.

Statistics profile generation is best suited to environments running large complex queries that apply many predicates, often having correlations in the data of the

predicate columns, and joining and grouping over several tables. It is less suitable to environments with a primarily transactional workload.

There are a few different ways to use this feature:

- In a test environment. Set AUTO_STATS_PROF and AUTO_PROF_UPD to ON in test systems, where the performance overhead of runtime monitoring can be easily tolerated. When the test system uses realistic data and queries, this will allow for learning the proper correlations and settings of statistics parameters for RUNSTATS, which then will be stored in the statistics profiles. These profiles can then be transferred to the production system, where queries can benefit without incurring any monitoring overhead.
- To address performance issues for specific queries in a production environment. If performance problems for a particular set of queries is detected and can be attributed to faulty statistics or correlations, you can turn AUTO_STATS_PROF on and execute the target workload for a period of time. Automatic statistics profiling will analyze the query feedback and create recommendations in the SYSTOOLS.OPT_FEEDBACK_RANKING* tables. You can inspect these recommendations and refine the statistics profiles manually based on the recommendations. To have DB2 automatically update the statistics profiles based on these recommendations, turn AUTO_PROF_UPD on when you turn AUTO_STATS_PROF on.

Note: There is some performance overhead associated with monitoring the queries and storing the query feedback data in the feedback warehouse.

Creating the query feedback warehouse

The feedback warehouse consists of five tables in the SYSTOOLS schema that store information about the predicates encountered during query execution and recommendations for statistics collection. The five tables are OPT_FEEDBACK_QUERY, OPT_FEEDBACK_PREDICATE, OPT_FEEDBACK_PREDICATE_COLUMN, OPT_FEEDBACK_RANKING, and OPT_FEEDBACK_RANKING_COLUMN.

To use automatic statistics profiling, you need to first create the query feedback warehouse using the SYSINSTALLOBJECTS stored procedure. This stored procedure is the common stored procedure for creating and dropping objects in the SYSTOOLS schema.

Invoke the SYSINSTALLOBJECTS stored procedure as follows:

```
call SYSINSTALLOBJECTS ( toolname, action, tablespacename, schemaname)
```

where:

toolname
: Specifies the name of the tool whose objects are to be created or dropped. In this case 'ASP' or 'AUTO STATS PROFILING'.

action
: Specifies the action to be taken: 'C' for create, 'D' for drop.

tablespacename
: The name of the table space in which the the feedback warehouse tables will be created. This input parameter is optional. If it is not specified, the default user space will be used.

schemaname
The name of the schema with which the objects will be created or dropped. This parameter is currently not used.

For example, to create the feedback warehouse in table space ″A″ enter: `call SYSINSTALLOBJECTS ('ASP', 'C', 'A', '')`

Catalog statistics tables

The following tables provide information about the system catalog tables that contain catalog statistics and the RUNSTATS options that collect specific statistics.

Table 53. 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 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 new CARD statistics are updated. The previous CARD statistics are retained.

Table 54. Column Statistics (SYSCAT.COLUMNS and SYSSTAT.COLUMNS)

Statistic	Description	RUNSTATS Option	
		Table	**Indexes**
COLCARD	column cardinality	Yes	Yes (Note 1)
AVGCOLLEN	average length of column	Yes	Yes (Note 1)
HIGH2KEY	second highest value in column	Yes	Yes (Note 1)
LOW2KEY	second lowest value in column	Yes	Yes (Note 1)
NUMNULLS	the number of NULLs in a column	Yes	Yes (Note 1)
SUB_COUNT	the average number of subelements	Yes	No (Note 2)
SUB_DELIM_LENGTH	average length of each delimiter separating each subelement	Yes	No (Note 2)

Note:

1. Column statistics are gathered for the first column in the index key.
2. These statistics provide information about data in columns that contain a series of subfields or subelements that are delimited by blanks. The SUB_COUNT and SUB_DELIM_LENGTH statistics are collected only for single-byte character set string columns of type CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC.

Table 55. Multicolumn Statistics (SYSCAT.COLGROUPS and SYSSTAT.COLGROUPS)

Statistic	Description	RUNSTATS Option	
		Table	**Indexes**
COLGROUPCARD	cardinality of the column group	Yes	No

Note: The multicolumn distribution statistics listed in the following two tables are not collected by RUNSTATS. You cannot update them manually.

Table 56. Multicolumn 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 57. Multicolumn Distribution Statistics 2 (SYSCAT.COLGROUPDISTCOUNTS and SYSSTAT.COLGROUPDISTCOUNTS)

Statistic	Description	RUNSTATS Option	
		Table	**Indexes**
TYPE	F = frequency value Q = quantile value	Yes	No
SEQNO	Sequence number *n* that represents the *n*th TYPE value	Yes	No
VALCOUNT	If TYPE = F, VALCOUNT is the number of occurrences of COLVALUEs for the column group identified by this SEQNO. If TYPE = Q, VALCOUNT is the number of rows whose value is less than or equal to COLVALUEs for the column group with this SEQNO.	Yes	No

Table 57. Multicolumn Distribution Statistics 2 (SYSCAT.COLGROUPDISTCOUNTS and SYSSTAT.COLGROUPDISTCOUNTS) (continued)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
DISTCOUNT	If TYPE = Q, this column contains the number of distinct values that are less than or equal to COLVALUEs for the column group with this SEQNO. Null if unavailable.	Yes	No

Table 58. 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 occupied by the index (Note 3)	No	Yes
FIRSTKEYCARD	number of distinct values in first column of the index	No	Yes
FIRST2KEYCARD	number of distinct values in first two columns of the index	No	Yes
FIRST3KEYCARD	number of distinct values in first three columns of the index	No	Yes
FIRST4KEYCARD	number of distinct values in first four columns of the index	No	Yes
FULLKEYCARD	number of distinct values in all columns of the index, excluding any key value in a type-2 index for which all 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 based on 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)

Table 58. Index Statistics (SYSCAT.INDEXES and SYSSTAT.INDEXES) (continued)

Statistic	Description	RUNSTATS Option	
		Table	**Indexes**
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 accessible in sequence. This is the number of index pages that the prefetchers can detect as being in sequence.	No	Yes
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 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 record identifiers (RIDs) in the index, including deleted RIDs in type-2 indexes.	No	Yes
NUMRIDS_DELETED	the total number of RIDs marked deleted in the index, except RIDs on leaf pages on which all record identifiers are marked deleted	No	Yes
NUM_EMPTY_LEAFS	the total number of leaf pages on which all record identifiers are marked deleted	No	Yes
INDCARD	Number of index entries (index cardinality)	No	Yes

Table 58. Index Statistics (SYSCAT.INDEXES and SYSSTAT.INDEXES) (continued)

Statistic	Description	RUNSTATS Option	
		Table	Indexes

Note:

1. Detailed index statistics are gathered 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 a respectable size. If the table is greater than about 25 pages, then CLUSTERFACTOR and PAGE_FETCH_PAIRS statistics are collected. In this case, CLUSTERRATIO is -1 (not collected). If the table is a relatively small table, only CLUSTERRATIO is filled in by RUNSTATS while CLUSTERFACTOR and PAGE_FETCH_PAIRS are not. If the DETAILED clause is not specified, only the CLUSTERRATIO statistic is collected.
3. This statistic measures the percentage of pages in the file containing the index that belongs to that table. For a table having only one index defined on it, DENSITY should normally 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. These statistics cannot be computed when this table is in a DMS table space.
5. Prefetch statistics will not be collected during a LOAD or CREATE INDEX even if statistics collection is specified when the command is invoked. Prefetch statistics are also not collected if the Sequential Detection Flag configuration parameter (seqdetect) is turned off.
6. When RUNSTATS options for Table is ″No″ it means statistics are not collected when table statistics are collected, and ″Yes″ for Indexes means statistics are collected when the RUNSTATS command is used with the INDEXES options.

Table 59. Column Distribution Statistics (SYSCAT.COLDIST and SYSSTAT.COLDIST)

Statistic	Description	RUNSTATS Option	
		Table	Indexes
DISTCOUNT	If TYPE is Q, the number of distinct values that are less than or equal to COLVALUE statistics	Distribution (Note 2)	No
TYPE	Indicator of whether 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 frequency or quantile statistic is collected	Distribution	No
VALCOUNT	Frequency with which the data value occurs in column, or for quantiles, the number of values less than or equal to the data value (COLVALUE)	Distribution	No

Note:

1. Column distribution statistics are gathered by specifying the WITH DISTRIBUTION clause on the RUNSTATS command. Note that distribution statistics may **not** be gathered 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.

Distribution statistics

You can collect two kinds of data distribution statistics:

- Frequency statistics

  These statistics provide information about the column and the data value with the highest number of duplicates, the next highest number of duplicate values, and so on to the level specified by the value of the *num_freqvalues* database configuration parameter. To disable collection of frequent-value statistics, set *num_freqvalues* to 0.

  You can also set *num_freqvalues* as RUNSTATS options for each table or statistical view and for specific columns.

- Quantile statistics

  These 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 below 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 Kth position from the low end of the range.

  To specify the number of sections 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 an optimizer estimation error of a maximum of plus or minus 2.5% for any equality or less-than or greater-than predicate and a maximum error of plus or minus 5% for any BETWEEN predicate. To disable collection of quantile statistics, set num_quantiles to 0 or 1.

  You can set *num_quantiles* for each table or statistical view, and for specific columns.

Note: If you specify larger *num_freqvalues* and *num_quantiles* values, more CPU resources and memory, as specified by the *stat_heap_sz* database configuration parameter, are required when you execute RUNSTATS.

When to collect distribution statistics To decide whether distribution statistics should be created and updated for a given table or statistical view, consider the following two factors:

- Whether applications use static or dynamic SQL and XQuery statements.

  Distribution statistics are most useful for dynamic queries and static queries that do not use host variables. When using queries with host variables, the optimizer makes limited use of distribution statistics.

- Whether data in columns is distributed uniformly.

  Create distribution statistics if at least one column in the table has a highly "non-uniform" 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 non-uniform data distributions might occur, possibly together:

- Data might be clustered in one or more sub-intervals 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):

C1

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 deal with this kind of data distribution.

To help determine whether column data is not uniformly distributed, execute a query such as the following example:

```
SELECT C1, COUNT(*) AS OCCURRENCES
  FROM T1
GROUP BY C1
ORDER BY OCCURRENCES DESC;
```

- Duplicate data values might occur often. Consider a column in which data is distributed with the following frequencies:

Data Value	Frequency
20	5
30	10
40	10
50	25
60	25
70	20
80	5

To help the optimizer deal with duplicate values, create both quantile and frequent-value statistics.

When to collect index statistics only

You might collect statistics based only on index data in the following situations:

- A new index has been created since the RUNSTATS utility was run and you do not want to collect statistics again on the table data.
- There have been many changes to the data that affect the first column of an index.

What level of statistical precision to specify

To determine the precision with which distribution statistics are stored, you specify the database configuration parameters, *num_quantiles* and *num_freqvalues*. You can also specify these parameters as RUNSTATS options when you collect statistics for a table or for columns. The higher you set these values, the greater precision RUNSTATS uses when it create and updates distribution statistics. However, greater precision requires greater use of resources, both during RUNSTATS execution and in the storage required in the catalog tables.

For most databases, specify between 10 and 100 for the *num_freqvalues* database configuration parameter. Ideally, frequent-value statistics should be created such that the frequencies of the remaining values are either approximately equal to each other or negligible compared to the frequencies of the most frequent values. The

database manager might collect less than this number, because these statistics will only be collected for data values that occur more than once. If you need to collect only quantile statistics, set *num_freqvalues* to zero.

To set the number of quantiles, specify between 20 and 50 as the setting of the *num_quantiles* database configuration parameter. A rough rule of thumb for determining the number of quantiles is:

- Determine the maximum error that is tolerable in estimating the number of rows of any range query, as a percentage, P
- The number of quantiles should be approximately 100/P if the predicate is a BETWEEN predicate, and 50/P if the predicate is any other type of range predicate (<, <=, >, or >=).

For example, 25 quantiles should result in a maximum estimate error of 4% for BETWEEN predicates and of 2% for ″>″ predicates. In general, specify at least 10 quantiles. More than 50 quantiles should be necessary only for extremely non-uniform data. If you need only frequent value statistics, set *num_quantiles* to zero. If you set this parameter to “1”, because the entire range of values fits in one quantile, no quantile statistics are collected.

Optimizer use of distribution statistics

The optimizer uses distribution statistics for better estimates of the cost of various possible access plans to satisfy queries.

If you do not execute RUNSTATS with the WITH DISTRIBUTION clause, the catalog statistics tables contain information only about the size of the table or statistical view and the highest and lowest values in the table or statistical view, the degree of clustering of the table to any of its indexes, and the number of distinct values in indexed columns.

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:

The optimizer needs to estimate the number of rows containing a column value that satisfies an equality or range predicate in order to select the least expensive access plan. The more accurate the estimate, the greater the likelihood that the optimizer will choose the optimal access plan. For example, consider the query

```
SELECT C1, C2
  FROM TABLE1
  WHERE C1 = 'NEW YORK'
  AND C2 <= 10
```

Assume that there is an index on both C1 and C2. One possible access plan is to use the index on C1 to retrieve all rows with `C1 = 'NEW YORK'` and then check each retrieved row to see if `C2 <= 10`. An alternate plan is to use the index on C2 to retrieve all rows with `C2 <= 10` and then check each retrieved row to see if `C1 = 'NEW YORK'`. Because the primary cost in executing the query is usually the cost of retrieving the rows, the best plan is the plan that requires the fewest retrievals. Choosing this plan requires estimating the number of rows that satisfy each predicate.

When distribution statistics are not available but RUNSTATS has been executed against a table or a statistical view, the only information available to the optimizer is the second-highest data value (HIGH2KEY), second-lowest data value (LOW2KEY), number of distinct values (COLCARD), and number of rows (CARD) for a column. The number of rows that satisfy an equality or range predicate is then estimated under the assumption that the frequencies of the data values in a column are all equal and the data values are evenly spread out over the interval (LOW2KEY, HIGH2KEY). Specifically, the number of rows satisfying an equality predicate `C1 = KEY` is estimated as CARD/COLCARD, and the number of rows satisfying a range predicate `C1 BETWEEN KEY1 AND KEY2` is estimated as:

```
      KEY2 - KEY1
------------------  x CARD       (1)
 HIGH2KEY - LOW2KEY
```

These estimates are accurate only when the true distribution of data values in a column is reasonably uniform. When distribution statistics are unavailable and either the frequencies of the data values differ widely from each other or the data values are clustered in a few sub-intervals of the interval (LOW_KEY,HIGH_KEY), the estimates can be off by orders of magnitude and the optimizer may choose a less than optimal access plan.

When distribution statistics are available, the errors described above can be greatly reduced by using frequent-value statistics to compute the number of rows that satisfy an equality predicate and using frequent-value statistics and quantiles to compute the number of rows that satisfy a range predicate.

Extended examples of distribution-statistics use

To understand how the optimizer might use distribution statistics, consider first a query that contains an equality predicate of the form `C1 = KEY`.

Example for Frequent-Value Statistics

If frequent-value statistics are available, the optimizer can use these statistics to choose an appropriate access plan, as follows:

- If `KEY` is one of the N most frequent values, then 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 as:

```
CARD - NUM_FREQ_ROWS
--------------------          (2)
   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`) for which the frequency of the data values is as follows:

Data Value	Frequency
1	2
2	3

Data Value	Frequency
3	40
4	4
5	1

If frequent-value statistics based on only the most frequent value (that is, N = 1) are available, for this column, the number of rows in the table is 50 and the column cardinality is 5. For the predicate `C1 = 3`, exactly 40 rows satisfy it. If the optimizer assumes that data is evenly distributed, it estimates the number of rows that satisfy the predicate as 50/5 = 10, with an error of -75%. If the optimizer can use frequent-value statistics, the number of rows is estimated as 40, with no error.

Consider another example in which 2 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%. You may use the following formula to calculate the estimation error (as a percentage):

```
estimated rows  - actual rows
----------------------------  X 100
          actual rows
```

Using the frequent value statistics (N = 1), the optimizer will estimate the number of rows containing this value using the formula (2) given above, for example:

```
(50 - 40)
--------- = 3
 (5 - 1)
```

and the error is reduced by an order of magnitude as shown below:

```
 3 - 2
------- = 50%
   2
```

Example for Quantile Statistics

The following explanations of quantile statistics use 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 less than or equal to `V`. To computer a K-quantile, sort the rows in the column according to increasing data values; the K-quantile is the data value in the Kth 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 (`C`) that contains the following values:

C
0.0
5.1
6.3
7.1
8.2
8.4
8.5
9.1

93.6
100.0

and 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

First consider the predicate `C <= 8.5`. For the data given above, exactly 7 rows satisfy this predicate. Assuming a uniform data distribution and using formula (1) from above, with KEY1 replaced by LOW2KEY, the number of rows that satisfy the predicate is estimated as:

```
 8.5 - 5.1
---------- x 10 *= 0
93.6 - 5.1
```

where *= means “approximately equal to”. The error in this estimation is approximately -100%.

If quantile statistics are available, the optimizer estimates the number of rows that satisfy this same predicate (`C <= 8.5`) by locating 8.5 as the highest value in one of the quantiles and estimating the number of rows by using the corresponding value of K, which is 7. In this case, the error is reduced to 0.

Now consider the predicate `C <= 10`. Exactly 8 rows satisfy this predicate. If the optimizer must assume a uniform data distribution and use formula (1), 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 above 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 improves the accuracy of the estimates in the above examples because the real data values are ″clustered″ in the range 5 - 10, but the standard estimation formulas assume that the data values are spread out 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
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

Also suppose that frequent value statistics are available based on the 3 most frequent values.

Consider the predicate `C BETWEEN 20 AND 30`. From the distribution of the data values, you can see that exactly 10 rows satisfy this predicate. Assuming a uniform data distribution and using formula (1), the number of rows that satisfy the predicate is estimated as:

```
30 - 20
-------  x 100 = 25
70 - 30
```

which has an error of 150%.

Using frequent-value statistics and quantiles, 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 (2), `r_1` is estimated as:

```
100 - 80
-------- = 5
  7 - 3
```

Using linear interpolation, `r_2` is estimated as:

```
  30 - 20
  ------- x (# rows with value > 20 and <= 40)
  40 - 20
  30 - 20
= ------- x (25 - 5)
  40 - 20
= 10,
```

yielding a final estimate of 15 and reducing the error by a factor of 3.

Detailed index statistics

If you execute RUNSTATS for indexes with the DETAILED clause, you collect statistical information about indexes that allows the optimizer to estimate how many data page fetches will be required, based on various buffer-pool sizes. This additional information helps the optimizer make better estimates of the cost of accessing a table through an index.

Note: When you collect detailed index statistics, RUNSTATS takes longer and requires more memory and CPU processing. The SAMPLED DETAILED option, for which information calculated only for a statistically significant number of entries, requires 2MB of the statistics heap. Allocate an additional 488 4K pages to the *stat_heap_sz* database configuration parameter setting for this additional memory requirement. If the heap appears to be too small, RUNSTATS returns an error before attempting to collect statistics.

The DETAILED statistics PAGE_FETCH_PAIRS and CLUSTERFACTOR will be collected only if the table is of a sufficient size: around 25 pages. In this case, CLUSTERFACTOR will be a value between 0 and 1; and CLUSTERRATIO will be -1 (not collected). For tables smaller than 25 pages, CLUSTERFACTOR will be -1 (not collected), and CLUSTERRATIO will be a value between 0 and 100; even if the DETAILED clause is specified for an index on that table.

The DETAILED statistics provide concise information about the number of physical I/Os required to access the data pages of a table if a complete index scan is performed under different buffer sizes. As RUNSTATS scans the pages of the index, it models the different buffer sizes, and gathers estimates of how often a page fault occurs. For example, if only one buffer page is available, each new page referenced by the index results in a page fault. In a worse case, each row might reference a different page, resulting in at most the same number of I/Os as 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), then all table pages are read once. As a result, the number of physical I/Os is a monotone, non-increasing 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 the table rows are clustered in relation to the index, 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.

You should collect DETAILED index statistics when queries reference columns that are not included in the index. In addition, DETAILED index statistics should be used in the following circumstances:

- The table has multiple unclustered indexes with varying degrees of clustering
- The degree of clustering is non-uniform among the key values
- The values in the index are updated non-uniformly

It is difficult to evaluate these conditions without previous knowledge or without forcing an index scan under varying buffer sizes and then monitoring the physical I/Os that result. Probably the cheapest way to determine whether any of these situations occur is to collect the DETAILED statistics for an index, examine them, and retain them if the PAGE_FETCH_PAIRS that result are non-linear.

Sub-element statistics

If tables contain columns that contain sub-fields or sub-elements separated by blanks, and queries reference these columns in WHERE clauses, you should collect sub-element statistics to ensure the best access plans.

For example, suppose a database contains a table, DOCUMENTS, in which each row describes a document, and suppose that in DOCUMENTS there is a column called KEYWORDS that contains a list of relevant keywords relating to this document for text retrieval purposes. The values in KEYWORDS might be as follows:

```
'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 5 sub-elements, each of which is a word (the keyword), separated from the others by one blank.

For queries that specify LIKE predicates on such columns using the % match_all character:

```
SELECT .... FROM DOCUMENTS WHERE KEYWORDS LIKE '%simulation%'
```

it is often beneficial for the optimizer to know some basic statistics about the sub-element structure of the column.

The following statistics are collected when you execute RUNSTATS with the LIKE STATISTICS clause:

SUB_COUNT
The average number of sub-elements.

SUB_DELIM_LENGTH
The average length of each delimiter separating each sub-element, where a delimiter, in this context, is one or more consecutive blank characters.

In the KEYWORDS column example, SUB_COUNT is 5, and SUB_DELIM_LENGTH is 1, because each delimiter is a single blank character.

The DB2_LIKE_VARCHAR registry variable affects the way in which the optimizer deals with a predicate of the form:

```
COLUMN LIKE '%xxxxxx'
```

where *xxxxxx* is any string of characters; that is, any LIKE predicate whose search value starts with a % character. (It might or might not end with a % character). These are referred to as ″wildcard LIKE predicates″. For all predicates, the optimizer has to estimate how many rows match the predicate. For 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.

To examine the values of the sub-element statistics, query SYSIBM.SYSCOLUMNS. For example:

```
select substr(NAME,1,16), SUB_COUNT, SUB_DELIM_LENGTH
   from sysibm.syscolumns where tbname = 'DOCUMENTS'
```

Note: RUNSTATS might take longer if you use the LIKE STATISTICS clause. For example, RUNSTATS might take between 15% and 40%, and longer on a table with five character columns, if the DETAILED and DISTRIBUTION options are not used. If either the DETAILED or the DISTRIBUTION option is specified, the overhead percentage is less, even though the absolute amount of overhead is the same. If you are considering using this option, you should assess this overhead against improvements in query performance.

Catalog statistics that users can update

Statistics for user-defined functions

To create statistical information for user-defined functions (UDFs), you edit the SYSSTAT.FUNCTIONS catalog view. If UDF statistics are available, the optimizer can use them when it estimates costs for various access plans. The RUNSTATS utility does not collect statistics for UDFs. If statistics are not available the statistic column values are -1 and the optimizer uses default values that assume a simple UDF.

The following table provides information about the statistic columns for which you can provide estimates to improve performance:

Table 60. Function Statistics (SYSCAT.ROUTINES and SYSSTAT.FUNCTIONS)

Statistic	Description
IOS_PER_INVOC	Estimated number of read/write requests executed each time a function is executed.
INSTS_PER_INVOC	Estimated number of machine instructions executed each time a function is executed.
IOS_PER_ARGBYTE	Estimated number of read/write requests executed per input argument byte.
INSTS_PER_ARGBYTES	Estimated number of machine instructions executed per input argument byte.
PERCENT_ARGBYTES	Estimated average percent of input argument bytes that the function will actually process.
INITIAL_IOS	Estimated number of read/write requests executed only the first/last time the function is invoked.
INITIAL_INSTS	Estimated number of machine instructions executed only the first/last time the function is invoked.
CARDINALITY	Estimated number of rows generated by a table function.

For example, consider a UDF (EU_SHOE) that converts an American shoe size to the equivalent European shoe size. (These two shoe sizes could be UDTs.) For this UDF, you might set the statistic columns as follows:

- INSTS_PER_INVOC: set to the estimated number of machine instructions required to:
  - Invoke 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: set each to 0, since 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 a UDF (LOCATE) 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, on average, 75 percent of the second argument is searched. Based on this information, PERCENT_ARGBYTES should be set to 75. The above estimate of the average of 75 percent is based on the following additional assumptions:

- 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/write requests that are performed only the first or last time the function is invoked, such as to record the cost of setting up a scratchpad area.

To obtain information about I/Os and instructions used by a user-defined function, you can use output provided by your programming language compiler or by monitoring tools available for your operating system.

Catalog statistics for modeling and what-if planning

You can change the statistical information in the system catalogs so that it does not reflect the actual state of tables and indexes but allows you to examine various possible changes to the database for planning purposes. The ability to update selected system catalog statistics allows 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. If you do, the optimizer might not choose the best access plan for production queries that contain dynamic SQL or XQuery statements.

Requirements

You must have explicit DBADM authority for the database to modify statistics for tables and indexes and their components. That is, your user ID is recorded as having DBADM authority in the SYSCAT.DBAUTH table. Belonging to a DBADM group does not explicitly provide this authority. A DBADM can see statistics rows for all users, and can execute SQL UPDATE statements against the views defined in the SYSSTAT schema to update the values of these statistical columns.

A user without DBADM authority can see only those rows which contain statistics for objects over which they have CONTROL privilege. If you do not have DBADM authority, you can change statistics for individual database objects if you have the following privileges for each object:

- Explicit CONTROL privilege on tables. You can also update statistics for columns and indexes for these tables.
- Explicit CONTROL privilege on nicknames in a federated database system. You can also update statistics for columns and indexes for these nicknames. Note that the update only affects local metadata (data-source table statistics are not changed). These updates affect only the global access strategy generated by the DB2 optimizer.
- Ownership of user-defined functions (UDFs)

The following shows an example of updating the table statistics for the EMPLOYEE table:

```
UPDATE SYSSTAT.TABLES
SET  CARD   = 10000,
     NPAGES = 1000,
     FPAGES = 1000,
     OVERFLOW = 2
WHERE TABSCHEMA = 'userid'
  AND TABNAME   = 'EMPLOYEE'
```

You must be careful when manually updating catalog statistics. Arbitrary changes can seriously alter the performance of subsequent queries. Even in a non-production database that you are using for testing or modeling, you can use any of the following methods to refresh updates you applied to these tables and bring the statistics to a consistent state:

- ROLLBACK the unit of work in which the changes have been made (assuming the unit of work has not been committed).
- Use the RUNSTATS utility to recalculate and refresh the catalog statistics.
- Update the catalog statistics to indicate that statistics have not been gathered. (For example, setting column NPAGES to -1 indicates that the number-of-pages statistic has not been collected.)
- Replace the catalog statistics with the data they contained before you made any changes. This method is possible only if you used the *db2look* tool to capture the statistics before you made any changes.

In some cases, the optimizer may determine that some particular statistical value or combination of values is not valid. It will use default values and issue a warning. Such circumstances are rare, however, since most of the validation is done when updating the statistics.

Statistics for modeling production databases

Sometimes you may want your test system to contain a subset of your production system's data. However, access plans selected for such a test system are not necessarily the same as those that would be selected on the production system, unless the catalog statistics and the configuration parameters for the test system are updated to match those of the production system.

A productivity tool, *db2look*, can be run against the production database to generate the update statements required to make the catalog statistics of the test database match those in production. These update statements can be generated by using *db2look* in mimic mode (`-m` option). In this case, *db2look* will generate a command processor script containing all the statements required to mimic the catalog

statistics of the production database. This can be useful when analyzing SQL or XQuery statements through Visual Explain in a test environment.

You can recreate database data objects, including tables, views, indexes, and other objects in a database, by extracting DDL statements with *db2look -e*. You can run the command processor script created from this command against another database to recreate the database. You can use `-e` option and the `-m` option together in a script that re-creates the database and sets the statistics.

After running the update statements produced by `db2look` against the test system, the test system can be used to validate the access plans to be generated in production. Since the optimizer uses the type and configuration of the table spaces to estimate I/O costs, the test system must have the same table space geometry or layout. That is, the same number of containers of the same type, either SMS or DMS.

The *db2look* tool is found under the *bin* subdirectory.

For more information on how to use this productivity tool, type the following on a command line:

```
db2look -h
```

The Control Center also provides an interface to the *db2look* utility called "Generate DDL". Using the Control Center allows the results file from the utility to be integrated into the Script Center. You can also schedule the *db2look* command from the Control Center. One difference when using the Control Center is that only single table analysis can be done as opposed to a maximum of thirty tables in a single call using the *db2look* command. You should also be aware that LaTex and Graphical outputs are not supported from the Control Center.

You can also run the `db2look` utility against an OS/390 or z/OS database. The `db2look` utility extracts the DDL and UPDATE statistics statements for OS/390 objects. This is very useful if you would like to extract OS/390 or z/OS objects and re-create them in DB2 Database for Linux, UNIX, and Windows.

There are some differences between statistics for DB2 Database for Linux, UNIX, and Windows, and the OS/390 statistics. The `db2look` utility performs the appropriate conversions from DB2 for OS/390 or z/OS to DB2 Database for Linux, UNIX and Windows when this is applicable and sets to a default value (-1) the DB2 Database for Linux, UNIX and Windows statistics for which a DB2 for OS/390 counterpart does not exist. Here is how the `db2look` utility maps the DB2 for OS/390 or z/OS statistics to those of DB2 Database for Linux, UNIX and Windows. In the discussion below, "UDB_x" stands for a DB2 Database for Linux, UNIX and Windows statistics column; and, "S390_x" stands for a DB2 for OS/390 or z/OS statistics column.

1. Table Level Statistics.

   UDB_CARD = S390_CARDF
   UDB_NPAGES = S390_NPAGES

   There is no S390_FPAGES. However, DB2 for OS/390 or z/OS has another statistics called PCTPAGES which represents the percentage of active table space pages that contain rows of the table. So it is possible to calculate UDB_FPAGES based on S390_NPAGES and S390_PCTPAGES as follows:

   ```
   UDB_FPAGES=(S390_NPAGES * 100)/S390_PCTPAGES
   ```

There is no S390_OVERFLOW to map to UDB_OVERFLOW. Therefore, the db2look utility just sets this to the default value:

```
UDB_OVERFLOW=-1
```

2. Column Level Statistics.

   UDB_COLCARD = S390_COLCARDF
   UDB_HIGH2KEY = S390_HIGH2KEY
   UDB_LOW2KEY = S390_LOW2KEY

   There is no S390_AVGCOLLEN to map to UDB_AVGCOLLEN so the db2look utility just sets this to the default value:

   ```
   UDB_AVGCOLLEN=-1
   ```

3. Index Level Statistics.

   UDB_NLEAF = S390_NLEAF
   UDB_NLEVELS = S390_NLEVELS
   UDB_FIRSTKEYCARD= S390_FIRSTKEYCARD
   UDB_FULLKEYCARD = S390_FULLKEYCARD
   UDB_CLUSTERRATIO= S390_CLUSTERRATIO

   The other statistics for which there are no OS/390 or z/OS counterparts are just set to the default. That is:

   ```
   UDB_FIRST2KEYCARD = -1
   UDB_FIRST3KEYCARD = -1
   UDB_FIRST4KEYCARD  = -1
   UDB_CLUSTERFACTOR = -1
   UDB_SEQUENTIAL_PAGES = -1
   UDB_DENSITY  = -1
   ```

4. Column Distribution Statistics.

   There are two types of statistics in DB2 for OS/390 or z/OS SYSIBM.SYSCOLUMNS. Type “F” for frequent values and type “C” for cardinality. Only entries of type “F” are applicable to DB2 Database for Linux, UNIX, and Windows and these are the ones that will be considered.

   ```
   UDB_COLVALUE = S390_COLVALUE
   UDB_VALCOUNT = S390_FrequencyF * S390_CARD
   ```

   In addition, there is no column SEQNO in DB2 for OS/390 SYSIBM.SYSCOLUMNS. Because this required for DB2, db2look generates one automatically.

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 the statistics. It is also important to preserve the consistency of relationships between 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). Assume that you want to reduce COLCARD from 100 to 25, and CARD from 200 to 50. If you update SYSSTAT.TABLES first, you should get an error (since CARD would be less than COLCARD). The correct order is to update COLCARD in SYSSTAT.COLUMNS first, then update CARD in SYSSTAT.TABLES. The situation occurs in reverse if you want to increase COLCARD to 250 from 100, and CARD to 300 from 200. In this case, you must update CARD first, then COLCARD.

When a conflict is detected between an updated statistic and another statistic, an error is issued. However, errors may not always be issued when conflicts arise. In some situations, the conflict is difficult to detect and report in an error, especially if the two related statistics are in different catalogs. For this reason, you should be careful to avoid causing such conflicts.

The most common checks you should make, before updating a catalog statistic, are:

1. Numeric statistics must be -1 or greater than or equal to zero.
2. Numeric statistics representing percentages (for example, CLUSTERRATIO in SYSSTAT.INDEXES) must be between 0 and 100.

Note: For row types, the table level statistics NPAGES, FPAGES, and OVERFLOW are not updateable for a sub-table.

When a table is first created, system catalog statistics are set to -1 to indicate the table has no statistics. Until statistics are gathered, DB2 uses default values for SQL or XQuery statement compilation and optimization. Updating the table or index statistics may fail if the new values are inconsistent with the default values. Therefore it is recommended that you perform RUNSTATS after creating a table, before attempting to update the table or index statistics.

Rules for updating column statistics manually

When you are updating statistics in SYSSTAT.COLUMNS, follow the guidelines below.

- When manually updating HIGH2KEY and LOW2KEY in SYSSTAT.COLUMNS, follow the behavior of the generated values:
  - The values for HIGH2KEY, LOW2KEY must be valid values for the data type of the corresponding user column.
  - The length of HIGH2KEY, LOW2KEY values must be the smaller of 33 or the maximum length of the target column data type, not including additional quotes which can bring the length of the string up 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, LOW2KEY values.
  - The HIGH2KEY/LOW2KEY values are stored in such a way that they can be used on the SET clause of an UPDATE statement and without manipulation on cost calculations. For character strings, this means single quotes are added to the beginning and end of the string and an extra quote is added for every quote already in the string. Examples of user column values and their corresponding values in the HIGH2KEY,LOW2KEY are given in the table below.

Table 61. HIGH2KEY and LOW2KEY values for datatypes

Datatype in user column	User data	Corresponding HIGH2KEY, LOW2KEY value
INTEGER	-12	-12
CHAR	abc	'abc'
CHAR	ab'c	'ab''c'

  - HIGH2KEY should be greater than LOW2KEY whenever there are more than three distinct values in the corresponding column.
- The cardinality of a column (COLCARD statistic in SYSSTAT.COLUMNS) cannot be greater than the cardinality of its corresponding table or statistical view (CARD statistic in SYSSTAT.TABLES).

- The number of nulls in a column (NUMNULLS statistic in SYSSTAT.COLUMNS) cannot be greater than the cardinality of its corresponding table or statistical view (CARD statistic in SYSSTAT.TABLES).
- No statistics are supported for columns with data types: LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB.

Rules for updating distribution statistics manually

You update distribution statistics manually only to model a production database or perform what-if tests on an artificially constructed database. Do not update distribution statistics on a production database.

Make sure that all the statistics in the catalog are consistent. Specifically, for each column, the catalog entries for the frequent data statistics and quantiles must satisfy the following constraints:

- Frequent value statistics (in the SYSSTAT.COLDIST catalog). These constraints include:
  - The values in column VALCOUNT must be unchanging or decreasing for increasing values of SEQNO.
  - The number of values in column COLVALUE must be less than or equal to the number of distinct values in the column, which is stored in column COLCARD in catalog view SYSSTAT.COLUMNS.
  - The sum of the values in column VALCOUNT must be less than or equal to the number of rows in the column, which is stored in column CARD in catalog view SYSSTAT.TABLES.
  - In most cases, the values in the column COLVALUE should lie between the second-highest and second-lowest data values for the column, which are stored in columns HIGH2KEY and LOW2KEY, respectively, in catalog view SYSSTAT.COLUMNS. There may be one frequent value greater than HIGH2KEY and one frequent value less than LOW2KEY.
- Quantiles (in the SYSSTAT.COLDIST catalog). These constraints include:
  - The values in column COLVALUE must be unchanging or decreasing for increasing values of SEQNO
  - The values in column VALCOUNT must be increasing for increasing values of SEQNO
  - The largest value in column COLVALUE must have a corresponding entry in column VALCOUNT equal to the number of rows in the column
  - In most cases, the values in the column COLVALUE should lie between the second-highest and second-lowest data values for the column, which are stored in columns HIGH2KEY and LOW2KEY, respectively, in catalog view SYSSTAT.COLUMNS.

Suppose that distribution statistics are available for a column C1 with “R” rows and you wish to modify the statistics to correspond to a column with the same relative proportions of data values, but with “(F x R)” rows. To scale up the frequent-value statistics by a factor of F, each entry in column VALCOUNT must be multiplied by F. Similarly, to scale up the quantiles by a factor of F, each entry in column VALCOUNT must be multiplied by F. If you do not follow these rules, the optimizer might use the wrong filter factor and cause unpredictable performance when you run the query.

Rules for updating table and nickname statistics manually

The only statistical values that you can update in SYSSTAT.TABLES are CARD, FPAGES, NPAGES, and OVERFLOW, and for MDC tables, ACTIVE_BLOCKS. Keep in mind that:

1. CARD must be greater than or equal to all COLCARD values in SYSSTAT.COLUMNS that correspond to that table.
2. CARD must be greater than NPAGES.
3. FPAGES must be greater than NPAGES.
4. NPAGES must be less than or equal to any ″Fetch″ value in the PAGE_FETCH_PAIRS column of any index (assuming this statistic is relevant for the index).
5. CARD must not be less than or equal to any ″Fetch″ value in the PAGE_FETCH_PAIRS column of any index (assuming this statistic is relevant for the index).

When working within a federated database system, use caution when manually providing or updating statistics on a nickname over a remote view. The statistical information, such as the number of rows this nickname will return, might not reflect the real cost to evaluate this remote view and thus might mislead the DB2 optimizer. Situations that can benefit from statistics updates include remote views defined on a single base table with no column functions applied on the SELECT list. Complex views may require a complex tuning process which might require that each query be tuned. Consider creating local views over nicknames instead so the DB2 optimizer knows how to derive the cost of the view more accurately.

Rules for updating index statistics manually

When you update the statistics in SYSSTAT.INDEXES, follow the rules described below:

1. PAGE_FETCH_PAIRS (in SYSSTAT. INDEXES) must adhere to the following rules:
   - Individual values in the PAGE_FETCH_PAIRS statistic must be separated by a series of blank delimiters.
   - Individual values in the PAGE_FETCH_PAIRS statistic must not be longer than 10 digits and must be less than the maximum integer value (MAXINT = 2147483647).
   - There must always be a valid PAGE_FETCH_PAIRS value if the CLUSTERFACTOR is greater than zero.
   - There must be exactly 11 pairs in a single PAGE_FETCH_PAIR statistic.
   - Buffer size entries of PAGE_FETCH_PAIRS must be ascending in value.
   - Any buffer size value in a PAGE_FETCH_PAIRS entry cannot be greater than MIN(NPAGES, 524287) for 32-bit operating system or MIN(NPAGES, 2147483647) for 64-bit operating system where NPAGES is the number of pages in the corresponding table (in SYSSTAT.TABLES).
   - “Fetches” entries of PAGE_FETCH_PAIRS must be descending in value, with no individual “Fetches” entry being less than NPAGES. “Fetch” size values in a PAGE_FETCH_PAIRS entry cannot be greater than the CARD (cardinality) statistic of the corresponding table.
   - If buffer size value is the same in two consecutive pairs, then page fetch value must also be the same in both the pairs (in SYSSTAT.TABLES).

   A valid PAGE_FETCH_UPDATE 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
```

2. CLUSTERRATIO and CLUSTERFACTOR (in SYSSTAT.INDEXES) must adhere to the following rules:
   - 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_PAIR statistic.
3. For relational indexes the following rules apply to FIRSTKEYCARD, FIRST2KEYCARD, FIRST3KEYCARD, FIRST4KEYCARD, FULLKEYCARD, and INDCARD:
   - FIRSTKEYCARD must be equal to FULLKEYCARD for a single-column index.
   - FIRSTKEYCARD must be equal to COLCARD (in SYSSTAT.COLUMNS) for the corresponding column.
   - If any of these index statistics are not relevant, you should set them to -1. For example, if you have an index with only 3 columns, set FIRST4KEYCARD to -1.
   - For multiple column indexes, if all the statistics are relevant, the relationship between them must be:

     ```
     FIRSTKEYCARD <= FIRST2KEYCARD <= FIRST3KEYCARD <= FIRST4KEYCARD
        <= FULLKEYCARD <= INDCARD == CARD
     ```
4. For indexes over XML data the following rules apply to FIRSTKEYCARD, FIRST2KEYCARD, FIRST3KEYCARD, FIRST4KEYCARD, FULLKEYCARD, and INDCARD:
   - The relationship between them must be:

     ```
     FIRSTKEYCARD <= FIRST2KEYCARD <= FIRST3KEYCARD <= FIRST4KEYCARD
        <= FULLKEYCARD <= INDCARD
     ```
5. 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.

Chapter 21. Routines

Guidelines for stored procedures

Stored procedures permit one call to a remote database to execute a preprogrammed procedure in a database application environment in which many situations are repetitive. For example, for receiving a fixed set of data, performing the same set of multiple requests against a database, or returning a fixed set of data might represent several accesses to the database.

Processing a single SQL statement for a remote database requires sending two transmissions: one request and one receive. Because an application contains many SQL statements it requires many transmissions to complete its work.

However, when a IBM data server client uses a stored procedure that encapsulates many SQL statements, it requires only two transmissions for the entire process.

Stored procedures usually run in processes separate from the database agents. This separation requires the stored procedure and agent processes to communicate through a router. However, a special kind of stored procedure that runs in the agent process might improve performance, although it carries significant risks of corrupting data and databases.

These risky stored procedures are those created as *not fenced*. For a not-fenced stored procedure, nothing separates the stored procedure from the database control structures that the database agent uses. If a DBA wants to ensure that the stored procedure operations will not accidentally or maliciously damage the database control structures, the *not fenced* option is omitted.

Because of the risk of damaging your database, use *not fenced* stored procedures **only** when you need the maximum possible performance benefits. In addition, make absolutely sure that the procedure is well coded and has been thoroughly tested before allowing it to run as a not-fenced stored procedure. If a fatal error occurs while running a not-fenced stored procedure, the database manager determines whether the error occurred in the application or database manager code and performs the appropriate recovery.

A not-fenced stored procedure can corrupt the database manager beyond recovery, possibly resulting in lost data and the possibility of a corrupt database. Exercise extreme caution when you run not-fenced trusted stored procedures. In almost all cases, the proper performance analysis of an application results in the good performance without using not-fenced stored procedures. For example, triggers might improve performance.

Improving the performance of SQL procedures

Overview of how DB2 compiles SQL PL and inline SQL PL

Before discussing how to improve the performance of SQL procedures we should discuss how DB2 compiles them upon the execution of the CREATE PROCEDURE statement.

When an SQL procedure is created, DB2 separates the SQL queries in the procedure body from the procedural logic. To maximize performance, the SQL queries are statically compiled into sections in a package. For a statically compiled query, a section consists mainly of the access plan selected by the DB2 optimizer for that query. A package is a collection of sections. For more information on packages and sections, please refer to the DB2 SQL Reference. The procedural logic is compiled into a dynamically linked library.

During the execution of a procedure, every time control flows from the procedural logic to an SQL statement, there is a "context switch" between the DLL and the DB2 engine. As of DB2 Version 8.1, SQL procedures run in "unfenced mode". That is they run in the same addressing space as the DB2 engine. Therefore the context switch we refer to here is not a full context switch at the operating system level, but rather a change of layer within DB2. Reducing the number of context switches in procedures that are invoked very often, such as procedures in an OLTP application, or that process large numbers of rows, such as procedures that perform data cleansing, can have a noticeable impact on their performance.

Whereas an SQL procedure containing SQL PL is implemented by statically compiling its individual SQL queries into sections in a package, an inline SQL PL function is implemented, as the name suggests, by inlining the body of the function into the query that uses it. Queries in SQL functions are compiled together, as if the function body were a single query. The compilation occurs every time a statement that uses the function is compiled. Unlike what happens in SQL procedures, procedural statements in SQL functions are not executed in a different layer than dataflow statements. Therefore, there is no context switch every time control flows from a procedural to a dataflow statement or vice versa.

If there are no side-effects in your logic use an SQL function instead

Because of the difference in compilation between SQL PL in procedures and inline SQL PL in functions, it is reasonable to presume that a piece of procedural code will execute faster in a function than in a procedure if it only queries SQL data and does no data modifications - that is it has no side-effects on the data in the database or external to the database.

That is only good news if all the statements that you need to execute are supported in SQL functions. SQL functions can not contain SQL statements that modify the database. As well, only a subset of SQL PL is available in the inline SQL PL of functions. For example, you cannot execute CALL statements, declare cursors, or return result sets in SQL functions.

Here is an example of an SQL procedure containing SQL PL that was a good candidate for conversion to an SQL function to maximize performance:

```
CREATE PROCEDURE GetPrice (IN Vendor CHAR&(20&),
                           IN Pid INT, OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN
  IF Vendor eq; ssq;Vendor 1ssq;
    THEN SET price eq; (SELECT ProdPrice
                        FROM V1Table
                        WHERE Id = Pid);
  ELSE IF Vendor eq; ssq;Vendor 2ssq;
    THEN SET price eq; (SELECT Price FROM V2Table
                        WHERE Pid eq; GetPrice.Pid);
  END IF;
END
```

Here is the rewritten SQL function:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
RETURNS  DECIMAL(10,3)
LANGUAGE SQL
BEGIN
  DECLARE price DECIMAL(10,3);
  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice
                             FROM V1Table
                             WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table
                WHERE Pid = GetPrice.Pid);
  END IF;
  RETURN price;
END
```

Remember that the invocation of a function is different than a procedure. To invoke the function, use the VALUES statement or invoke the function where an expression is valid, such as in a SELECT or SET statement. Any of the following are valid ways of invoking the new function:

```
VALUES (GetPrice('IBM', 324))

SELECT VName FROM Vendors WHERE GetPrice(Vname, Pid) < 10

SET price = GetPrice(Vname, Pid)
```

Avoid multiple statements in an SQL PL procedure when just one is sufficient

Although it is generally a good idea to write concise SQL, it is very ease to forget to do this in practice. For example the following SQL statements:

```
INSERT INTO tab_comp VALUES (item1, price1, qty1);
INSERT INTO tab_comp VALUES (item2, price2, qty2);
INSERT INTO tab_comp VALUES (item3, price3, qty3);
```

can be rewritten as a single statement:

```
INSERT INTO tab_comp VALUES (item1, price1, qty1),
                            (item2, price2, qty2),
                            (item3, price3, qty3);
```

The multi-row insert will require roughly one third of the time required to execute the three original statements. Isolated, this improvement might seem negligible, but if the code fragment is executed repeatedly, for example in a loop or in a trigger body, the improvement can be significant.

Similarly, a sequence of SET statements like:

```
SET A = expr1;
SET B = expr2;
SET C = expr3;
```

can be written as a single VALUES statement:

```
VALUES expr1, expr2, expr3 INTO A, B, C;
```

This transformation preserves the semantics of the original sequence if there are no dependencies between any two statements. To illustrate this, consider:

```
SET A = monthly_avg * 12;
SET B = (A / 2) * correction_factor;
```

Converting the previous two statements to:

```
VALUES (monthly_avg * 12, (A / 2) * correction_factor) INTO A, B;
```

does not preserve the original semantics because the expressions before the INTO keyword are evaluated 'in parallel'. This means that the value assigned to *B* is not based on the value assigned to *A*, which was the intended semantics of the original statements.

Reduce multiple SQL statements to a single SQL expression

Like other programming languages, the SQL language provides two types of conditional constructs: procedural (IF and CASE statements) and functional (CASE expressions). In most circumstances where either type can be used to express a computation, using one or the other is a matter of taste. However, logic written using CASE expressions is not only more compact, but also more efficient than logic written using CASE or IF statements.

Consider the following fragment of SQL PL code:

```
IF (Price <= MaxPrice) THEN
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, Price);
ELSE
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, MaxPrice);
END IF;
```

The condition in the IF clause is only being used to decide what value is inserted in the tab_comp.Val column. To avoid the context switch between the procedural and the dataflow layers, the same logic can be expressed as a single INSERT with a CASE expression:

```
INSERT INTO tab_comp(Id, Val)
       VALUES(Oid,
            CASE
               WHEN (Price <= MaxPrice) THEN Price
               ELSE MaxPrice
            END);
```

It's worth noting that CASE expressions can be used in any context where a scalar value is expected. In particular, they can be used on the right-hand side of assignments. For example:

```
IF (Name IS NOT NULL) THEN
  SET ProdName = Name;
ELSEIF (NameStr IS NOT NULL) THEN
  SET ProdName = NameStr;
ELSE
  SET ProdName = DefaultName;
END IF;
```

can be rewritten as:

```
SET ProdName = (CASE
                  WHEN (Name IS NOT NULL) THEN Name
                  WHEN (NameStr IS NOT NULL) THEN NameStr
                  ELSE  DefaultName
                END);
```

In fact, this particular example admits an even better solution:

```
SET ProdName = COALESCE(Name, NameStr, DefaultName);
```

Don't underestimate the benefit of taking the time to analyze and consider rewriting your SQL. The performance benefits will pay you back many times over

for the time invested in analyzing and rewriting your procedure.

Exploit the set-at-a-time semantics of SQL

Procedural constructs such as loops, assignment and cursors allow us to express computations that would not be possible to express using just SQL DML statements. But when we have procedural statements at our disposal, there is a risk that we could turn to them even when the computation at hand can, in fact, be expressed using just SQL DML statements. As we've mentioned earlier, the performance of a procedural computation can be orders of magnitude slower than the performance of an equivalent computation expressed using DML statements. Consider the following fragment of code:

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_sel VALUES (20, v2);
  ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

To begin with, the loop body can be improved by applying the transformation discussed in the last section - "Reduce multiple SQL statements to a single SQL expression":

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  INSERT INTO tab_sel VALUES (CASE
                                WHEN v1 > 20 THEN 20
                                ELSE v1
                              END, v2);
  FETCH cur1 INTO v1, v2;
END WHILE;
```

But upon closer inspection, the whole block of code can be written as an INSERT with a sub-SELECT:

```
INSERT INTO tab_sel (SELECT (CASE
                               WHEN col1 > 20 THEN 20
                               ELSE col1
                             END),
                             col2
                     FROM tab_comp);
```

In the original formulation, there was a context switch between the procedural and the dataflow layers for each row in the SELECT statements. In the last formulation, there is no context switch at all, and the optimizer has a chance to globally optimize the full computation.

On the other hand, this dramatic simplification would not have been possible if each of the INSERT statements targeted a different table, as shown below:

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_default VALUES (20, v2);
```

```
  ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

However, the set-at-a-time nature of SQL can also be exploited here:

```
INSERT INTO tab_sel (SELECT col1, col2
                     FROM tab_comp
                     WHERE col1 <= 20);
INSERT INTO tab_default (SELECT col1, col2
                         FROM tab_comp
                         WHERE col1 > 20);
```

When looking at improving the performance of existing procedural logic, any time spent in eliminating cursor loops will likely pay off.

Keep the DB2 optimizer informed

When a procedure is created, its individual SQL queries are compiled into sections in a package. The DB2 optimizer chooses an execution plan for a query based, among other things, on table statistics (for example, table sizes or the relative frequency of data values in a column) and indexes available at the time the query is compiled. When tables suffer significant changes, it may be a good idea to let DB2 collect statistics on these tables again. And when statistics are updated or new indexes are created, it may also be a good idea to rebind the packages associated with SQL procedures that use the tables, to let DB2 create plans that exploit the latest statistics and indexes.

Table statistics can be updated using the RUNSTATS command. To rebind the package associated with an SQL procedure, you can use the REBIND_ROUTINE_PACKAGE built-in procedure that is available in DB2 Version 8.1. For example, the following command can be used to rebind the package for procedure MYSCHEMA.MYPROC:

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'MYSCHEMA.MYPROC', 'ANY')
```

where 'P' indicates that the package corresponds to a procedure and 'ANY' indicates that any of the functions and types in the SQL path are considered for function and type resolution. See the Command Reference entry for the REBIND command for more details.

Use arrays

You can use arrays to efficiently pass collections of data between applications and stored procedures and to store and manipulate transient collections of data within SQL procedures without having to use relational tables. Operators on arrays available within SQL procedures allow for the efficient storage and retrieval of data. Applications that create arrays of moderate size will experience significantly better performance than applications that create very large arrays (on the scale of multiple megabytes), as the entire array is stored in main memory. See *Related links* section for additional information.

Chapter 22. Query access plans

The SQL and XQuery compiler process

The SQL and XQuery compiler performs several steps to produce an access plan that can be executed. These steps are shown in the following figure and described in the sections below the figure. Note that some steps occur only for queries in a federated database.

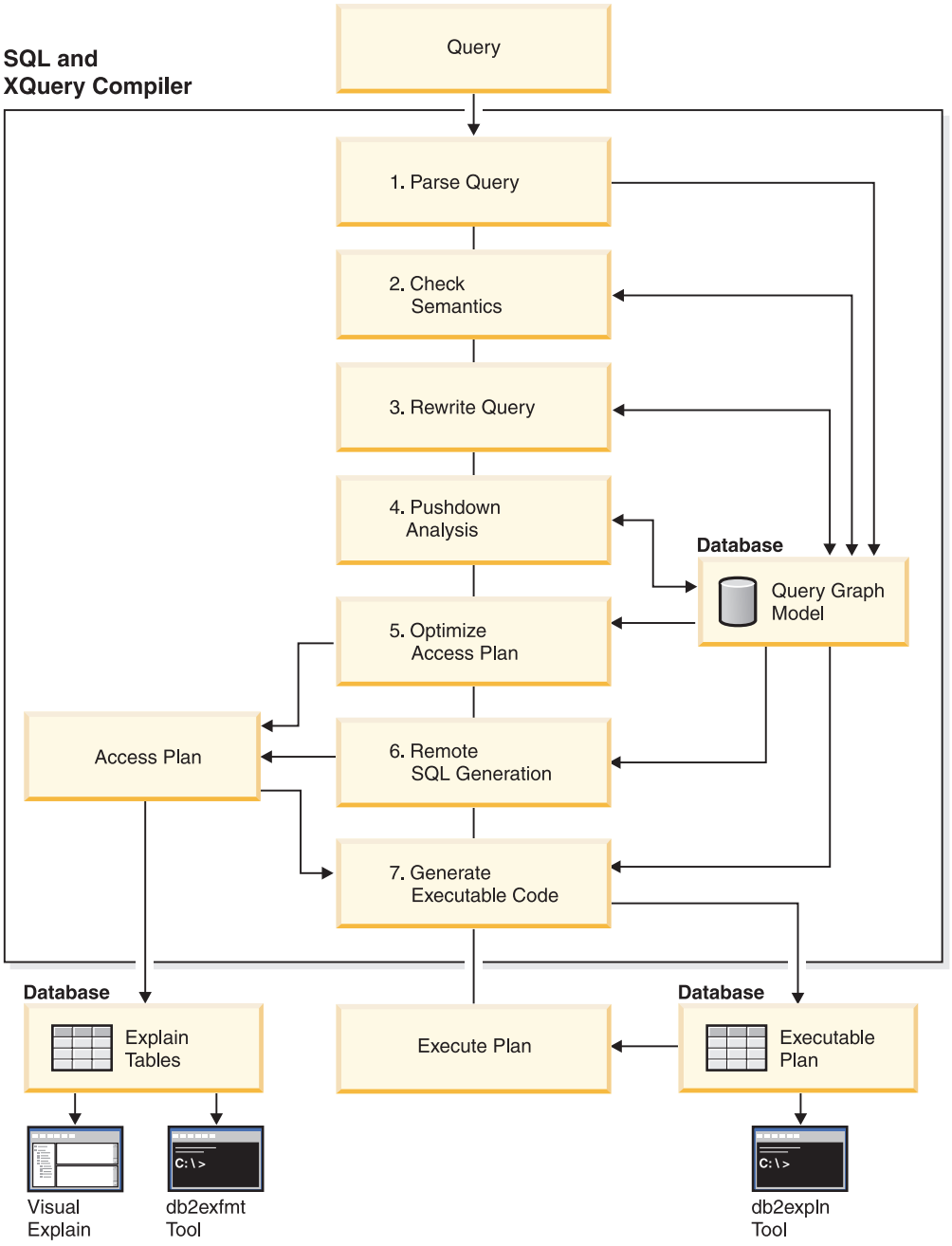


Figure 22. Steps performed by SQL and XQuery compiler

Query Graph Model

The *query graph model* is an internal, in-memory database that represents the query as it is processed in the steps described below:

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 the 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. As a simple example of semantic checking, the compiler verifies that the data type of the column specified for the YEAR scalar function is a datetime data type.

   The compiler also adds the 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 of 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 stored in the query graph model to transform the query into a form that can be optimized more easily and stores the result in the query graph model.

   For example, the compiler might move a predicate, altering the level at which it is applied and 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 of 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)**

   The major task in this step is to recommend 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.

   This step is bypassed unless you are executing federated database queries.

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 alternative plan, the optimizer uses the statistics for tables, indexes, columns and functions. Then it 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 component.

The optimizer also considers whether there are different sized buffer pools when determining page size selection. That the environment includes a partitioned database is also considered as well as the ability to enhance the chosen plan for the possibility of intra-query parallelism in a symmetric multi-processor (SMP) environment. This information is used by the optimizer to help select the best access plan for the query.

The output of this step of the compiler is an access plan. This access plan provides the information captured in the Explain tables. The information used to generate the access plan can be captured with an explain snapshot.

6. **Remote SQL Generation (Federated Databases)**

   The final plan selected by the optimizer might consist of a set of steps that operate on a remote data source. For operations that are performed by each data source, the remote SQL generation step creates an efficient SQL statement based on the data-source SQL dialect.

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 for a query. Examples for which this optimization is possible include code page conversions and the use of host variables.

   To enable query (re)optimization of static and dynamic SQL and XQuery statements that have host variables, special registers, or parameter markers, bind the package with the REOPT bind option. If used, the access path for an SQL or XQuery statement, belonging to that package and containing host variables, parameter markers or special registers, will be optimized using the values of these variables rather than default estimates 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 the package is executed, the database manager will use the information stored in the system catalog tables to determine how to access the data and provide results for the query. This information is used by the *db2expln* tool.

Note: Execute RUNSTATS at appropriate intervals on tables that change often. The optimizer needs up-to-date statistical information about the 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 suspects that RUNSTATS was executed on empty or nearly empty tables, it may either use defaults or attempt to derive certain statistics based on the number of file pages used to store the table on disk (FPAGES). The total number of occupied blocks is stored in the ACTIVE_BLOCKS column.

Query rewriting methods and examples

During the rewrite query stage, the query compiler transforms SQL and XQuery statements into forms that can be optimized more easily, and as a result, can improve the possible access paths. Rewriting queries is particularly important for very complex queries, including those queries with 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, use the Explain facility or Visual Explain.

Queries might be rewritten in any of the following three primary ways:

- **Operation merging**

  To construct the 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 the query uses different functions, rewriting can reduce the number of calculations that need to be done.

- **Operation movement**

  To construct the query with the minimum number of operations and predicates, the compiler rewrites queries 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 the extended example provided, the DISTINCT operation is removed completely.

  – Example - General Predicate Pushdown

    During query rewrite, the optimizer can change the order of applying predicates so that more selective predicates are applied to the query as early as possible.

  – Example - Decorrelation

    In a partitioned database environment, the movement of results sets between 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 query rewriting.

- **Predicate Translation**

  The SQL and XQuery compiler rewrites queries to translate existing predicates to more optimal predicates for the specific query. 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 the query to allow the optimizer to consider additional table joins when selecting the best access plan for the query.

  – Example - OR to IN Transformations

During query rewriting, 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: view merges

Suppose you have access to the following two views of the EMPLOYEE table, one showing employees with a high level of education and the other view showing employees earning more than $35,000:

```
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
```

Now suppose you perform the following query to list the employees who have a high education level and who are earning more than $35,000:

```
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 may 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 can sometimes be written or generated which have unnecessary joins. Queries such as the following could also be produced during the query rewrite stage.

```
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
```

In this query, 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
```

Another example assumes that a referential constraint exists between the EMPLOYEE and DEPARTMENT sample 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 users know that the query can be re-written, they may not be able to do so because they do not have access to the underlying tables. They may only have access to the view shown above. Therefore, this type of optimization has to be performed within the database manager.

Redundancy in referential integrity joins is likely where:

- Views are defined with joins
- Queries are automatically generated.

  For example, there are automated tools in query managers which prevent users from writing optimized queries.

Example - Shared Aggregation

Using multiple functions within a query can generate several calculations which take time. Reducing the number of calculations to be done within the query results in an improved plan. The SQL and XQuery compiler takes a query using multiple functions such as:

```
SELECT SUM(SALARY+BONUS+COMM) AS OSUM,
  AVG(SALARY+BONUS+COMM) AS OAVG,
  COUNT(*) AS OCOUNT
FROM EMPLOYEE;
```

and transforms the query in the following way:

```
SELECT OSUM,
  OSUM/OCOUNT
  OCOUNT
FROM (SELECT SUM(SALARY+BONUS+COMM) AS OSUM,
  COUNT(*) AS OCOUNT
FROM EMPLOYEE) AS SHARED_AGG;
```

This rewrite reduces the query from 2 sums and 2 counts to 1 sum and 1 count.

Compiler rewrite example: DISTINCT elimination

If the EMPNO column was defined as the primary key of the EMPLOYEE table, the following query:

```
SELECT DISTINCT EMPNO, FIRSTNME, LASTNAME
  FROM EMPLOYEE
```

would be rewritten by removing the DISTINCT clause:

```
SELECT EMPNO, FIRSTNME, LASTNAME
  FROM EMPLOYEE
```

In the above example, since the primary key is being selected, the SQL and XQuery compiler knows that each row returned will already be unique. In this case, the DISTINCT key word is redundant. If the query is not rewritten, the optimizer would need to build a plan with the necessary processing, such as a sort, to ensure that the columns are distinct.

Example - General Predicate Pushdown

Altering the level at which a predicate is normally applied can result in improved performance. For example, given the following view which 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'
```

And given the following query:

```
SELECT FIRSTNME, PHONENO
  FROM D11_EMPLOYEE
 WHERE LASTNAME = 'BROWN'
```

The query rewrite stage of the compiler will push the predicate `LASTNAME = 'BROWN'` down into the view D11_EMPLOYEE. This allows the predicate to be applied sooner and potentially more efficiently. The actual query that could be executed in this example is:

```
SELECT FIRSTNME, PHONENO
  FROM EMPLOYEE
 WHERE LASTNAME = 'BROWN'
   AND WORKDEPT = 'D11'
```

Pushdown of predicates is not limited to views. Other situations in which predicates may be pushed down include UNIONs, GROUP BYs, and derived tables (nested table expressions or common table expressions).

Example - Decorrelation

In a partitioned database environment, the SQL and XQuery compiler can rewrite the following query:

Find all the employees who are working on programming projects and 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 = A.PROJNO
          AND E1.EMPNO = P1.EMPNO)
```

Since this query is correlated, and since both PROJECT and EMPLOYEE are unlikely to be partitioned on PROJNO, the broadcast of each project to each database partition is possible. In addition, the subquery would have to be evaluated many times.

The SQL and XQuery 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 once only for each project:

  ```
  WITH DIST_PROJS(PROJNO, EMPNO) AS
  (SELECT DISTINCT PROJNO, EMPNO
   FROM PROJECT P1
   WHERE P1.PROJNAME LIKE '%PROGRAMMING%')
  ```

- Using the distinct list of employees working on the programming projects, join this to 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)
  ```

- Then the new query would be:

  ```
  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
  ```

The rewritten query computes the AVG_COMP per project (AVG_PRE_PROJ) and can then broadcast the result to all database partitions containing the EMPLOYEE table.

Compiler rewrite example: implied predicates

The following query produces a list of the managers whose departments report to "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
```

The query rewrite adds the following implied predicate:

```
DEPT.MGRNO = PROJ.RESPEMP
```

As a result of this rewrite, the optimizer can consider additional joins when it is trying to select the best access plan for the query.

In addition to the above predicate transitive closure, query rewrite also derives additional local predicates based on the transitivity implied by equality predicates. For example, the following query lists 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'
```

For this query, the rewrite stage adds the following implied predicate:

```
EMP.WORKDEPT > 'E00'
```

As a result of this rewrite, the optimizer reduces the number of rows to be joined.

Example - OR to IN Transformations

Suppose 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, converting the OR clause to the following IN predicate allows the query to be processed more efficiently:

```
SELECT *
  FROM EMPLOYEE
 WHERE DEPTNO IN ('D11', 'D21', 'E21')
```

Note: In some cases, the database manager might convert an IN predicate to a set of OR clauses so that index ORing might be performed.

Predicate typology and access plans

A user application requests a set of rows from the database with a query statement that specifies qualifiers for the specific rows to be returned as the result set. These qualifiers usually appear in the WHERE clause of the query. Such qualifiers are called *predicates*. 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 listed below, ordered in terms of performance from best to worst:

1. Range delimiting predicates
2. Index SARGable predicates
3. Data SARGable predicates
4. Residual predicates.

Note: *SARGable* refers to a term that can be used as a *s*earch *arg*ument.

The following table summarizes the predicate categories. Subsequent sections describe each category in more detail.

Table 62. 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 number of rows passed internally	Yes	Yes	Yes	No
Reduce 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 involved 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, DEPT = :hv2) are range-delimiting predicates, while YEARS > :hv5 is an index SARGable predicate.

The optimizer uses the index data when it evaluates these predicates instead of reading the base table. These *index SARGable* predicates reduce the set of rows that need to be read from the table, but they do not affect the number of index pages that are accessed.

Predicates on XML data occurring in XMLEXISTS and XMLTABLE expressions are also supported by XSCAN data operator scans. Some of these predicates are also supported by index range scans.

Data SARGable Predicates

Predicates that cannot be evaluated by the index manager, but can be evaluated by data management services are called *data SARGable* predicates. These predicates usually require accessing individual rows from a table. If required, Data Management Services 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.

For example, consider a single index defined on the PROJECT table:

```
INDEX IX0:  PROJNO ASC
```

For the following query, then, the DEPTNO = 'D11' predicate is considered to be data SARGable.

```
SELECT PROJNO, PROJNAME, RESPEMP
  FROM PROJECT
 WHERE DEPTNO = 'D11'
 ORDER BY PROJNO
```

Residual Predicates

Residual predicates require more I/O costs than accessing a table. They might have the following characteristics:

- 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.

Sometimes predicates that are applied only to the 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

Federated database pushdown analysis

For queries in federated databases, the optimizer performs pushdown analysis to find out whether an operation can be performed at a remote data source. An operation might be a function, such as relational operator, system or user function, or an SQL operator, such as GROUP BY, ORDER BY, and so on.

Note: Although the DB2 SQL compiler has much information about data source SQL support, this data may need adjustment over time because data sources can be upgraded and/or customized. In such cases, make enhancements known to DB2 by changing local catalog information. Use DB2 DDL statements (such as CREATE FUNCTION MAPPING and ALTER SERVER) 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 DB2 to retrieve the entire table from the remote data source and then filter it locally against the predicate. Network constraints and large table size 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 DB2 to retrieve the entire table from the remote data source.

For example, assume that a nickname N1 references the data source table EMPLOYEE in a DB2 for OS/390® or z/OS data source. Also assume that the table has 10,000 rows, that one of the columns contains the last names of employees, and that one of the columns contains salaries. Consider the following statement:

```
SELECT LASTNAME, COUNT(*)  FROM N1
   WHERE LASTNAME  > 'B' AND SALARY > 50000
   GROUP BY LASTNAME;
```

Several possibilities are considered, depending on whether the collating sequences at DB2 and DB2 for OS/390 or z/OS are the same:

- If the collating sequences are the same, the query predicate can probably be pushed down to DB2 for OS/390 or z/OS. Filtering and grouping results at the data source is usually more efficient than copying the entire table to DB2 and performing the operations locally. For the query above, the predicate and the GROUP BY operation can take place at the data source.
- If the collating sequence is not the same, the entire predicate cannot be evaluated at the data source. However, the optimizer might decide to pushdown the `SALARY > 50000` portion of the predicate. The range comparison must still be done at DB2.
- If the collating sequence is 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 at DB2 is the best (least cost) 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 on data sources. Many factors affect whether a function or an SQL operator is evaluated at a remote data source. Factors to be evaluated are classified in the following three groups:

- 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 supported by DB2. This dialect might offer more functionality than the SQL dialect supported by a server accessed by a query. DB2 can compensate for the lack of function at a data server, but doing so may require that the operation take place at DB2.

SQL Capabilities: Each data source supports a variation of the SQL dialect and different levels of functionality. For example, consider the GROUP BY list. 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, DB2 might have to perform the 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 DB2 cannot determine a good method to bind a value for a function, this function must be evaluated locally.

SQL Limits: Although DB2 might allow the use of larger integers than its remote data sources, values that exceed remote limits cannot be embedded in statements sent to data sources. Therefore, the function or operator that operates on this constant must be evaluated locally.

Server Specifics: Several factors fall into this category. One example is whether NULL values are sorted as the highest or lowest value, or depend on the ordering. If NULL values are sorted at a data source differently from DB2, ORDER BY operations on a nullable expression cannot be remotely evaluated.

Collating Sequence: Retrieving data for local sorts and comparisons usually decreases performance. Therefore, consider configuring the federated database to use the same collating sequences that your data sources use. 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 if improved performance results.

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. Comparison statements might return unexpected results if you submit statements to a case-insensitive data source. The weights assigned to the characters ″I″ and ″i″ in a case-insensitive data source are the same. DB2, by default, is case sensitive and assigns different weights to the characters.

To improve performance, the federated server allows sorts and comparisons to take place at data sources. For example, in DB2 for OS/390 or z/OS, sorts defined by ORDER BY clauses are implemented by a collating sequence based on an EBCDIC code page. To use the federated server to retrieve DB2 for OS/390 or z/OS data 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, DB2 retrieves the data to the federated database. Because users expect to see the query results ordered by the collating sequence defined for the federated server, by ordering the data locally the federated server ensures that this expectation is fulfilled. Submit your query in pass-through 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. In particular, review your settings for *collating_sequence*, *varchar_no_trailing_blanks*, and *pushdown*.

DB2 Type Mapping and Function Mapping Factors: The default local data type mappings provided by DB2 are designed to provide sufficient buffer space for each data source data type, which avoids loss of data. Users 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, DB2 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. An example of this situation is the IS NULL relational operator. Most data sources support it, but some may have restrictions, such as only allowing a column name 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 than if it is evaluated locally by DB2.

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 the 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 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 equal 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 (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 SQL statement ALTER NICKNAME 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 performed on columns so indicated. Based on this indication, DB2 may generate a different but equivalent form of a predicate to be used in the remote SQL statement sent to a data source. A user 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.

The table below describes these options.

Table 63. Column Options and Their Settings

Option	Valid Settings	Default Setting
numeric_string	**'Y'** Yes, this column contains only strings of numeric data. IMPORTANT: If the column contains only numeric strings followed by trailing blanks, do not specify 'Y'. **'N'** No, this column is not limited to strings of numeric data. If you set numeric_string to 'Y' for a column, you are informing the optimizer that this column contains no blanks that could interfere with sorting of the column data. This option is useful when the collating sequence of a data source is different from DB2. Columns marked with this option are not excluded from local (data source) evaluation because of a different collating sequence.	'N'
varchar_no_trailing_blanks	Specifies whether this data source uses non-blank padded VARCHAR comparison semantics. For variable-length character strings that contain no trailing blanks, non-blank-padded comparison semantics of some DBMSs return the same results as DB2 comparison semantics. If you are certain that all VARCHAR table/view columns at a data source contain no trailing blanks, consider setting this server option to 'Y' for a data source. This option is often used with Oracle data sources. Ensure that you consider all objects that might have nicknames, including views. **'Y'** This data source has non-blank-padded comparison semantics similar to DB2. **'N'** This data source does not have the same non-blank-padded comparison semantics as DB2.	'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 at DB2 to combine the results from two referenced data sources that use one operator, such as a set operator (e.g. UNION). The operator cannot be evaluated at a remote data source directly.

Guidelines for analyzing where a federated query is evaluated

DB2 provides two utilities to show where queries are evaluated:

- Visual explain. Start it with the **db2cc** command. Use it to view the query access plan graph. The execution location for each operator is included in the detailed display of an operator.

  If a query is pushed down, you should see a RETURN operator. The RETURN operator is a standard DB2 operator. For a SELECT statement that selects data from a nickname, you also see a SHIP operator. The SHIP operator 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 supported by the data source. It can contain any valid query for that data source.

  If an INSERT, DELETE, or UPDATE query can be entirely pushed down to the remote database, you might not see a SHIP statement in the access plan. All remotely executed INSERT, UPDATE, and DELETE statements are shown for the RETURN operator. However, if a query cannot be entirely pushed down, the SHIP operator shows which operations were performed remotely.
- SQL explain. Start it with the **db2expln** or the **dynexpln** command. Use it to view the access plan strategy as text.

Understanding why a query is evaluated at a data source or at DB2

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 predicate is very selective and thus 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 involving 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. Does this predicate, if remotely evaluated, require bind-in of some value? If so, would it violate SQL restrictions at this data source?
  – Global optimization. The optimizer may have decided that local processing is more cost effective.

- Why isn't the GROUP BY operator evaluated remotely?

  There are several areas you can check:

  – 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:
    - Limited number of GROUP BY items
    - Limited byte counts of combined GROUP BY items
    - Column specification only on the GROUP BY list
  – Does the data source support this SQL operator?
  – Global optimization. The optimizer may have decided that local processing is more cost effective.
  – Does the GROUP BY operator clause contain a character expression? If it does, verify that the remote data source has the same case sensitivity as DB2.

- Why isn't the set operator evaluated remotely?

  There are several areas you can check:

  – Are both of its operands completely evaluated 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 or long fields valid input for this specific set operator?

- Why isn't the ORDER BY operation evaluated remotely?

  Consider:

  – 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, does the remote data source not have the same collating sequence or case sensitivity as DB2?

– Does the data source have any restrictions on this operator? For example, is there a limited number of ORDER BY items? Does the 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. This generation of remote SQL helps produce a globally optimal access strategy for a query.

The optimizer uses the output of pushdown analysis to decide whether each operation is evaluated locally at DB2 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 transmitting the data or messages between DB2 and data sources.

Although the goal is to produce an optimized query, the following major factors affect the output from global optimization and thus affect query performance.

- Server characteristics
- Nickname characteristics

Server characteristics and options that affect global optimization

The following data source server factors can affect global optimization:

- Relative ratio of CPU speed

  Use the *cpu_ratio* server option to specify how fast or slow the data-source CPU speed is compared with the DB2 CPU. A low ratio indicates that the data-source computer CPU is faster than the DB2 computer CPU. If the ratio is low, the DB2 optimizer is more likely to consider pushing down CPU-intensive operations to the data source.

- Relative ratio of I/O speed

  Use the *io_ratio* server option to indicate how much faster or slower the data source system I/O speed is compared with the DB2 system. A low ratio indicates that the data source workstation I/O speed is faster than the DB2 workstation I/O speed. If the ratio is low, the DB2 optimizer considers pushing down I/O-intensive operations to the data source.

- Communication rate between DB2 and the data source

  Use the *comm_rate* server option to indicate network capacity. Low rates, which indicate a slow network communication between DB2 and the data source, encourage the DB2 optimizer to reduce the number of messages 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 optimizer considers the data retrieved from this data source as 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, DB2 does not send any plan hints to the data source.

Plan hints are statement fragments that provide extra information for data-source optimizers. For some queries this information can improve performance. The plan hints can help the data source optimizer decide whether to use an index, which index to use, or which table join sequence to use.

If plan hints are enabled, the query sent to the data source contains additional information. For example, a statement sent to an Oracle optimizer with plan hints 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

  DB2 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 DBMSs. In other words, DB2 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, DB2 can use information about indexes at data sources. For this reason, it is important that the index information available to DB2 is current. The index information for nicknames is initially acquired when the nickname is created. Index information is not gathered 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:

- DB2 cannot retrieve any index information from a data source during nickname creation.
- You want an index for a view nickname.
- You want to encourage the DB2 optimizer to use a specific nickname as the inner table of a nested loop join. The user 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 views that are not simple select statements, such as a view created by joining two tables, query performance might suffer. For example, you create an index over a view that is a join of two tables, the optimizer might choose that view as the inner element in a nested-loop join. The query will have poor performance because the join is evaluated several times. An alternative is to create nicknames for each of the tables referenced in the data source view and create a local view at DB2 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 DB2 can retrieve the statistical data stored at a data source, it cannot automatically detect updates to existing statistical data at data sources. Furthermore, DB2 cannot handle changes in object definition or structural changes, such as adding a column, to objects at data sources. If the statistical data or structural data for an object has changed, you have two choices:

- Run the equivalent of RUNSTATS at the data source. Then drop the current nickname and re-create it. Use this approach if structural information has changed.
- Manually update the statistics in the SYSSTAT.TABLES view. This approach requires fewer steps but it does not work if structural information has changed.

Global analysis of federated database queries

The following two utilities provided show global access plans:

- Visual Explain. Start it from the Control Center, or execute the *db2cc* command, which starts the Control Center. Use Visual Explain to view the query access plan graph. The execution location for each operator is included in the detailed display of an operator. You can also find the remote SQL statement generated for each data source in the SHIP or RETURN operator, depending on the type of the query. By examining the details of each operator, you can see the number of rows estimated by the DB2 optimizer as input to and output from each operator. You can also see the estimated cost to execute each operator including the communications cost.
- SQL explain. Start it with the *db2expln* or *dynexpln* command. Use SQL explain to view the access plan strategy as text. Although SQL explain does not provide cost information, you can get the access plan generated by the remote optimizer for those data sources supported by the remote explain function.

Understanding DB2 optimization decisions

Consider the following optimization questions and key areas to investigate for performance improvements:

- Why isn't a join between two nicknames of the same data source being evaluated remotely?

  Areas to examine include:

  – Join operations. Can the 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.

  – Number of rows in the join result (with Visual Explain). Does the join produce a much larger set of rows than the two nicknames combined? Do the numbers make sense? If the answer is no, consider updating the nickname statistics manually (SYSSTAT.TABLES).
- Why isn't the GROUP BY operator being evaluated remotely?

  Areas to examine include:

  – Operator syntax. Verify that the operator can be evaluated at the remote data source.

  – Number of rows. Check the estimated number of rows in the GROUP BY operator input and output using visual explain. Are these two numbers very

close? If the answer is yes, the DB2 optimizer considers it more efficient to evaluate this GROUP BY locally. Also, do these two numbers make sense? If the answer is no, consider updating the nickname statistics manually (SYSSTAT.TABLES).

- 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 still relies on its cost estimate to generate a globally optimal 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 CPU speed is much slower than the CPU speed for DB2 and thus it may turn out to be more beneficial to perform the operations at DB2 instead. If results are not satisfactory, verify your server statistics in SYSCAT.SERVEROPTIONS.

- Why does a plan generated by the optimizer, and completely evaluated at a remote data source, have much worse performance than the original query executed directly at the remote data source?

  Areas to examine include:

  – The remote SQL statement generated by the DB2 optimizer. Ensure that it is identical to the original query. Check for predicate ordering changes. A good query optimizer should not be sensitive to the predicate ordering of a query; unfortunately, not all DBMS optimizers are identical, and thus it is likely that the optimizer of the remote data source may generate a different plan based on the input predicate ordering. If this is true, this is a problem inherent in the remote optimizer. Consider either modifying the predicate ordering on the input to DB2 or contacting the service organization of the remote data source for assistance.

    Also, check for predicate replacements. A good query optimizer should not be sensitive to equivalent predicate replacements; unfortunately, not all DBMS optimizers are identical, and thus it is possible that the optimizer of the remote data source may generate a different plan based on the input predicate. For example, some optimizers cannot generate transitive closure statements for predicates.

  – The number of returned rows. You can get this number from Visual Explain. If the query returns a large number of rows, network traffic is a potential bottleneck.

  – Additional functions. Does the remote SQL statement contain additional functions compared with the original query? Some of the extra functions may be generated to convert data types. Ensure 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 its estimates, the optimizer selects an optimal access plan. An *access plan* specifies the order of operations 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 the time that the application is executed.

There are two ways to access data in a table:

- Scanning the entire table sequentially

- Locating specific table rows by first accessing an index on the table

To produce the results that the query requests, rows are selected depending on the terms of the predicate, which are usually stated in a WHERE clause. The selected rows in accessed tables are joined to produce the result set, and the result set might be further processed by grouping or sorting the output.

Data access through index scans

An *index scan* occurs when the database manager accesses an index for any of the following reasons:

- To narrow the set of qualifying rows (by scanning the rows in a certain range of the index) before accessing the base table. 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.
- To order the output.
- To retrieve the requested column data directly. If all of the requested data is in the index, the indexed table does not need to be accessed. This is known as an *index-only access*.

If indexes are created with the ALLOW REVERSE SCANS option, scans may also be performed in the direction opposite to that with which they were defined.

Note: The optimizer chooses a table scan if no appropriate index has been created or if an index scan would be more costly. An index scan might be more costly when the table is small the index-clustering ratio is low, or the query requires most of the table rows. To find out whether the access plan uses a table scan or an index scan, use the Explain facility.

Index Scans to Delimit 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 that can be used to delimit the range of an index scan in the following cases:

- Tests for equality against a constant, a host variable, an expression that evaluates to a constant, or a keyword
- Tests for “IS NULL” or “IS NOT NULL”
- Tests for equality against a basic subquery, which is a subquery that does not contain ANY, ALL, or SOME, and the subquery does not have a correlated column reference to its immediate parent query block (that is, the SELECT for which this subquery is a subselect).
- Tests for strict and inclusive inequality.

The following examples illustrate when an index might be used to limit a range:

- Consider an index with the following definition:

```
INDEX IX1:  NAME    ASC,
            DEPT    ASC,
            MGR     DESC,
            SALARY  DESC,
            YEARS   ASC
```

  In this case, the following predicates might be used to limit the range of the scan of index IX1:

```
WHERE  NAME = :hv1
  AND  DEPT = :hv2
```

or

```
WHERE  MGR  = :hv1
  AND  NAME = :hv2
  AND  DEPT = :hv3
```

Note that in the second WHERE clause, the predicates do not have to be specified in the same order as the key columns appear in the index. Although the examples use host variables, other variables such as parameter markers, expressions, or constants would have the same effect.

- Consider a single index created using the ALLOW REVERSE SCANS parameter. Such indexes support scans in the direction defined when the index was created as well as in the opposite or reverse direction. The statement might look something like this:

  ```
  CREATE INDEX iname ON tname (cname DESC) ALLOW REVERSE SCANS
  ```

  In this case, the index (`iname`) is formed based on DESCending values in `cname`. By allowing reverse scans, although the index on the column is defined for scans in descending order, a scan can be done in ascending order. The actual use of the index in both directions is not controlled by you but by the optimizer when creating and considering access plans.

In the following WHERE clause, only the predicates for NAME and DEPT would be used in delimiting the range of the index scan, but not the predicates for SALARY or YEARS:

```
WHERE  NAME   = :hv1
  AND  DEPT   = :hv2
  AND  SALARY = :hv4
  AND  YEARS  = :hv5
```

This is because there is a key column (MGR) separating these columns from the first two index key columns, so the ordering would be off. However, once the range is determined by the NAME = :hv1 and DEPT = :hv2 predicates, the remaining predicates can be evaluated against the remaining index key columns.

Index Scans to Test Inequality

Certain inequality predicates can delimit the range of an index scan. There are two types of inequality predicates:

- Strict inequality predicates

  The strict inequality operators used for range delimiting predicates are greater than (>) and less than (<).

  Only one column with strict inequality predicates is considered for delimiting a range for an index scan. In the following example, the predicates on the NAME and DEPT columns can be used to delimit the range, but the predicate on the MGR column cannot be used.

  ```
  WHERE  NAME   = :hv1
    AND  DEPT   > :hv2
    AND  DEPT   < :hv3
    AND  MGR    < :hv4
  ```

- Inclusive inequality predicates

  The following are inclusive inequality operators that can be used for range delimiting predicates:

  – >= and <=

– BETWEEN
– LIKE

For delimiting a range for an index scan, multiple columns with inclusive inequality predicates will be considered. In the following example, all of the predicates can be used to delimit the range of the index scan:

```
WHERE  NAME   = :hv1
  AND  DEPT  >= :hv2
  AND  DEPT  <= :hv3
  AND  MGR   <= :hv4
```

To further illustrate this example, suppose that `:hv2 = 404`, `:hv3 = 406`, and `:hv4 = 12345`. The database manager will scan the index for all of departments 404 and 405, but it will stop scanning department 406 when it reaches the first manager that has an employee number (MGR column) greater than 12345.

Index Scans to Order Data

If the query requires output in sorted order, an index might 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, "= ANY" subquery, "> ALL" subquery, "< ALL" subquery, INTERSECT or EXCEPT, UNION. An exception to this is when the index key columns are compared for equality against "constant values", which is any expression that evaluates to a constant. In this case the ordering column can be other than the first index key columns.

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 thus 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. This uniqueness ensures that there is only one PROJNAME value for each PROJNO value.

Types of index access

In some cases, the optimizer might find that all data that a query requires from a table 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 via 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 the index without accessing the table. This is known as an *index-only* access.

To illustrate an index-only access, 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, and without reading the base table:

```
SELECT NAME, DEPT, MGR, SALARY
  FROM EMPLOYEE
 WHERE NAME = 'SMITH'
```

Often, however, required columns that do not appear in the index. To obtain the data for these columns, the table rows must be read. To allow the optimizer to choose an 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 uniqueness of the NAME column and also stores and maintains data for DEPT, MGR, SALARY, and YEARS columns, which allows the following query to be satisfied by accessing only the index:

```
SELECT NAME, DEPT, MGR, SALARY
   FROM EMPLOYEE
   WHERE NAME='SMITH'
```

When you consider adding INCLUDE columns to an index, however, consider whether the additional storage space and maintenance costs are justified. If queries that can be satisfied by reading only such an index 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 the two indexes:

```
WHERE DEPT = :hv1
  OR (JOB   = :hv2
  AND YEARS >= :hv3)
```

Scanning index IX2 produces a list of row IDs (RIDs) that satisfy the `DEPT = :hv1` predicate. Scanning index IX3 produces a list of RIDs satisfying the `JOB = :hv2 AND YEARS >= :hv3` predicate. These two lists of RIDs are combined and duplicates removed before the table is accessed. This is known as *index ORing*.

Index ORing may also be used for predicates specified in the 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 with applications that create multiple indexes on corresponding columns in the same table and a query using multiple AND predicates is run against that table. Multiple index scans against each indexed column in such a query produce values which are hashed to create bitmaps. The second bitmap is used to probe the first bitmap to generate the qualifying rows that are fetched to create the final returned data set.

For example, given the following two index definitions:

```
INDEX IX4: SALARY   ASC
INDEX IX5: COMM     ASC
```

the following predicates could be resolved using these two indexes:

```
WHERE SALARY BETWEEN 20000 AND 30000
   AND COMM BETWEEN 1000 AND 3000
```

In this example, scanning index `IX4` produces a bitmap satisfying the `SALARY BETWEEN 20000 AND 30000` predicate. Scanning `IX5` and probing the bitmap for `IX4` results in the list of qualifying RIDs that satisfy both predicates. This is known as "dynamic bitmap ANDing". It occurs only if the table has sufficient cardinality and the columns have sufficient values in the qualifying range, or sufficient duplication if equality predicates are used.

To realize the performance benefits of dynamic bitmaps when scanning multiple indexes, it may be necessary to change the value of the sort heap size (*sortheap*) database configuration parameter, and the sort heap threshold (*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*.

Note: The optimizer does not combine index ANDing and index ORing in accessing a single table.

Index Access in Range clustered tables

Unlike standard tables, a range clustered table does not require a physical index that maps a key value to a row like a traditional B-tree index. Instead, it leverages the sequential nature of the column domain and uses a functional mapping to generate the location of a given row in a table. In the simplest example of this mapping, the first key value in the range is the first row in the table, and 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 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 ordering.

Index access and cluster ratios

When it chooses an access plan, the optimizer estimates the number of I/Os required to fetch required pages from disk to the buffer pool. This estimate includes a prediction of buffer-pool usage, since additional I/Os are not required to read rows in a page that is already in the buffer pool.

For index scans, information from the system catalog tables (SYSCAT.INDEXES) helps the optimizer estimate I/O cost of reading data pages into the buffer pool. It uses information from the following columns in the SYSCAT.INDEXES table:

- CLUSTERRATIO information indicates the degree to which the table data is clustered in relation to this index. The higher the number, the better 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.
- PAGE_FETCH_PAIRS 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 execute RUNSTATS on the index with the DETAILED clause.

If index clustering statistics are not available, the optimizer uses default values, which assume poor clustering of the data to the index.

The degree to which the data is clustered with respect to the index can have a significant impact on performance and you should try to keep one of the indexes on the table close to 100 percent clustered.

In general, only one index can be one hundred percent clustered, except in those cases where the keys are a superset of the keys of the clustering index or where there is de facto correlation between the key columns of the two indexes.

When you reorganize an table, you can specify an index that will be used to cluster the rows and attempt to preserve this characteristic during insert processing. Because updates and inserts may make the table less well clustered in relation to the index, you might need to periodically reorganize the table. To reduce the frequency of reorganization on a table that has frequent changes due to INSERTs, UPDATEs, and DELETES, use the PCTFREE parameter when you alter a table. This allows for additional inserts to be clustered with the existing data.

Joins

A *join* is the process of combining information 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 joining criterion.

For example, consider the following two tables:

Table1		Table2	
PROJ	**PROJ_ID**	**PROJ_ID**	**NAME**
A	1	1	Sam
B	2	3	Joe
C	3	4	Mary
D	4	1	Sue
		2	Mike

To join Table1 and Table2 where the 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
```

This query yields the following set of result rows:

PROJ	PROJ_ID	NAME
A	1	Sam
A	1	Sue
B	2	Mike
C	3	Joe
D	4	Mary

Depending on the existence of a join predicate, as well as various costs involved as determined by 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 as the inner. 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 present. 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. 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
- Hash join

These methods are described in the following sections.

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, consider that column A in tables T1 and T2 have the following values:

Outer table T1: column A	Inner table T2: column A
2	3
3	2
3	2
	3
	1

To perform a nested-loop join, the database manager performs the following steps:

1. Read the first row from T1. The value for A is "2"
2. Scan T2 until a match ("2") is found, and then join the two rows
3. Scan T2 until the next match ("2") is found, and then join the two rows
4. Scan T2 to the end of the table
5. Go back to T1 and read the next row ("3")
6. Scan T2, starting at the first row, until a match ("3") is found, and then join the two rows
7. Scan T2 until the next match ("3") is found, and then join the two rows
8. Scan T2 to the end of the table
9. Go back to T1 and read the next row ("3")
10. Scan T2 as before, joining all rows which match ("3").

- Performing an index lookup on the inner table for each accessed row of the outer table

  This method can be used for the specified predicates if there is a predicate of the following 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. Consider the following examples:

  ```
  OUTER.C1 + OUTER.C2 <= INNER.C1
  OUTER.C4 < INNER.C3
  ```

  This method might significantly reduce the number of rows accessed in the inner table for each access of the outer table, although it 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 orders the outer table, based on the join columns, the number of read operations to access pages from disk for the inner table might be reduced, because they are more likely to be be in the buffer pool already. If the join uses a highly clustered index to access the inner table and if the outer table has been sorted, the number of index pages accessed might be minimized.

In addition, 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.

Merge Join

Merge join, sometimes known as *merge scan join* or *sort merge join*, requires a predicate of the form table1.column = table2.column. This is called an *equality join predicate*. 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 there are repeated values occur, a group of rows in the inner table might be scanned again. For example, if column A in tables T1 and T2 has the following values:

Outer table T1: column A	Inner table T2: column A
2	1
3	2
3	2
	3
	3

To perform a merge join, the database manager performs the following steps:

1. Read the first row from T1. The value for A is "2".
2. Scan T2 until a match 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. The end of T2 is reached.
8. 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, for which the column types are the same. For columns of type CHAR, the length must be the same. For columns of type DECIMAL, the precision and scale must be the same. For columns of type DECFLOAT, the precision must be the same. The column type cannot be a LONG field column, or a large object (LOB) column.

First, the designated INNER table is scanned and the rows copied into memory buffers 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 to rows from the INNER table by first comparing the hash value computed for the columns of the join predicates. If the hash value for the OUTER row column matches the hash value of the INNER row column, the actual join predicate column values are compared.

OUTER table rows that correspond to portions of the table 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, and the hash values of their rows are matched, and the join predicates are checked.

For the full performance benefits of hash join, you might need to change the value of the *sortheap* database configuration parameter and the *sheapthres* database manager configuration parameter.

For the full performance benefits 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 available for *sheapthres*, 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 specified by the *sheapthres* parameter.

Increasing the *sortheap* value should also improve 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 are the following search strategies, which are determined by the optimization class of the query:

- Greedy join enumeration
  - Efficient with respect to space and time
  - 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 chosen by the greedy join enumeration is the same as the access plan chosen by dynamic programming join enumeration. This is particularly true if the query has many join predicates on the same column, either explicitly specified, or implicitly generated through predicate transitive closure.
- Dynamic programming join enumeration
  - Space and time requirements increase exponentially as the number of joined tables increases
  - Efficient and exhaustive search for best access plan
  - Similar to the strategy used by DB2 for OS/390 or z/OS.

The join-enumeration algorithm is an important determinant of the number of plan combinations that the optimizer explores.

Star-Schema Joins

The tables 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, creating a result table consisting of the cross product of the size of the two tables that is usually very large. Since such a plan is unlikely to perform well, the optimizer avoids even determining the cost of such an 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 the other tables are called dimension tables. The dimension tables all 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 beneficial when multiple join predicates match a multi-column index.

DB2 can recognize queries against databases designed with star schemas that have at least two dimension tables and can increase the search space to include possible plans that compute the Cartesian product of dimension tables. If the plan that computes the Cartesian products has the lowest estimated cost, it is selected by the optimizer.

The star schema join strategy discussed above 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 a relatively high selectivity across all dimension tables, the following star 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 row ID (RID) values to dynamically create a bitmap.
2. Use AND predicates against the previous bitmap for each 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 multi-column indexes. Explicit referential-integrity constraints between the fact table and the dimension tables are not required, although the relationship between the fact table and the dimension tables should actually be related in this way.

The dynamic bitmaps created and used by star join techniques require sort heap memory, the size of which is specified by the Sort Heap Size (*sortheap*) database configuration parameter.

Early Out Joins

The optimizer may select an early out join when it detects that each row from one of the table only needs to be joined with at most one row from the other table.

An early out join is possible when we have a join predicate on the key column(s) of one of the tables. For example, consider the following query that returns the names of employees and their direct 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 the ID column is a key in the EMPLOYEE table and that every employee has at most one manager, then the above join can avoid searching for a subsequent matching row in the MANAGER table.

An early out join is also possible when there is a DISTINCT in the query. For example, consider the following query that returns the names of car makes having a model that sells 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 make, we only need to determine whether any one of its manufactured models sell for more than $30000. Joining a car make with all of its manufactured models selling for more than $30000 is unnecessary towards the correctness of the query result.

An early out join is also possible when the join feeds a GROUP BY with a MIN or a 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
```

Let us define the 'qualifying set' as 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 descending on DATE, then we only need to return the 1st row from the qualifying set for each symbol as that 1st row is the most recent date for that particular symbol. The other rows in the qualifying set are not required for the correctness of the query.

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 another 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 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 to T4 (T3xT4) and finally 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. Depending on the query optimization class, the optimizer places different constraints on the maximum number of tables that may 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 improve performance of frequently executed joins in a partitioned database environment by allowing the database to manage precomputed values of the table data.

Consider an example of a query and a replicated materialized table. The following assumptions are made:

- The SALES table is in the multipartition table space REGIONTABLESPACE, and is split on the REGION column.
- The EMPLOYEE and DEPARTMENT tables are in a single-partition database partition group.

Create a replicated materialized query table based on the 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;
```

To update the content of the replicated materialized query table, run the following statement:

```
REFRESH TABLE R_EMPLOYEE;
```

Note: After using the REFRESH statement, you should run RUNSTATS on the replicated table as you would any other table.

The following example 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 is on only one database partition, the database manager uses the R_EMPLOYEE table, which is replicated on each of the database partitions where the SALES tables is stored. The performance enhancement occurs because the employee information does not have to be moved across the network to each database partition to calculate the join.

Replicated materialized query tables in collocated joins

Replicated materialized query tables can also assist in the collocation of joins. For example, if a star schema contains a large fact table spread across twenty nodes, 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 on the fact table do not correspond to the distribution key of the fact table.

Consider a table called FACT (C1, C2, C3, ...) split on C1; and a table called DIM1 (C1, dim1a, dim1b, ...) split on C1; and a table called DIM2 (C2, dim2a, dim2b, ...) split on C2; and so on.

In this case, you see that 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 between DIM2 with the predicate WHERE DIM2.C2 = FACT.C2 cannot be collocated because FACT is split on column C1 and not on column C2. In this case, you might replicate DIM2 in the database partition group of the fact table so that the join occurs locally on each database partition.

Note: The replicated materialized query tables discussion here is related to intra-database replication. Inter-database replication is concerned with subscriptions, control tables, and data located in different databases and on different operating systems.

When you create a replicated materialized query table, the source table can be a single-node table or a multi-node table in a database partition group. In most cases, the replicated table is small and can be placed in a single-node database partition group. You can limit the data to be replicated by specifying only a subset of the columns from the table or by specifying the number of rows through the predicates used, or by using both methods. The data capture option is not required for replicated materialized query tables to function.

A replicated materialized query table can also be created in a multi-node 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 that 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 constraint violations that are not present on the source tables, you cannot create unique indexes or put constraints on the replicated tables. Constraints are disallowed even if the same constraint occurs on the source table.

Replicated tables can be referenced directly in a query, but you cannot use the NODENUMBER() predicate with a replicated table to see the table data on a particular partition.

Use the EXPLAIN facility to see if a replicated materialized query table was used by the access plan for a query. Whether the access plan chosen by the optimizer uses the replicated materialized query table depends on the information that needs to be joined. The optimizer might not use the replicated materialized query table 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 in partitioned databases

In some ways, join strategies are different in a partitioned database environment than in a non-partitioned database environment. Additional techniques can be applied to standard join methods to improve performance.

One consideration for those tables involved in frequent joins in a partitioned database environment is that of table collocation. Table collocation provides the means in a partitioned database environment to locate data from one table with the data from another table at the same database partition based on the same distribution key. Once collocated, data to be joined can participate in a query without having to be moved to another database partition as part of the query activity. Only the answer set for the join is moved to the coordinator node.

Table Queues

The descriptions of join techniques in a partitioned database environment use the following terminology:

- table queue

  A mechanism for transferring rows between database partitions, or between processors in a single partition database.
- directed table queue

  A table queue in which rows are hashed to one of the receiving database partitions.
- broadcast table queue

  A table queue in which rows are sent to all of the receiving database partitions, but are not hashed.

A table queue is used in the following circumstances:

- To pass table data from one database partition to another when using inter-partition parallelism
- To pass table data within a database partition when using intra-partition parallelism
- To pass table data within a database partition when using a single partition database.

Each table queue is 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, the 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 being issued by the application. When the FETCH 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 that is issued by the 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 the synchronous table queues.
- *Merging table queues.*

  These table queues preserve order.

- *Non-merging table queues.*

  These table queues are also known as "regular" table queues. They do not preserve order.
- *Listener table queues.*

  These table queues are use 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 in partitioned database environments

The following figures illustrate join methods in a partitioned database environment.

Note: In the diagrams q1, q2, and q3 refer to table queues in the examples. The tables that are referenced are divided across two database partitions for the purpose of these scenarios. The arrows indicate the direction in which the table queues are sent. The coordinator node is database partition 0.

Collocated Joins

A collocated join occurs locally on the database partition where 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 key must participate in the equality join predicates.

The following figure provides an example.

Note: Replicated materialized query tables enhance the likelihood of collocated joins.

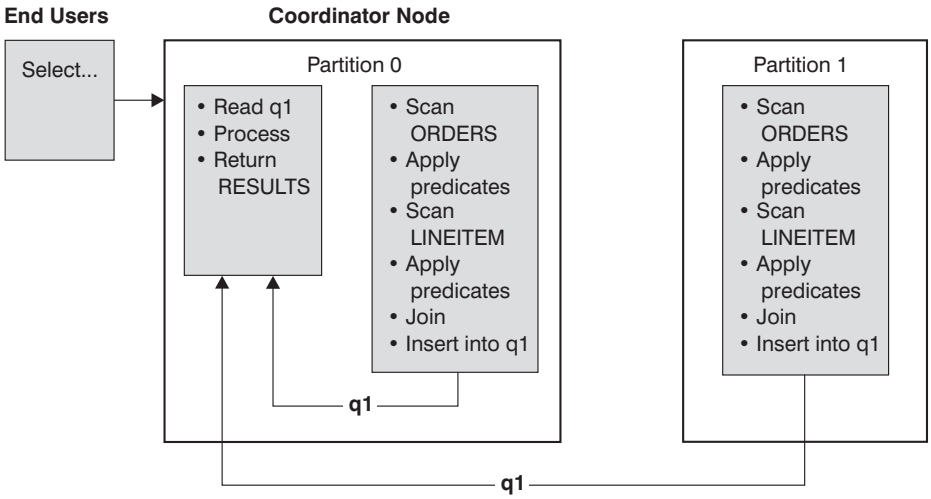
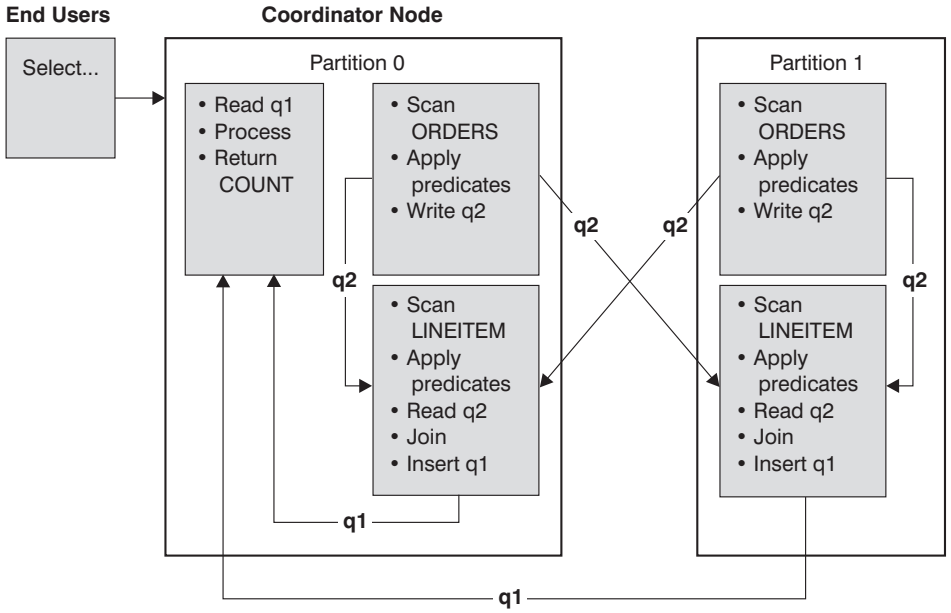


Figure 23. Collocated Join Example

Broadcast Outer-Table Joins

Broadcast outer-table joins are 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 is 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. The following figures provide an example.

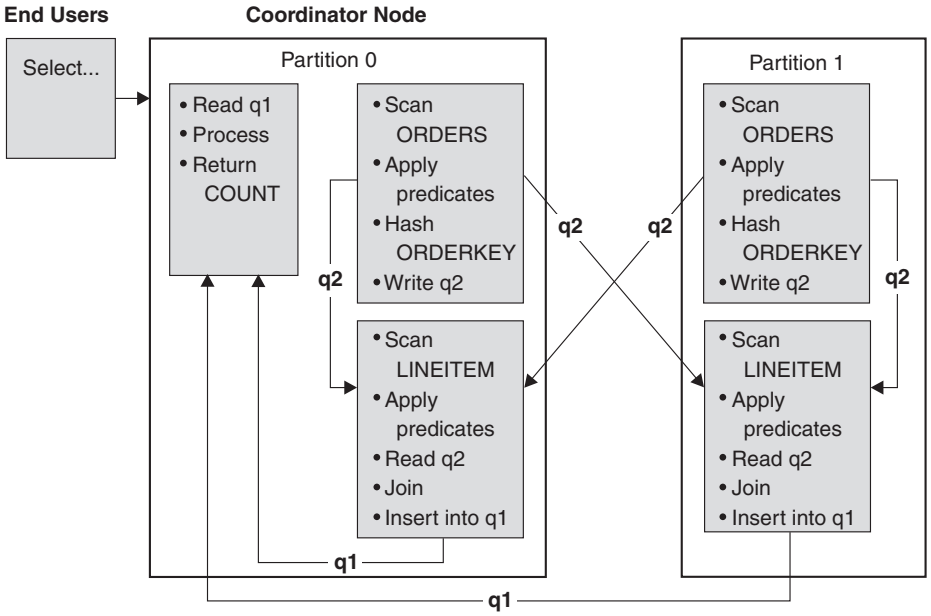


The ORDERS table is sent to all database partitions that have the LINEITEM table. Table queue q2 is broadcast to all database partitions of the inner table.

Figure 24. Broadcast Outer-Table Join Example

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. The following figure provides an example.

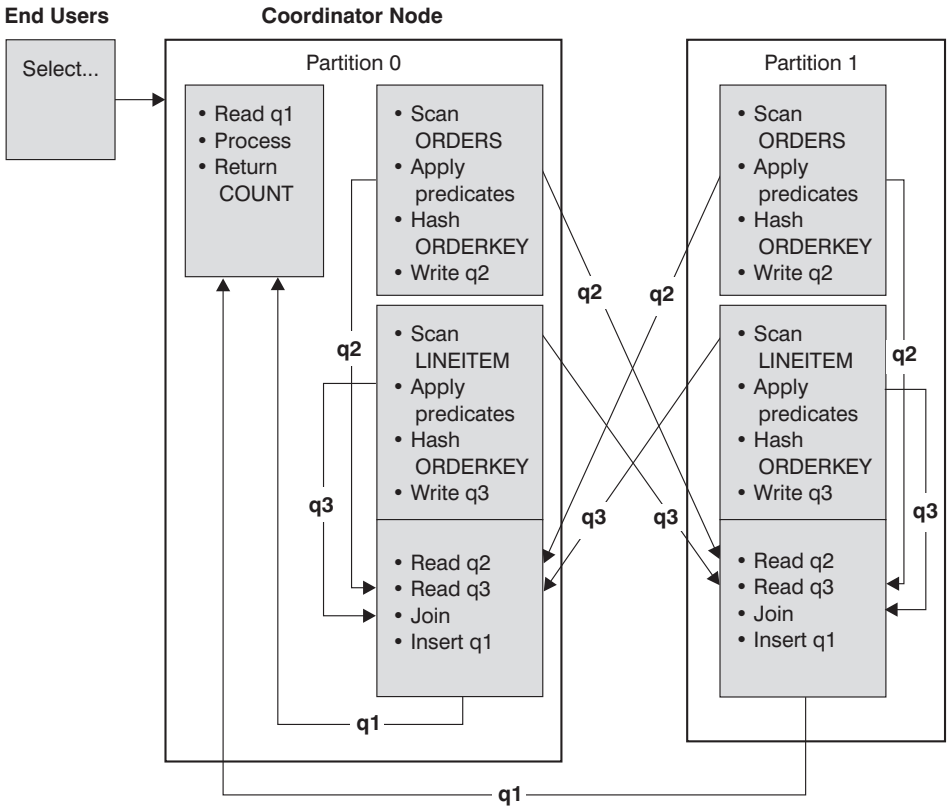


The LINEITEM table is partitioned on the ORDERKEY column.
The ORDERS table is partitioned on a different column.
The ORDERS table is hashed and sent to the correct LINEITEM table database partition.
In this example, the join predicate is assumed to be:
ORDERS.ORDERKEY = LINEITEM.ORDERKEY.

Figure 25. 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. The following figure provides an example. An example is shown in the following figure.

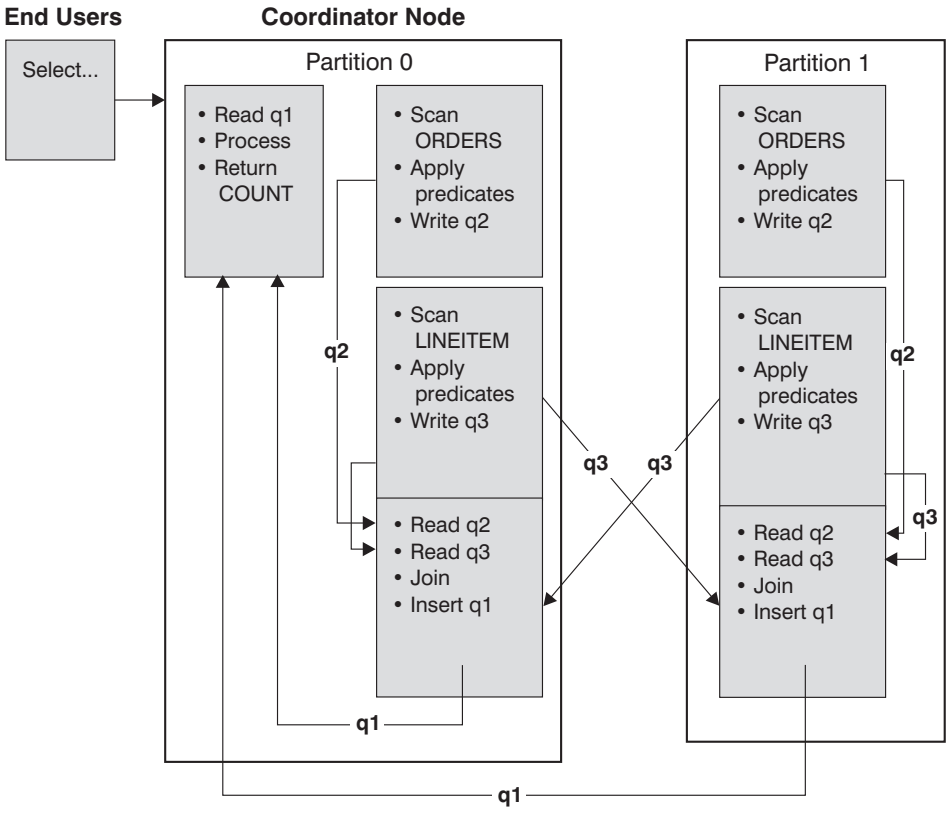


Neither table is partitioned on the ORDERKEY column.
Both tables are hashed and are sent to new database partitions where they are joined.
Both table queue q2 and q3 are directed.
In this example, the join predicate is assumed to be:
ORDERS.ORDERKEY = LINEITEM.ORDERKEY

Figure 26. Directed Inner-Table and Outer-Table Join Example

Broadcast Inner-Table Joins

In the broadcast inner-table join strategy, the inner table is broadcast to all the database partitions of the outer join table. The following figure provides an example.

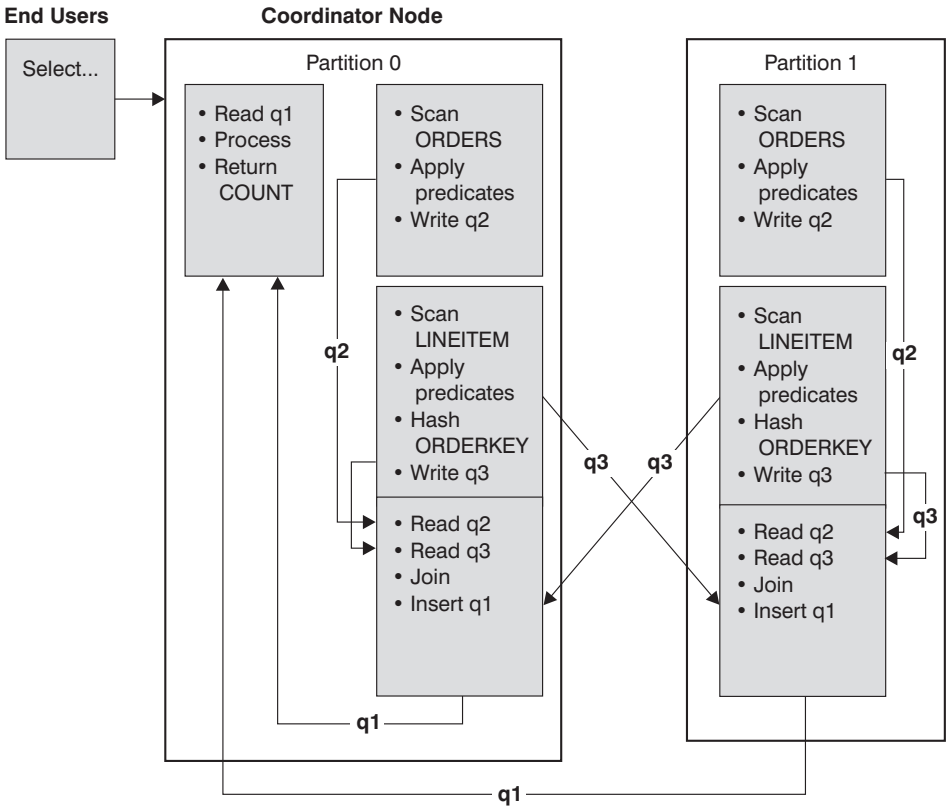


The LINEITEM table is sent to all database partitions that have the ORDERS table. Table queue q3 is broadcast to all database partitions of the outer table.

Figure 27. Broadcast Inner-Table Join Example

Directed Inner-Table Joins

With the directed inner-table join strategy, each row of the inner table is sent to one database partition of the outer join table, based on the splitting attributes of the outer table. The join occurs on this database partition. The following figure 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 ORDERS table database partition.
In this example, the join predicate is assumed to be:
ORDERS.ORDERKEY = LINEITEM.ORDERKEY.

Figure 28. Directed Inner-Table Join Example

Effects of sorting and grouping

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 sort data in one of the following ways:

- Piping the results of the sort when the query is executed.
- Internal handling of the sort within the database manager.

Piped versus 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 the 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 available in the sort heap, several sort passes are performed, in which each pass sorts a subset of the entire set of rows. Each sort 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 the sort passes are complete, these sorted subsets must be merged into a single sorted set of rows. If the sort is piped, the rows are handed directly to Relational Data Services as they are merged.

Group and sort pushdown operators

In some cases, the optimizer can choose to push down a sort or aggregation operation to Data Management Services from the Relational Data Services component. Pushing down these operations improves performance by allowing the Data Management Services component to pass data directly to a sort or aggregation routine. Without this pushdown, Data Management Services first passes this data to Relational Data Services, which then interfaces with the sort or aggregation routines. For example, the following query benefits from this optimization:

```
SELECT WORKDEPT, AVG(SALARY) AS AVG_DEPT_SALARY
  FROM EMPLOYEE
  GROUP BY WORKDEPT
```

Group operations in sorts

When sorting produces the order 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.

An aggregation in sort requires as many as the following three stages of aggregation to ensure that proper results are returned.

1. The first stage of aggregation, partial aggregation, calculates the aggregate values until the sort heap is filled. In partial aggregation unaggregated 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.
2. 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 intra-partition and inter-partition parallelism. In intra-partition parallelism, the grouping is finished when a global grouping key is available. In inter-partition 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 intra-partition parallelism when each agent finishes merging its spilled sort runs before reducing to a single agent to complete the aggregation.
3. 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. Sort cannot perform complete aggregation because they cannot guarantee that the sort will not split. Complete aggregation takes in unaggregated data and produces final aggregates. If the distribution does not prohibit its use, this method of aggregation is usually used to group data that is already in the correct order.

Optimization strategies

Optimization strategies for intra-partition 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 execution 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 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.

Intra-partition 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. A range of pages or rows is assigned to a subagent. A subagent scans its assigned range and is assigned another range when it has completed its 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 the parallel table scan with subagents being assigned a range of records. A subagent is assigned a new range when it has complete its work on the current range.

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 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, then 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.

Intra-partition 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 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 completed, each subagent reads the entire sort. If the number of rows is small, this sort may be used to rebalance the data stream.

- **Shared sort**

  This sort is the same as a replicated sort, except the subagents open a parallel scan on the sorted result to distribute the data among the subagents in a way similar to the round-robin sort.

Intra-partition 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.

Intra-partition parallel aggregation strategies

Aggregation operations can be performed in parallel by subagents. 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 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.

Intra-partition parallel join strategies

Join operations can be performed in parallel by subagents. 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 re-evaluated independently by each subagent. A merged join can be parallelized if its inner and outer streams are value-partitioned for partitioned sorts.

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.

Note: MDC table optimization strategies can also implement the performance advantages of intra-partition parallelism and inter-partition 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 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 quickly scan only the resulting blocks of the table to fetch the result set.

Rollout deletion

When conditions are met to allow delete using rollout, a more efficient way to delete rows from MDC tables is used. The conditions are:

- The DELETE statement is searched, not positioned (that is, does not use the "WHERE CURRENT OF" clause).
- No WHERE clause (all rows are to be deleted) or the only conditions in the WHERE clause are on 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 may qualify for rollout, if its foreign key is a subset of its 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).

For 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` with the SET CURRENT MDC ROLLOUT MODE statement. There is no change in the logging of index updates, as compared to a standard delete, so the performance improvement depends on how many RID indexes there are. The fewer RID indexes, the better the improvement is, as a percentage of the total time and log space.

An estimate of the amount of space saved in the log can be made with this formula, 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 containing the records deleted:

```
S + 38*N - 50*P
```

This figure is the reduction in actual log data. The saving on active log space requirement is double due to saving for space 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_ROLL_OUT** registry variable to DEFER or by specifying `DEFERRED` with the SET CURRENT MDC ROLLOUT MODE statement. In a deferred rollout, RID indexes are cleaned up asynchronously in the background after the commit of the delete. This method of rollout can result in significantly faster deletion times for very large deletes or when a number of RID indexes exist on a 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. As well, 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 DB2 is unable to allocate the memory structures it requires, the deferred cleanup rollout fails and a message is written to the administrator log.

When to use deferred cleanup rollout

If delete performance is the most important factor to you, 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. Here are other issues to consider when deciding between immediate index cleanup and deferred index cleanup:

- Size of delete: Choose deferred cleanup rollout for very large deletes. 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 the time saved during the delete.
- 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 statement 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 on all tables which have deferred cleanup pending.

To disable rollout behavior in deletions, you can set the **DB2_MDC_ROLLOUT** registry variable to OFFor specify NONE with the SET CURRENT MDC ROLLOUT MODE statement.

Optimization strategies for partitioned tables

Data partition elimination refers to the database servers ability to determine, based on the query predicates, that only a subset of the data partitions of a table need to be accessed to answer a query. Data partition elimination offers particular benefit 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 given table is partitioned into multiple storage objects based on the 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. If you issue the following statement:

```
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 you are interested in customer information for the year 2000. If you issue the following query:

```
SELECT * FROM custlist WHERE subsdate BETWEEN '1/1/2000' AND '12/31/2000';
```

As Figure 29 on page 306 shows, the database server determines that only one data partition in table space 4 (ts4) must be accessed to resolve this query.

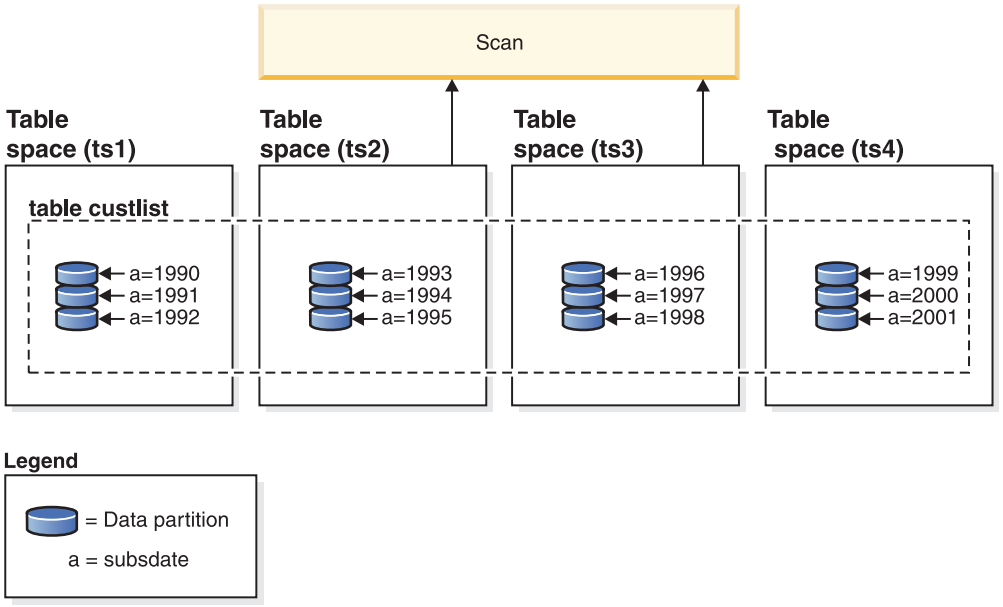


Figure 29. The performance benefits of data partition elimination on a partitioned table

Another example of data partition elimination shown in figure Figure 30, is an index scan involving two indexes and 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);
```

If you issue the following query:

```
SELECT * FROM multi WHERE
sale_date BETWEEN '6/1/2005' AND '7/31/2005' AND REGION = 'NW';
```

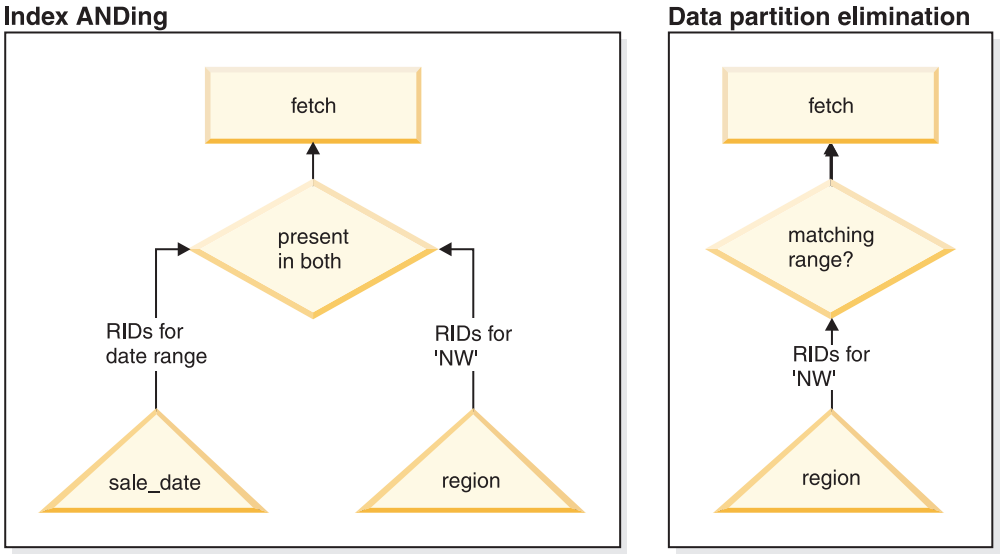


Figure 30. Optimizer decision path for both table partitioning and index ANDing

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 30 on page 306 demonstrates, with table partitioning, the index is read to find matches for both region and sale_date, allowing for fast retrieval of matching rows.

DB2 Explain

You can also use DB2 Explain to determine the partition elimination chosen by the DB2 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 for cases where multiple columns are used as the table partitioning key.

For example, if you issue the following statement:

```
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)
```

Next, issue the following query:

```
 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, since the non-leading columns used for the table partitioning key are not independent.

Multi-range support

It is possible to achieve data partition elimination on data partitions with multiple ranges (that is, OR'ed together). Using the table 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, you can issue the following statement:

```
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 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, if you have the following query:

```
 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 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)
```

Predicates 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 runtime when the necessary values are bound.

During runtime when T1 is the inner of a NLJN, data partition elimination occurs dynamically, based on the predicates, for every outer value of T2.A. During runtime 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 spaces ts6 and ts7 to be accessed.

Consider 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 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 spaces ts4 and ts5.
4. Scans the data partitions in table spaces ts4 and ts5 of table T1 until a row is found after applying T1.A = 2 and T1.B > 15. The first row found that qualifies is row 3 of T1.
5. Joins the matching row.
6. Scans the data partitions in table spaces ts4 and ts5 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 next row (replacing the value of A with 3) of T2 until all the rows from T2 are exhausted.

Materialized query tables

Materialized query tables (MQTs) are a powerful way to improve response time for complex queries, especially queries that might require some of the following operations:

- Aggregate data over one or more dimensions
- Joins and aggregate data over a group of tables
- Data from a commonly accessed subset of data, that is, from a "hot" horizontal or vertical database partition
- Repartitioned data from a table, or part of a table, in a partitioned database environment

Knowledge of MQTs is integrated into the SQL and XQuery compiler. In the compiler, the query rewrite phase and the optimizer match queries with MQTs and determine whether to substitute an MQT for a query that accesses the base tables. If an MQT is used, the EXPLAIN facility can provide information about which MQT was selected.

Because MQTs behave like regular tables in many ways, the same guidelines for optimizing data access using table space definitions, creating indexes, and issuing RUNSTATS apply to MQTs.

To help you understand the power of MQTs, the following example shows a multidimensional analysis query and how it takes advantage of MQTs.

In this example, assume 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. This schema is classified as a multi-star because has two large tables, one containing transaction items and the other identifying the purchase transactions.

Three hierarchical dimensions describe a transaction: product, location, and time. 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 or region information and is represented in a single de-normalized 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.

An MQT is created with the sum and count of sales for each level of the following hierarchies:

- Product
- Location
- Time, composed of year, month, day.

Many queries can be satisfied from this stored aggregate data. The following example shows how to create an MQT that computes sum and count of sales along the product group and line dimensions; along the city, state, and country dimension; 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.SALESCUBE;
```

Queries that can take advantage of such pre-computed sums would include the following:

- Sales by month and product group
- Total sales for years after 1990
- Sales for 1995 or 1996
- Sum of sales for a product group or product line
- Sum of sales for a specific product group or product line AND for 1995, 1996
- Sum of sales for a specific country.

While the precise answer is not included in the MQT for any of these queries, the cost of computing the answer using the MQT could be significantly less than using a large base table, because a portion of the answer is already computed. MQTs can reduce expensive joins, sorts, and aggregation of base data.

The following sample queries would obtain significant performance improvements because they can use the already computed results in the example MQT.

The first example returns the total sales for 1995 and 1996:

```
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);
```

The second example returns the total sales by product group for 1995 and 1996:

```
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;
```

The larger the base tables are, the larger the improvements in response time can be because the MQT grows more slowly than the base table. MQTs can effectively eliminate overlapping work among queries by doing the computation once when the MQTs are built and refreshed and reusing their content for many queries.

Explain facility

The SQL or XQuery compiler can capture information about the access plan and environment of static or dynamic SQL and XQuery statements. The captured information helps you understand how individual SQL or XQuery statements are executed so that you can tune the statements and your database manager configuration to improve performance.

You collect and use explain data for the following reasons:

- To understand how the database manager accesses tables and indexes to satisfy your query
- To evaluate your performance-tuning actions

  When you change some aspect of the database manager, the SQL or XQuery statements, or the database, you should examine the explain data to find out how your action has changed performance.

The captured information includes:

- Sequence of operations to process the query
- Cost information
- Predicates and selectivity estimates for each predicate
- Statistics for all objects referenced in the SQL or XQuery statement at the time that the explain information is captured
- Values for the host variables, parameter markers, or special registers used to reoptimize the SQL or XQuery statement.

Before you can capture explain information, you create the relational tables in which the optimizer stores the explain information and you set the special registers that determine what kind of explain information is captured.

To display explain information, you can use either a command-line tool or Visual Explain. The tool that you use determines how you set the special registers that determine what explain data is collected. For example, if you expect to use Visual Explain only, you need only capture snapshot information. 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, you should capture all explain information.

Guidelines for using explain information

You use explain information for the following two major purposes:

- To understand why application performance has changed

- To evaluate performance tuning efforts

Analysis of performance changes

To help you understand the reasons for changes in query performance, you need the before and after explain information which you can obtain by performing the following steps:

- Capture explain information for the query before you make any changes and save the resulting explain tables, or you might save the output from the `db2exfmt` explain tool.
- Save or print the current catalog statistics if you do not want to, or cannot, access Visual Explain to view this information. You might also use the `db2look` productivity tool to help perform this task.
- Save or print the data definition language (DDL) statements, including those for CREATE TABLE, CREATE VIEW, CREATE INDEX, 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 also collect this information at bind time. To analyze a performance change, you compare the information that you collected with information that you collect about the query and environment when you start your analysis.

As a simple example, your analysis might show that an index is no longer being used as part of the access path. Using the catalog statistics information in Visual Explain, 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 these actions:

- Reorganize the index
- Collect new statistics for your table and indexes
- Gather explain information when rebinding your query.

After you perform one of the actions, examine the access plan again. If the index is used again, performance of the query might no longer be a problem. If the index is still not used or if performance is still a problem, perform a second action 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 adjusting configuration parameters, adding containers, collecting fresh catalog statistics, and so on.

After you make a change in any of these areas, you can use the explain facility to determine the impact, if any, that the change has on the access plan chosen. For example, if you add an index or materialized query table (MQT) based on the index guidelines, the explain data can help you determine whether the index or materialized query table is actually used as you 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.

Guidelines for capturing explain information

Explain data is captured if you request it when an SQL or XQuery statement is compiled. Consider how you expect to use the captured information when you request explain data.

Note:

1. If incremental bind SQL or XQuery statements are compiled at run time, data is placed in the explain tables at run time and not bind time. For these statements, the explain table qualifier and authorization ID inserted is that of the package owner and not that of the user running the package.
2. Explain information is captured only when the SQL or XQuery statement is compiled. After the 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 statement is compiled and explain data is captured every time this statement is prepared or executed.
3. If a package is bound using the bind option REOPT ONCE/ALWAYS, SQL or XQuery statements containing host variables, parameter markers, global variables or special registers will be compiled and the access path will be created using real values of these variables if they are known, and using default estimates if the values are not known at compilation time.
4. If the FOR REOPT ONCE clause is used, then an attempt is made to match the specified SQL or XQuery statement against the same statement in the package cache. The values of this already reoptimized cached query statement will be used to reoptimize the specified query statement. The Explain tables will contain the newly generated reoptimized access plan and the values used for this reoptimization, if the user has the required access privileges.
5. In a multi-partition system, the statement should be explained on the same database partition on which it was originally compiled and reoptimized using REOPT ONCE, otherwise an error will be 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 commands 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:
  - 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 the 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 placed by the user in the ADVISE_INDEX table for evaluation. In EVALUATE INDEXES mode, all dynamic statements are explained as if these virtual indexes were available. The query compiler then chooses to use the virtual indexes if the 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 reoptimization at execution time, when actual values for the host variables, special registers, global variables or parameter markers are available.

- The EXPLAIN ALL option has been specified on the BIND or PREP command. The query compiler captures explain data for dynamic SQL and XQuery at run-time, even if the CURRENT EXPLAIN MODE special register is set to NO. The SQL or XQuery statement also executes and returns the results of the query.

Capturing explain snapshot information

When an explain snapshot is requested, explain information is stored in the SNAPSHOT column of the EXPLAIN_STATEMENT table in the format required by Visual Explain. This format is not usable by other applications. Additional information on the contents of the explain snapshot information is available from Visual Explain itself. This information includes information about data objects and data operators.

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 EXPLSNAP ALL or EXPLSNAP YES clauses are specified on the BIND or the PREP commands 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 FOR SNAPSHOT clause, only explain snapshot information is captured. With the WITH SNAPSHOT clause, all explain information is captured in addition snapshot information.

  – The CURRENT EXPLAIN SNAPSHOT special register is set to:

    - YES: The query compiler captures snapshot explain data and executes the SQL or XQuery statement.

    - EXPLAIN: The query compiler captures snapshot explain data, but does not execute the SQL or XQuery statement.

  – You specify the EXPLSNAP ALL option on the BIND or PREP command. The query compiler captures snapshot explain data at run-time, even if the setting of the CURRENT EXPLAIN SNAPSHOT special register is NO. It also executes the SQL or XQuery statement.

Guidelines for analyzing explain information

The primary use of explain information is analysis of the 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 kind of analysis:

- **Index use**

  The proper indexes can significantly benefit performance. Using the explain output, you can determine if the indexes you have created to help a specific set of queries are being used. In the explain output, you should 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 might be used instead of an existing index or no index at all. After you create a new index, use the RUNSTATS command to collect statistics for that index and recompile the query. Over time you may notice through the explain data that instead of an index scan, a table scan is now being used. 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 may 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, and then recompile the query. To determine whether reorganizing table has improved the access plan, re-examine the explain output for the recompiled query.

- **Access type**

  Analyze the explain output and look for types of access to the data that are not usually optimal for the type of application you are running. For example:

  - **Online transaction processing (OLTP) queries**

    OLTP applications are prime candidates to use index scans with range delimiting predicates, because they tend to return only a few rows that are qualified using an equality predicate against a key column. If your OLTP queries are using a table scan, you may want to analyze the explain data to determine the reasons why an index scan was not used.

  - **Browse-only queries**

    The search criteria for a "browse" type query may be very vague, causing a large number of rows to qualify. If users usually look at only a few screens of the 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 from 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, then 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 OLTP queries, typically run faster with nested-loop joins. However, there may be extenuating circumstances in either case, such as the use of local predicates or indexes, that might change how these typical joins work.

Using access plans to self-diagnose performance problems from REFRESH TABLE and SET INTEGRITY statements

EXPLAIN for REFRESH TABLE and SET INTEGRITY statements allow you to generate access plans that can be used to self-diagnose performance problems with these statements. This helps 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 set the CURRENT EXPLAIN MODE special register to NO after.

Restrictions:

- The REFRESH TABLE and SET INTEGRITY statements do not qualify for reoptimization, 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 also indicates that the specified explainable statement is to be reoptimized, is not applicable to the REFRESH TABLE and SET INTEGRITY statements.

Scenario

This scenario shows how you can generate and use access plan from EXPLAIN and REFRESH TABLE statements to self-diagnose the cause of your performance problems.

Step 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;
```

Step 2 Issue the EXPLAIN and REFRESH TABLE statements, as follows:

```
EXPLAIN PLAN FOR REFRESH TABLE MQT;
```

Note: 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;
```

Step 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
```

Step 4 Analyze the access plan to determine the cause of the performance problems. For example, by analyzing the plan from the above statements, if T is a large table, a TABLE SCAN would be very expensive, if it is used. Creating an index might improve the performance of the query.

Explain tools

DB2 provides 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 or methods give you the flexibility you need to capture, display, and analyze explain information.

Detailed optimizer information that allows for in-depth analysis of an access plan is stored in explain tables separate from the actual access plan itself. Use one or more of the following methods of getting information from the explain tables:

- Use Visual Explain to view explain snapshot information.

  Invoke Visual Explain from the Control Center to see a graphical display of a query access plan. You can analyze both static and dynamic SQL and XQuery statements.

  Visual Explain allows you to view snapshots captured or taken on another platform. For example, a Windows client can graph snapshots generated on a DB2 for HP-UX server.
- Use the db2exfmt tool to display explain information in preformatted output.
- Use the db2expln and dynexpln tools.

  To see the access plan information available for one or more packages of static SQL or XQuery statements, use the db2expln tool from the command line. db2expln shows the actual implementation of the chosen access plan. It does not show optimizer information.

  The dynexpln tool, which uses db2expln within it, provides a quick way to explain dynamic SQL or XQuery statements that contain no parameter markers. This use of db2expln from within dynexpln is done by transforming the input SQL or XQuery statement into a static statement within a pseudo-package. When this occurs, the information may not always be completely accurate. If complete accuracy is desired, use the explain facility.

  The db2expln tool does provide a relatively compact and English-like overview of what operations will occur at runtime by examining the actual access plan generated.
- Write your own queries against the explain tables.

  Writing your own queries allows for easy manipulation of the output and for comparison among different queries or for comparisons of the same query over time.

Note: The location of the command-line explain tools and others, such as db2batch, dynexpln, and db2_all, is in the misc subdirectory of the sqllib directory. If the tools are moved from this path, the command-line methods might not work.

The following table summarizes the different tools available with the DB2 explain facility and their individual characteristics. Use this table to select the tool most suitable for your environment and needs.

Table 64. Explain Facility Tools

Desired Characteristics	Visual Explain	Explain tables	db2exfmt	db2expln	dynexpln
GUI-interface	Yes				
Text output			Yes	Yes	Yes

Table 64. Explain Facility Tools (continued)

Desired Characteristics	Visual Explain	Explain tables	db2exfmt	db2expln	dynexpln
"Quick and dirty" static SQL and XQuery analysis				Yes	
Static SQL and XQuery supported	Yes	Yes	Yes	Yes	
Dynamic SQL and XQuery supported	Yes	Yes	Yes	Yes	Yes*
CLI applications supported	Yes	Yes	Yes		
Available to DRDA® Application Requesters		Yes			
Detailed optimizer information	Yes	Yes	Yes		
Suited for analysis of multiple statements		Yes	Yes	Yes	Yes
Information accessible from within an application		Yes			
Note:					
* Indirectly using db2expln; there are some limitations.					

Displaying catalog statistics in effect at explain time

The explain facility captures the statistics in effect at the time a statement is explained. These statistics may be different than those stored in the system catalogs, especially if real-time statistics gathering is enabled. If the explain tables are populated but an explain snapshot was not created, than only some statistics are recorded in the EXPLAIN_OBJECT table.

In order to capture all catalog statistics relevant to the statement being explained, create an explain snapshot at the same time the explain tables are populated. Then use the SYSPROC.EXPLAIN_FORMAT_STATS function to format the catalog statistics in the snapshot.

If the db2exfmt tool is used to format the explain information, it automatically uses the SYSPROC.EXPLAIN_FORMAT_STATS function to display the catalog statistics, if a snapshot was collected. Visual Explain automatically displays all statistics contained in the snapshot.

SQL and XQuery Explain tools

The `db2expln` tool describes the access plan selected for SQL and XQuery statements. It can be used 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 stored in the system catalog tables. For dynamic SQL and XQuery statements, `db2expln` examines the sections in the query cache.

The `dynexpln` tool can also be used to describe the access plan selected for dynamic statements. It creates a static package for the statements and then uses the `db2expln` tool to describe them. However, because the dynamic SQL and XQuery statements can be examined by `db2expln`, this utility is retained only for backward compatibility.

The explain tools (db2expln and dynexpln) are located in the bin subdirectory of your instance sqllib directory. If db2expln and dynexpln are not in your current directory, they must be in a directory that appears in your PATH environment variable.

The db2expln program connects and uses the db2expln.bnd, db2exsrv.bnd, and db2exdyn.bnd files to bind itself to a database the first time the database is accessed.

To run db2expln, you must have the SELECT privilege on the system catalog views as well as the EXECUTE privilege for the db2expln, db2exsrv, and db2exdyn packages. To run dynexpln, you must have BINDADD authority for the database, and the SQL schema you are using to connect to the database must exist or you must have the IMPLICIT_SCHEMA authority for the database. To explain dynamic SQL and XQuery statements using either db2expln or dynexpln, you must also have any privileges needed for the query statements being explained. (Note that if you have SYSADM or DBADM authority, you will automatically have all these authorization levels.)

dynexpln

The dynexpln tool is still available for backward compatibility. However, you can use the **dynamic-options** of db2expln to perform all of the functions of dynexpln.

When you use the dynamic-options of db2expln, the statement is prepared as true dynamic SQL or XQuery statement and the generated plan is explained from the query cache. This explain-output method provides more accurate access plans than dynexpln, which prepares the statement as a static SQL or XQuery statement. It also allows the use of features available only in dynamic SQL and XQuery statements, such as parameter markers.

Usage notes for dynexpln: To explain dynamic statements, dynexpln creates a static application for the statements and then invokes db2expln. To create the static statements, dynexpln generates a trivial C program with the statements and then calls the DB2 precompiler to create the package. (The generated C program is not complete and cannot be compiled; it only contains enough information for the precompiler to build the package.)

The following are common messages displayed by dynexpln:

- All error messages from db2expln.

  Since dynexpln invokes db2expln, it is possible to see most of db2expln's error messages.
- `Error connecting to the database.`

  This message will appear in the output if an error occurred connecting to the database. A CLI error message will also be displayed indicating why the connection could not be completed. Correct the cause of the error and run dynexpln again.
- `The file "<filename>" must be removed before dynexpln will run.`

  This message will appear if the given file exists at the time dynexpln is run. Remove the file or change the value of the DYNEXPLN_PACKAGE environment variable to change the name of the file which will be created and run dynexpln again.
- `The package "<creator>.<package>" must be dropped before dynexpln will run.`

This message will appear if the given package exists at the time `dynexpln` is run. Drop the package and run or change the value of the **DYNEXPLN_PACKAGE** environment variable to change the name of the package which will be created and run `dynexpln` again.

- `Error writing file "<filename>".`

  This message will appear if the given file cannot be written to. Ensure that `dynexpln` can write files in the current directory and run it again.

- `Error reading input file "<filename>".`

  This message will appear if the file given with the **-f** option cannot be read from. Ensure that the file exists and that `dynexpln` can read it. Then run `dynexpln` again.

Environment Variables: There are two different environment variables that can be used in conjunction with `dynexpln`:

- **DYNEXPLN_OPTIONS** are the SQL and XQuery precompiler options you use when building the package for your statements. Use the same syntax variable as you would when issuing a PREP command through CLP.

  For example: `DYNEXPLN_OPTIONS="OPTLEVEL 5 BLOCKING ALL"`

- **DYNEXPLN_PACKAGE** is the name of the package which is created in the database. The statements to be described are placed in this package. If this variable is not defined, the package is given a default value of **DYNEXPLN**. (Only the first eight characters of the name in this environment variable are used.)

  The name is also used to create the names for the intermediate files that `dynexpln` uses.

Description of db2expln and dynexpln output

In the output, the explain information for each package appears in the following two parts:

- Package information such as date of bind and relevant bind options
- Section information such as the section number, followed by the SQL or XQuery statement being explained. Below the section information, the explain output of the access plan chosen for the SQL or XQuery statement appears.

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-justified heading with information about that step indented below it. Indentation bars appear in the left margin of the explain output for the access plan. These bars also mark the scope of the operation. Operations at a lower level of indentation, farther to the right, in the same operation are processed before returning to the previous level of indentation.

Remember that the access plan chosen was based on an augmented version of the original SQL or XQuery statement that is shown in the output. For example, the original statement may cause triggers and constraints to be activated. In addition, the query rewrite component of the query compiler might rewrite the SQL or XQuery statement to an equivalent but more efficient format. All of these factors are included in the information that the optimizer uses when it determines the most efficient plan to satisfy the statement. Thus, the access plan shown in the explain output may differ substantially from the access plan that you might expect for the original SQL or XQuery statement. The explain facility, which includes the explain tables, SET CURRENT EXPLAIN mode, and Visual Explain, shows the actual SQL or XQuery statement used for optimization in the form of an SQL- or XQuery-like statement which is created by reverse-translating the internal representation of the query.

When you compare output from db2expln or dynexpln to the output of the Explain facility, the operator ID option (**-opids**) can be very useful. Each time db2expln or dynexpln starts 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 match up the steps in the different representations of the access plan. Note that there is not always a one-to-one correspondence between the operators in the Explain facility output and the operations shown by db2expln and dynexpln.

Table access information: This statement tells the name and type of table being accessed. It has two formats that are used:

1. Regular tables of three types:
   - Access Table Name:

     ```
     Access Table Name = schema.name  ID = ts,n
     ```

     where:
     - *schema.name* is the fully-qualified name of the table being accessed
     - *ID* is the corresponding TABLESPACEID and TABLEID from the SYSCAT.TABLES catalog for the table
   - Access Hierarchy Table Name:

     ```
     Access Hierarchy Table Name = schema.name  ID = ts,n
     ```

     where:
     - *schema.name* is the fully-qualified name of the table being accessed
     - *ID* is the corresponding TABLESPACEID and TABLEID from the SYSCAT.TABLES catalog for the table
   - Access Materialized Query Table Name:

     ```
     Access Materialized Query Table Name = schema.name  ID = ts,n
     ```

     where:
     - *schema.name* is the fully-qualified name of the table being accessed
     - *ID* is the corresponding TABLESPACEID and TABLEID from the SYSCAT.TABLES catalog for the table
2. Temporary tables of two types:
   - Access Temporary Table ID:

     ```
     Access Temp Table  ID = tn
     ```

     where:
     - *ID* is the corresponding identifier assigned by db2expln
   - Access Declared Global Temporary Table ID:

     ```
     Access Global Temp Table  ID = ts,tn
     ```

     where:
     - *ID* is the corresponding TABLESPACEID from the SYSCAT.TABLES catalog for the table (ts); and the corresponding identifier assigned by db2expln (tn)

Following the table access statement, additional statements will be provided to further describe the access. These statements will be indented under the table access statement. The possible statements are:

- Number of columns
- Block access

- Parallel scan
- Scan directive
- Row access method
- Lock intents
- Predicates
- Miscellaneous statements

Number of Columns

The following statement indicates the number of columns being used from each row of the table:

```
#Columns = n
```

Block Access

The following statement indicates that the table has one or more dimension block indexes defined on it:

```
Clustered by Dimension for Block Index Access
```

If this text is not shown, the table was created without the DIMENSION clause.

Parallel Scan

The following statement indicates that the database manager will use several subagents to read from the table in parallel:

```
Parallel Scan
```

If this text is not shown, the table will only be read from by one agent (or subagent).

Scan Direction

The following statement indicates that the database manager will read rows in a reverse order:

```
Scan Direction = Reverse
```

If this text is not shown, the scan direction is forward, which is the default.

Row Access Method

One of the following statements will be displayed, indicating how the qualifying rows in the table are being accessed:

- The `Relation Scan` statement indicates that the table is being sequentially scanned to find the qualifying rows.
  - The following statement indicates that no prefetching of data will be done:

    ```
    Relation Scan
    |  Prefetch: None
    ```

  - The following statement indicates that the optimizer has predetermined the number of pages that will be prefetched:

    ```
    Relation Scan
    |  Prefetch: n Pages
    ```

  - The following statement indicates that data should be prefetched:

```
Relation Scan
|  Prefetch: Eligible
```

– The following statement indicates that the 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 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 will be followed by one row for each column in the index. Each column in the index will be listed in one of the following forms:

```
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:

```
#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 will be included in the index scan. An exclusive key value will not be included in the scan. The value for the key will be given by one of the following rows 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: ?
```

  If a literal string is shown, only the first 20 characters are displayed. If the string is longer than 20 characters, this will be shown by `...` at the end of the string. Some keys cannot be determined until the section is executed. This is shown by a `?` as the value.

- `Index-Only Access`

  If all the needed columns can be obtained from the index key, this statement will appear and no table data will be accessed.

- The following statement indicates that no prefetching of index pages will be done:

```
Index Prefetch: None
```

- The following statement indicates that index pages should be prefetched:

```
Index Prefetch: Eligible
```

- The following statement indicates that no prefetching of data pages will be done:

  ```
  Data Prefetch: None
  ```

- The following statement indicates that data pages should be prefetched:

  ```
  Data Prefetch: Eligible
  ```

- 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 predicates:

  ```
  Sargable Index Predicate(s)
  |  #Predicates = n
  ```

– If the qualifying rows are being accessed by using row IDs (RIDs) that were prepared earlier in the access plan, it will be indicated with the statement:

  ```
  Fetch Direct Using Row IDs
  ```

  If the table has one or more block indexes defined for it, then rows may be accessed by either block or row IDs. This is indicated by:

  ```
  Fetch Direct Using Block or Row IOs
  ```

Lock Intents

For each table access, the type of lock that will be 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 Exclusive (does not appear in `db2expln` output)
- None
- Share
- Next Key Share
- Update
- Next Key Weak Exclusive
- Weak Exclusive

Predicates

There are two statements that provide information about the predicates used in an access plan:

1. The following statement indicates the number of predicates that will be evaluated for each block of data retrieved from a blocked index.

   ```
   Block Predicates(s)
   |  #Predicates = n
   ```

2. The following statement indicates the number of predicates that will be evaluated while the data is being accessed. The count of predicates does not include push-down operations such as aggregation or sort.

   ```
   Sargable Predicate(s)
   |  #Predicates = n
   ```

3. The following statement indicates the number of predicates that will be evaluated once the data has been returned:

   ```
   Residual Predicate(s)
   |  #Predicates = n
   ```

The number of predicates shown in the above statements may not reflect the number of predicates 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 will be accessed:

  ```
  Single Record
  ```

- The following statement appears when the isolation level used for this table access uses a different isolation level than the statement:

  ```
  Isolation Level: xxxx
  ```

  A different isolation level may be used for a number of reasons, including:

  – A package was bound with Repeatable Read and affects referential integrity constraints; the access of the parent table to check referential integrity constraints is downgraded to an isolation level of Cursor Stability to avoid holding unnecessary locks on this table.

  – A package bound with Uncommitted Read issues a DELETE or UPDATE statement; the table access for the actual delete is upgraded to Cursor Stability.

- The following statement indicates that some or all of the rows read from the temporary table will be cached outside the buffer pool if sufficient sortheap memory is available:

  ```
  Keep Rows In Private Memory
  ```

- If the table has the volatile cardinality attribute set, it will be indicated by:

  ```
  Volatile Cardinality
  ```

Temporary table information: A temporary table is used by an access plan to store data during its execution in a transient or temporary work table. This table only exists while the access plan is being executed. Generally, temporary tables are used when subqueries need to be evaluated early in the access plan, or when intermediate results will not fit in the available memory.

If a temporary table needs to be created, then one of two possible statements may appear. These statements indicate that a temporary table is to be created and rows

inserted into it. The ID is an identifier 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.

- The following statement indicates an ordinary temporary table will be created:

  ```
  Insert Into Temp Table  ID = tn
  ```

- The following statement indicates an ordinary temporary table will be created by multiple subagents in parallel:

  ```
  Insert Into Shared Temp Table  ID = tn
  ```

- The following statement indicates a sorted temporary table will be created:

  ```
  Insert Into Sorted Temp Table  ID = tn
  ```

- The following statement indicates a sorted temporary table will be created by multiple subagents in parallel:

  ```
  Insert Into Sorted Shared Temp Table  ID = tn
  ```

- The following statement indicates a declared global temporary table will be created:

  ```
  Insert Into Global Temp Table  ID = ts,tn
  ```

- The following statement indicates a declared global temporary table will be created by multiple subagents in parallel:

  ```
  Insert Into Shared Global Temp Table  ID = ts,tn
  ```

- The following statement indicates a sorted declared global temporary table will be created:

  ```
  Insert Into Sorted Global Temp Table  ID = ts,tn
  ```

- The following statement indicates a sorted declared global temporary table will be created by multiple subagents in parallel:

  ```
  Insert Into Sorted Shared Global Temp Table  ID = ts,tn
  ```

Each of the above statements will be followed by:

```
#Columns = n
```

which indicates how many columns are in each row 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 additional statements may follow the original creation statement for a sorted temporary table:

- The following statement indicates the number of key columns used in the sort:

  ```
  #Sort Key Columns = n
  ```

  For each column in the sort key, one of the following lines will be displayed:

```
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
```

- If only the first rows of the sorted result are needed, the following is displayed:

```
Sort Limited To Estimated Row Count
```

- For sorts in a symmetric multiprocessor (SMP) environment, the type of sort 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 result from the sort will be left in the sort heap:

```
Piped
```

  and

```
Not Piped
```

  If a piped sort is indicated, the database manager will keep the sorted output in memory, rather than placing the sorted result in another temporary table.

- The following statement indicates that duplicate values will be removed during the sort:

```
Duplicate Elimination
```

- If aggregation is being performed in the sort, it will be indicated by one of the following statements:

```
Partial Aggregation
Intermediate Aggregation
Buffered Partial Aggregation
Buffered Intermediate Aggregation
```

Temporary Table Completion

After a table access that contains a push-down operation to create a temporary table (that is, a create temporary table that occurs within the scope of a table access), there will be a ″completion″ statement, which handles end-of-file by getting the temporary table ready to provide rows to subsequent temporary table access. One of the following lines will be displayed:

```
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. Table functions are indicated by:

```
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
```

The specific name uniquely identifies the table function invoked. The remaining rows detail the attributes of the function.

Join information: There are three types of joins:

- Hash join
- Merge join
- Nested loop join.

When the time comes in the execution of a section for a join to be performed, one of the following statements is displayed:

```
Hash Join
Merge Join
Nested Loop Join
```

It is possible for a left outer join to be performed. 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
```

For merge and nested loop joins, the outer table of the join will be the table referenced in the previous access statement shown in the output. The inner table of the join will be the table referenced in the access statement that is contained within the scope of the join statement. For hash joins, the access statements are reversed with the outer table contained within the scope of the join and the inner table appearing before the join.

For a hash or merge join, the following additional statements may appear:

- In some circumstances, a join simply needs to determine if any row in the inner table matches the current row in the outer. This is indicated with the statement:

  ```
  Early Out: Single Match Per Outer Row
  ```

- It is possible to apply predicates after the join has completed. The number of predicates being applied will be indicated as follows:

  ```
  Residual Predicate(s)
  |  #Predicates = n
  ```

For a hash join, the following additional statements may appear:

- The hash table is built from the inner table. If the hash table build was pushed down into a predicate on the inner table access, it is indicated by the following statement in the access of the inner table:

  ```
  Process Hash Table For Join
  ```

- While accessing the outer table, a probe table can be built to improve the performance of the join. The probe table build is indicated by the following statement in the access of the outer table:

  ```
  Process Probe Table For Hash Join
  ```

- The estimated number of bytes needed to build the hash table is represented by:

  ```
  Estimated Build Size: n
  ```

- The estimated number of bytes needed for the probe table is represented by:

  ```
  Estimated Probe Size: n
  ```

For a nested loop join, the following additional statement may 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 row that qualifies from `W`.
2. Join row from `W` with (next) row from `X` and call the result `P1` (for partial join result number 1).
3. Join `P1` with (next) row from `Y` to create `P2` .
4. Join `P2` with (next) row from `Z` to obtain 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 allows 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:

```
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 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 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
  ```

  and

  ```
  Not Piped
  ```

As was the case with temporary tables, a piped data stream may be written to disk, if insufficient memory exists at execution time. The access plan will provide 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 these SQL statements is self-explanatory. Possible statement text for these SQL operations 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 (IDs) are sorted and duplicates removed (in the case of index ORing) or that a technique is used to identify IDs appearing in all indexes being accessed (in the case of index ANDing) before the actual table access is performed. There are three main uses of ID preparation as indicated by the explain statements:

- Either of the following statements indicates that Index ORing is used to prepare the list of qualifying IDs:

  ```
  Index ORing Preparation
  Block Index ORing Preparation
  ```

  Index ORing refers to the technique of making more than one index access and combining the results to include the distinct IDs that appear in any of the indexes accessed. The optimizer will consider index ORing when predicates are connected by OR keywords or there is an IN predicate. The index accesses can be on the same index or different indexes.

- Another use of ID preparation is to prepare the input data to be used during list prefetch, as indicated by either of the following:

  ```
  List Prefetch Preparation
  Block List Prefetch RID Preparation
  ```

- *Index ANDing* refers to the technique of making more than one index access and combining the results to include IDs that appear in all of the indexes accessed. Index ANDing processing is started with either of these 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 IDs and use bit filter techniques to determine the IDs which appear in every index accessed. The following statements indicate that IDs are being 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
```

For any type of ID preparation, if list prefetch can be performed it will be indicated with the statement:

```
Prefetch: Enabled
```

Aggregation information: Aggregation is performed on those rows meeting the specified criteria, if any, provided by the SQL statement predicates. If some sort of aggregate function is to be done, one of the following statements appears:

```
Aggregation
Predicate Aggregation
Partial Aggregation
Partial Predicate Aggregation
Intermediate Aggregation
Intermediate Predicate Aggregation
Final Aggregation
Final Predicate Aggregation
```

Predicate aggregation states that the aggregation operation has been pushed-down to be processed as a predicate when the data is actually accessed.

Beneath either of the above aggregation statements will be a indication of the type of aggregate function being 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 is used, then subsequent to the table access statement in which the aggregation appeared, there will be an aggregation "completion", which carries out any needed processing on completion of each group or on end-of-file. One of the following lines is displayed:

```
Aggregation Completion
Partial Aggregation Completion
Intermediate Aggregation Completion
Final Aggregation Completion
```

Parallel processing information: Executing an SQL statement in parallel (using either intra-partition or inter-partition parallelism) requires some special operations. The operations for parallel plans are described below.

- When running an intra-partition parallel plan, portions of the plan will be executed simultaneously using several subagents. The creation of the subagents is indicated by the statement:

  ```
  Process Using n Subagents
  ```

- When running an inter-partition parallel plan, the section is broken into several subsections. Each subsection is sent to one or more nodes to be run. An important subsection is the *coordinator subsection*. The coordinator subsection is the first subsection in every plan. It gets 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 statement:

  ```
  Distribute Subsection #n
  ```

  The nodes that receive a subsection can be determined in one of eight ways:

  – The following indicates that the subsection will be sent to a node 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 indicates that the subsection will be sent to a predetermined node. (This is frequently seen when the statement uses the NODENUMBER() function.)

    ```
    Directed by Node Number
    ```

  – The following indicates that the subsection will be sent to the node corresponding to a predetermined database partition number in the given database partition group. (This is frequently seen when the statement uses the PARTITION() function.)

    ```
    Directed by Partition Number
    |  Partition Map ID = n, Nodegroup = ngname, #Nodes = n
    ```

  – The following indicates that the subsection will be sent to the node that provided the current row for the application's cursor.

    ```
    Directed by Position
    ```

  – The following indicates that only one node, determined when the statement was compiled, will receive the subsection.

    ```
    Directed to Single Node
    |  Node Number = n
    ```

  – Either of the following indicates that the subsection will be executed on the coordinator node.

    ```
    Directed to Application Coordinator Node
    Directed to Local Coordinator Node
    ```

  – The following indicates that the subsection will be sent to all the nodes listed.

    ```
    Broadcast to Node List
    |  Nodes = n1, n2, n3, ...
    ```

  – The following indicates that only one node, 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. Table queues are described as follows:

  – 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 inserted into the table queue is described by one of the following:

  All rows are sent to the coordinator node:

  ```
  Broadcast to Coordinator Node
  ```

  All rows are sent to every database partition where the given subsection is running:

  ```
  Broadcast to All Nodes of Subsection n
  ```

  Each row is sent to a database partition 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 node:

  ```
  Send to Random Node
  ```

- In some situations, a database partition table queue will have to temporarily overflow some rows to a temporary table. This possibility is identified by the statement:

  ```
  Rows Can Overflow to Temporary Table
  ```

- After a table access that contains a push-down operation to insert rows into a table queue, there will be a "completion" statement which handles rows that could not be immediately sent. 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 messages are always followed by an indication of the number of columns being retrieved.

  ```
  #Columns = n
  ```

- If the table queue sorts the rows at the receiving end, the table queue access will also have one of the following messages:

  ```
  Output Sorted
  Output Sorted and Unique
  ```

  These messages are followed by an indication of the number of keys used for the sort operation.

  ```
  #Key Columns = n
  ```

  For each column in the sort key, one of the following is displayed:

  ```
  Key n: (Ascending)
  Key n: (Descending)
  ```

– If predicates will be applied to rows by the receiving end of the table queue, the following message is shown:

```
Residual Predicate(s)
|  #Predicates = n
```

- Some subsections in a partitioned database environment explicitly loop back to the start of the subsection with the statement:

```
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 indicates that a data source will be read:

```
Ship Distributed Subquery #n
|   #Columns = n
```

It is possible to apply predicates to the data returned from the distributed subquery. The number of predicates being applied will be indicated as follows:

```
Residual Predicate(s)
|   #Predicates = n
```

An insert, update, or delete operation that occurs at a data source will be indicated by the appropriate message:

```
Ship Distributed Insert #n
Ship Distributed Update #n
Ship Distributed Delete #n
```

If a table is being explicitly locked at a data source, this will be indicated with the statement:

```
Ship Distributed Lock Table #n
```

DDL statements against a data source are split into two parts. The part invoked at the data source is indicated by:

```
Ship Distributed DDL Statement #n
```

If the federated server is a partitioned database, then part of the DDL statement must be run at he catalog node. This is indicated by:

```
Distributed DDL Statement #n Completion
```

The detail for each distributed substatement is provided separately. The options for distributed statements are described below:

- The data source for the subquery is shown by one of the following:

```
Server: server_name (type, version)
Server: server_name (type)
Server: server_name
```

- If the data source is relational, the SQL for the substatement is displayed as:

```
SQL Statement:
     statement
```

  Non-relational data sources are indicated with:

```
Non-Relational Data Source
```

- The nicknames referenced in the substatement are listed as follows:

```
Nicknames Referenced:
   schema.nickname ID = n
```

  If the data source is relational, the base table for the nickname is shows as:

```
Base = baseschema.basetable
```

If the data source is non-relational, the source file for the nickname is shown as:

```
Source File = filename
```

- If values are passed from the federated server to the data source before executing the substatement, the number of values will be shown by:

  ```
  #Input Columns: n
  ```

- If values are passed from the data source to the federated server after executing the substatement, the number of values will be shown by:

  ```
  #Output Columns: n
  ```

Miscellaneous explain information:

- Sections for data definition language statements will be indicated in the output with the following:

  ```
  DDL Statement
  ```

  No additional explain output is provided for DDL statements.

- Sections for SET statements for the updatable special registers such as **CURRENT EXPLAIN SNAPSHOT** will be indicated in the output with the following:

  ```
  SET Statement
  ```

  No additional explain output is provided for SET statements.

- If the SQL statement contains the DISTINCT clause, the following text may 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 be ordered so that duplicates can be skipped. 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 the columns in the index key are part of the DISTINCT operation
  – Duplicates that can be eliminated during sorting.

- The following statement will appear if the next operation is dependent on a specific record identifier:

  ```
  Positioned Operation
  ```

  If the position operation is against a federated data source, then the statement is:

  ```
  Distributed Positioned Operation
  ```

  This statement would appear for any SQL statement that uses the WHERE CURRENT OF syntax.

- The following statement will appear 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 will appear if there is a UNION operator in the SQL statement:

  ```
  UNION
  ```

- The following statement will appear 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 will appear:

```
Access Table Constructor
```

- The following statement will appear if there is an operation which is only processed under certain conditions:

```
Conditional Evaluation
|  Condition #n:
|  #Predicates = n
|  Action #n:
```

Conditional evaluation is used to implement such activities as the SQL 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 will appear 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 and 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 information will appear for deletes on multidimensional clustering tables that qualify for roll out optimization:

```
CELL DELETE with deferred cleanup
```

or

```
CELL DELETE with immediate cleanup
```

- The following statement will appear if there are rows being returned to the application:

```
Return Data to Application
|  #Columns = n
```

If the operation was pushed-down into a table access, it will require a completion phase. This phase appears as:

```
Return Data Completion
```

- The following information will 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 information will appear if one or more LOB locators are being freed:

```
     Free LOB Locators
```

Examples of db2expln and dynexpln output

The examples shown here can help you understand the layout and format of the output from `db2expln` and `dynexpln`. These examples were run against the SAMPLE database that is provided with DB2, unless shown otherwise. A brief discussion is included with each example. Significant differences from one example to the next have been shown in **bold**.

Example one: no parallelism: This example is simply requesting a list of all employee names, their jobs, department name and location, and the project names on which they are working. The essence of this access plan is that hash joins are used to join the relevant data from each of the specified tables. Since no indexes are available, the access plan does a relation scan of each table as it is joined.

```
******************** PACKAGE ****************************************

Package Name = "DOOLE"."EXAMPLE"  Version = ""

 Prep Date = 2002/01/04
 Prep Time = 14:05:00

 Bind Timestamp = 2002-01-04-14.05.00.415403

 Isolation Level          = Cursor Stability
 Blocking                 = Block Unambiguous Cursors
 Query Optimization Class = 5

 Partition Parallel       = No
 Intra-Partition Parallel = No

 SQL Path                 = "SYSIBM", "SYSFUN", "SYSPROC", "DOOLE"

-------------------- SECTION ---------------------------------------
Section = 1


SQL Statement:
  DECLARE EMPCUR CURSOR
  FOR
     SELECT e.lastname, e.job, d.deptname, d.location, p.projname
     FROM employee AS e, department AS d, project AS p
     WHERE e.workdept = d.deptno AND e.workdept = p.deptno


Estimated Cost        = 120.518692
Estimated Cardinality = 221.535980

(    6) Access Table Name = DOOLE.EMPLOYEE  ID = 2,5
         |  #Columns = 3
         |  Relation Scan
         |  |  Prefetch: Eligible
         |  Lock Intents
         |  |  Table: Intent Share
         |  |  Row  : Next Key Share
(    6) |  Process Build Table for Hash Join
(    2) Hash Join
         |  Estimated Build Size: 7111
         |  Estimated Probe Size: 9457
(    5)  |  Access Table Name = DOOLE.PROJECT  ID = 2,7
         |  |  #Columns = 2
         |  |  Relation Scan
         |  |  |  Prefetch: Eligible
         |  |  Lock Intents
         |  |  |  Table: Intent Share
```

```
        |  |  |  Row  : Next Key Share
(    5) |  |  Process Build Table for Hash Join
(    3) |  Hash Join
        |  |  Estimated Build Size: 5737
        |  |  Estimated Probe Size: 6421
(    4) |  |  Access Table Name = DOOLE.DEPARTMENT  ID = 2,4
        |  |  |  #Columns = 3
        |  |  |  Relation Scan
        |  |  |  |  Prefetch: Eligible
        |  |  |  Lock Intents
        |  |  |  |  Table: Intent Share
        |  |  |  |  Row  : Next Key Share
(    4) |  |  |  Process Probe Table for Hash Join
(    1) Return Data to Application
        |  #Columns = 5

End of section


Optimizer Plan:

              RETURN
              (   1)
                |
              HSJOIN
              (   2)
              /     \
         HSJOIN       TBSCAN
         (   3)       (   6)
         /    \         |
    TBSCAN    TBSCAN   Table:
    (   4)    (   5)   DOOLE
      |         |      EMPLOYEE
 Table:      Table:
 DOOLE       DOOLE
 DEPARTMENT  PROJECT
```

The first part of the plan accesses the DEPARTMENT and PROJECT tables and uses a hash join to join them. The result of this join is joined to the EMPLOYEE table. The resulting rows are returned to the application.

Example two: single-partition plan with intra-partition parallelism: This example shows the same SQL statement as the first example, but this query has been compiled for a four-way SMP machine.

```
******************** PACKAGE ************************************

Package Name = "DOOLE"."EXAMPLE"  Version = ""

 Prep Date = 2002/01/04
 Prep Time = 14:12:38

 Bind Timestamp = 2002-01-04-14.12.38.732627

 Isolation Level          = Cursor Stability
 Blocking                 = Block Unambiguous Cursors
 Query Optimization Class = 5

 Partition Parallel       = No
 Intra-Partition Parallel = Yes (Bind Degree = 4)

 SQL Path                 = "SYSIBM", "SYSFUN", "SYSPROC", "DOOLE"

-------------------- SECTION ----------------------------------------
Section = 1
```

```
SQL Statement:
  DECLARE EMPCUR CURSOR
  FOR
     SELECT e.lastname, e.job, d.deptname, d.location, p.projname
     FROM employee AS e, department AS d, project AS p
     WHERE e.workdept = d.deptno AND e.workdept = p.deptno

Intra-Partition Parallelism Degree = 4

Estimated Cost        = 133.934692
Estimated Cardinality = 221.535980

(    2) Process Using 4 Subagents
(    7) |  Access Table Name = DOOLE.EMPLOYEE  ID = 2,5
        |  |  #Columns = 3
        |  |  Parallel Scan
        |  |  Relation Scan
        |  |  |  Prefetch: Eligible
        |  |  Lock Intents
        |  |  |  Table: Intent Share
        |  |  |  Row  : Next Key Share
(    7) |  |  Process Build Table for Hash Join
(    3)  Hash Join
        |  |  Estimated Build Size: 7111
        |  |  Estimated Probe Size: 9457
(    6) |  |  Access Table Name = DOOLE.PROJECT  ID = 2,7
        |  |  |  #Columns = 2
        |  |  |  Parallel Scan
        |  |  |  Relation Scan
        |  |  |  |  Prefetch: Eligible
        |  |  |  Lock Intents
        |  |  |  |  Table: Intent Share
        |  |  |  |  Row  : Next Key Share
(    6) |  |  |  Process Build Table for Hash Join
(    4)  |  Hash Join
        |  |  |  Estimated Build Size: 5737
        |  |  |  Estimated Probe Size: 6421
(    5) |  |  |  Access Table Name = DOOLE.DEPARTMENT  ID = 2,4
        |  |  |  |  #Columns = 3
        |  |  |  |  Parallel Scan
        |  |  |  |  Relation Scan
        |  |  |  |  |  Prefetch: Eligible
        |  |  |  |  Lock Intents
        |  |  |  |  |  Table: Intent Share
        |  |  |  |  |  Row  : Next Key Share
(    5) |  |  |  |  Process Probe Table for Hash Join
(    2) |  Insert Into Asynchronous Local Table Queue  ID = q1
(    2) Access Local Table Queue  ID = q1  #Columns = 5
(    1) Return Data to Application
        |  #Columns = 5

End of section


Optimizer Plan:

              RETURN
              (   1)
                |
               LTQ
              (   2)
                |
              HSJOIN
              (   3)
             /      \
        HSJOIN         TBSCAN
```

```
        (   4)          (   7)
       /      \           |
    TBSCAN    TBSCAN   Table:
    (   5)    (   6)   DOOLE
      |         |      EMPLOYEE
 Table:      Table:
 DOOLE       DOOLE
 DEPARTMENT  PROJECT
```

This plan is almost identical to the plan in the first example. The main differences are the creation of four subagents when the plan first starts and the table queue at the end of the plan to gather the results of each of subagent's work before returning them to the application.

Example three: multipartition plan with inter-partition parallelism: This example shows the same SQL statement as the first example, but this query has been compiled on a partitioned database made up of three database partitions.

```
******************** PACKAGE ************************************

Package Name = "DOOLE"."EXAMPLE"  Version = ""

 Prep Date = 2002/01/04
 Prep Time = 14:54:57

 Bind Timestamp = 2002-01-04-14.54.57.033666

 Isolation Level          = Cursor Stability
 Blocking                 = Block Unambiguous Cursors
 Query Optimization Class = 5

 Partition Parallel       = Yes
 Intra-Partition Parallel = No

 SQL Path                 = "SYSIBM", "SYSFUN", "SYSPROC", "DOOLE"

-------------------- SECTION ---------------------------------------
Section = 1


SQL Statement:
  DECLARE EMPCUR CURSOR
  FOR
     SELECT e.lastname, e.job, d.deptname, d.location, p.projname
     FROM employee AS e, department AS d, project AS p
     WHERE e.workdept = d.deptno AND e.workdept = p.deptno


Estimated Cost        = 118.483406
Estimated Cardinality = 474.720032

        Coordinator Subsection:
(-----)    Distribute Subsection #2
           |  Broadcast to Node List
           |  |  Nodes = 10, 33, 55
(-----)    Distribute Subsection #3
           |  Broadcast to Node List
           |  |  Nodes = 10, 33, 55
(-----)    Distribute Subsection #1
           |  Broadcast to Node List
           |  |  Nodes = 10, 33, 55
(    2)    Access Table Queue  ID = q1  #Columns = 5
(    1)    Return Data to Application
           |  #Columns = 5

        Subsection #1:
```

```
(    8)    Access Table Queue  ID = q2  #Columns = 2
(    3)    Hash Join
           |  Estimated Build Size: 5737
           |  Estimated Probe Size: 8015
(    6)    |  Access Table Queue  ID = q3  #Columns = 3
(    4)    |  Hash Join
           |  |  Estimated Build Size: 5333
           |  |  Estimated Probe Size: 6421
(    5)    |  |  Access Table Name = DOOLE.DEPARTMENT  ID = 2,4
           |  |  |  #Columns = 3
           |  |  |  Relation Scan
           |  |  |  |  Prefetch: Eligible
           |  |  |  Lock Intents
           |  |  |  |  Table: Intent Share
           |  |  |  |  Row  : Next Key Share
(    5)    |  |  |  Process Probe Table for Hash Join
(    2)    Insert Into Asynchronous Table Queue  ID = q1
           |  Broadcast to Coordinator Node
           |  Rows Can Overflow to Temporary Table

        Subsection #2:
(    9)    Access Table Name = DOOLE.PROJECT  ID = 2,7
           |  #Columns = 2
           |  Relation Scan
           |  |  Prefetch: Eligible
           |  Lock Intents
           |  |  Table: Intent Share
           |  |  Row  : Next Key Share
(    9)    |  Insert Into Asynchronous Table Queue  ID = q2
           |  |  Hash to Specific Node
           |  |  Rows Can Overflow to Temporary Tables
(    8)    Insert Into Asynchronous Table Queue Completion  ID = q2

        Subsection #3:
(    7)    Access Table Name = DOOLE.EMPLOYEE  ID = 2,5
           |  #Columns = 3
           |  Relation Scan
           |  |  Prefetch: Eligible
           |  Lock Intents
           |  |  Table: Intent Share
           |  |  Row  : Next Key Share
(    7)    |  Insert Into Asynchronous Table Queue  ID = q3
           |  |  Hash to Specific Node
           |  |  Rows Can Overflow to Temporary Tables
(    6)    Insert Into Asynchronous Table Queue Completion  ID = q3

End of section


Optimizer Plan:

              RETURN
              (   1)
                |
               BTQ
              (   2)
                |
              HSJOIN
              (   3)
             /      \
         HSJOIN      DTQ
         (   4)     (   8)
        /      \      |
    TBSCAN     DTQ   TBSCAN
    (   5)    (   6) (   9)
      |         |      |
   Table:     TBSCAN  Table:
```

```
 DOOLE        (   7)   DOOLE
 DEPARTMENT     |      PROJECT
             Table:
             DOOLE
             EMPLOYEE
```

This plan has all the same pieces as the plan in the first example, but the section has been broken into four subsections. The subsections have the following tasks:

- **Coordinator Subsection**. This subsection coordinates the other subsections. In this plan, it causes the other subsections to be distributed and then uses a table queue to gather the results to be returned to the application.
- **Subsection #1**. This subsection scans table queue q2 and uses a hash join to join it with the data from table queue q3. A second hash join then adds in the data from the DEPARTMENT table. The joined rows are then sent to the coordinator subsection using table queue q1.
- **Subsection #2**. This subsection scans the PROJECT table and hashes to a specific node with the results. These results are read by Subsection #1.
- **Subsection #3**. This subsection scans the EMPLOYEE table and hashes to a specific node with the results. These results are read by Subsection #1.

Example four: multipartition plan with inter-partition and intra-partition parallelism: This example shows the same SQL statement as the first example, but this query has been compiled on a partitioned database made up of three database partitions, each of which is on a four-way SMP machine.

```
******************** PACKAGE ************************************

Package Name = "DOOLE"."EXAMPLE"  Version = ""

 Prep Date = 2002/01/04
 Prep Time = 14:58:35

 Bind Timestamp = 2002-01-04-14.58.35.169555

 Isolation Level          = Cursor Stability
 Blocking                 = Block Unambiguous Cursors
 Query Optimization Class = 5

 Partition Parallel       = Yes
 Intra-Partition Parallel = Yes (Bind Degree = 4)

 SQL Path                 = "SYSIBM", "SYSFUN", "SYSPROC", "DOOLE"

-------------------- SECTION ------------------------------------
Section = 1


SQL Statement:
  DECLARE EMPCUR CURSOR
  FOR
     SELECT e.lastname, e.job, d.deptname, d.location, p.projname
     FROM employee AS e, department AS d, project AS p
     WHERE e.workdept = d.deptno AND e.workdept = p.deptno

Intra-Partition Parallelism Degree = 4

Estimated Cost        = 145.198898
Estimated Cardinality = 474.720032

        Coordinator Subsection:
(-----)    Distribute Subsection #2
           |  Broadcast to Node List
           |  |  Nodes = 10, 33, 55
```

```
(-----)    Distribute Subsection #3
           |  Broadcast to Node List
           |  |  Nodes = 10, 33, 55
(-----)    Distribute Subsection #1
           |  Broadcast to Node List
           |  |  Nodes = 10, 33, 55
(    2)    Access Table Queue  ID = q1  #Columns = 5
(    1)    Return Data to Application
           |  #Columns = 5

        Subsection #1:
(    3)    Process Using 4 Subagents
(   10)    |  Access Table Queue  ID = q3  #Columns = 2
(    4)    |  Hash Join
           |  |  Estimated Build Size: 5737
           |  |  Estimated Probe Size: 8015
(    7)    |  |  Access Table Queue  ID = q5  #Columns = 3
(    5)    |  |  Hash Join
           |  |  |  Estimated Build Size: 5333
           |  |  |  Estimated Probe Size: 6421
(    6)    |  |  |  Access Table Name = DOOLE.DEPARTMENT  ID = 2,4
           |  |  |  |  #Columns = 3
           |  |  |  |  Parallel Scan
           |  |  |  |  Relation Scan
           |  |  |  |  |  Prefetch: Eligible
           |  |  |  |  Lock Intents
           |  |  |  |  |  Table: Intent Share
           |  |  |  |  |  Row  : Next Key Share
(    6)    |  |  |  |  Process Probe Table for Hash Join
(    3)    |  Insert Into Asynchronous Local Table Queue  ID = q2
(    3)    Access Local Table Queue  ID = q2  #Columns = 5
(    2)    Insert Into Asynchronous Table Queue  ID = q1
           |  Broadcast to Coordinator Node
           |  Rows Can Overflow to Temporary Table

        Subsection #2:
(   11)    Process Using 4 Subagents
(   12)    |  Access Table Name = DOOLE.PROJECT  ID = 2,7
           |  |  #Columns = 2
           |  |  Parallel Scan
           |  |  Relation Scan
           |  |  |  Prefetch: Eligible
           |  |  Lock Intents
           |  |  |  Table: Intent Share
           |  |  |  Row  : Next Key Share
(   11)    |  Insert Into Asynchronous Local Table Queue  ID = q4
(   11)    Access Local Table Queue  ID = q4  #Columns = 2
(   10)    Insert Into Asynchronous Table Queue  ID = q3
           |  Hash to Specific Node
           |  Rows Can Overflow to Temporary Tables

        Subsection #3:
(    8)    Process Using 4 Subagents
(    9)    |  Access Table Name = DOOLE.EMPLOYEE  ID = 2,5
           |  |  #Columns = 3
           |  |  Parallel Scan
           |  |  Relation Scan
           |  |  |  Prefetch: Eligible
           |  |  Lock Intents
           |  |  |  Table: Intent Share
           |  |  |  Row  : Next Key Share
(    8)    |  Insert Into Asynchronous Local Table Queue  ID = q6
(    8)    Access Local Table Queue  ID = q6  #Columns = 3
(    7)    Insert Into Asynchronous Table Queue  ID = q5
           |  Hash to Specific Node
           |  Rows Can Overflow to Temporary Tables
```

```
End of section


Optimizer Plan:

              RETURN
              (   1)
                |
               BTQ
              (   2)
                |
               LTQ
              (   3)
                |
              HSJOIN
              (   4)
             /      \
        HSJOIN       DTQ
        (   5)      (  10)
       /      \       |
   TBSCAN     DTQ     LTQ
   (   6)    (   7)  (  11)
     |         |       |
 Table:       LTQ     TBSCAN
 DOOLE       (   8)  (  12)
 DEPARTMENT    |       |
             TBSCAN  Table:
             (   9)  DOOLE
               |     PROJECT
           Table:
           DOOLE
           EMPLOYEE
```

This plan is similar to that in the third example, except that multiple subagents execute each subsection. Also, at the end of each subsection, a local table queue gathers the results from all of the subagents before the qualifying rows are inserted into the second table queue to be hashed to a specific node.

Example five: federated database plan: This example shows the same SQL statement as the first example, but this query has been compiled on a federated database where the tables DEPARTMENT and PROJECT are on a data source and the table EMPLOYEE is on the federated server.

```
******************** PACKAGE ************************************

Package Name = "DOOLE"."EXAMPLE"  Version = ""

 Prep Date = 2002/01/11
 Prep Time = 13:52:48

 Bind Timestamp = 2002-01-11-13.52.48.325413

 Isolation Level          = Cursor Stability
 Blocking                 = Block Unambiguous Cursors
 Query Optimization Class = 5

 Partition Parallel       = No
 Intra-Partition Parallel = No

 SQL Path                 = "SYSIBM", "SYSFUN", "SYSPROC", "DOOLE"

-------------------- SECTION ---------------------------------------
Section = 1


SQL Statement:
```

```
  DECLARE EMPCUR CURSOR
  FOR
     SELECT e.lastname, e.job, d.deptname, d.location, p.projname
     FROM employee AS e, department AS d, project AS p
     WHERE e.workdept = d.deptno AND e.workdept = p.deptno


Estimated Cost        = 1804.625000
Estimated Cardinality = 112000.000000

(    7) Ship Distributed Subquery #2
        |  #Columns = 2
(    2) Hash Join
        |  Estimated Build Size: 48444
        |  Estimated Probe Size: 232571
(    6) |  Access Table Name = DOOLE.EMPLOYEE  ID = 2,5
        |  |  #Columns = 3
        |  |  Relation Scan
        |  |  |  Prefetch: Eligible
        |  |  Lock Intents
        |  |  |  Table: Intent Share
        |  |  |  Row  : Next Key Share
(    6) |  |  Process Build Table for Hash Join
(    3) |  Hash Join
        |  |  Estimated Build Size: 7111
        |  |  Estimated Probe Size: 64606
(    4) |  |  Ship Distributed Subquery #1
        |  |  |  #Columns = 3
(    1) Return Data to Application
        |  #Columns = 5

Distributed Substatement #1:
(    4) Server: REMOTE  (DB2/UDB 8.1)
        SQL Statement:

             SELECT A0."DEPTNO", A0."DEPTNAME", A0."LOCATION"
             FROM "DOOLE"."DEPARTMENT" A0

        Nicknames Referenced:
           DOOLE.DEPARTMENT  ID = 32768
              Base = DOOLE.DEPARTMENT
        #Output Columns = 3

Distributed Substatement #2:
(    7) Server: REMOTE  (DB2/UDB 8.1)
        SQL Statement:

             SELECT A0."DEPTNO", A0."PROJNAME"
             FROM "DOOLE"."PROJECT" A0

        Nicknames Referenced:
           DOOLE.PROJECT  ID = 32769
              Base = DOOLE.PROJECT
        #Output Columns = 2

End of section

Optimizer Plan:

               RETURN
               (   1)
                 |
               HSJOIN
               (   2)
              /      \
        HSJOIN          SHIP
        (   3)         (   7)
```

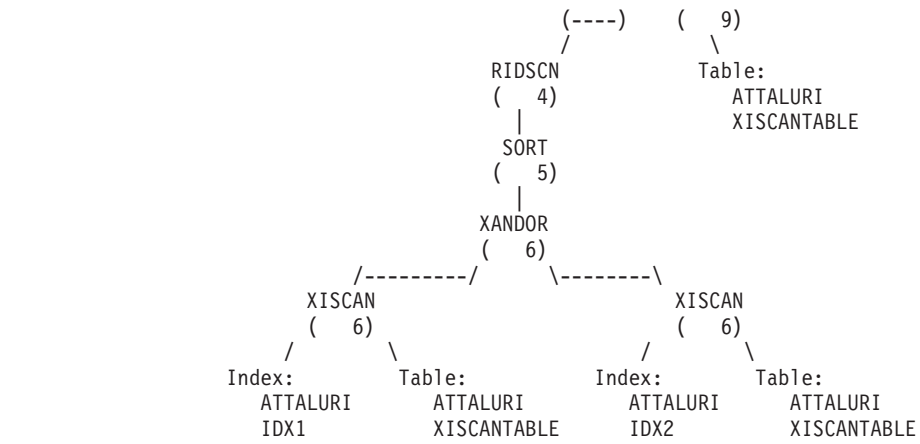
```
          /      \          |
       SHIP      TBSCAN   Nickname:
      (   4)     (   6)   DOOLE
        |          |      PROJECT
   Nickname:    Table:
   DOOLE        DOOLE
   DEPARTMENT   EMPLOYEE
```

This plan has all the same pieces as the plan in the first example, except that the data for two of the tables are coming from data sources. The two tables are accessed through distributed subqueries which, in this case, simply select all the rows from those tables. Once the data is returned to the federated server, it is joined to the data from the local table.

Example six: XANDOR and XISCAN operators: This example shows how the XANDOR operator combines XISCAN scans of two individual indexes over XML data (*IDX1* and *IDX2*) that have been defined on the same table *XISCANTABLE*.

```
IBM DB2 Database SQL Explain Tool

****************** DYNAMIC ***************************************

================== STATEMENT ========================================

        Isolation Level          = Cursor Stability
        Blocking                 = Block Unambiguous Cursors
        Query Optimization Class = 5

        Partition Parallel       = No
        Intra-Partition Parallel = No

        SQL Path                 = "SYSIBM", "SYSFUN", "SYSPROC", "ATTALURI"


Query Statement:
  xquery
  for $c in db2-fn:xmlcolumn("XISCANTABLE.XMLCOL ")/a[@x="1" ]/b[@y=
          "2" ] return $c

Section Code Page = 819

Estimated Cost = 192.266113
Estimated Cardinality = 1.800000

(    6) Index ANDing and ORing over XML
         |  Xpath is
         |  |  /child::element(a)[./child::element(b)
         |  |  /attribute::attribute(y)(:Index Search over XML 1:)
         |  |   and ./attribute::attribute(x)(:Index Search over XML 2:)
         |  |  ]
         |  Index Search over XML 1
         |  |  Access Table Name = ATTALURI.XISCANTABLE
         |  |  |  Index Scan over XML:  Name = ATTALURI.IDX1  ID = 6
         |  |  |  |  Physical Index over XML
         |  |  |  |  Index Columns:
         |  |  |  |  |  1: XMLCOL (Ascending)
         |  |  |  #Key Columns = 4
         |  |  |  |  Start Key: Inclusive Value
         |  |  |  |  |  |  1: ?
         |  |  |  |  |  |  2: ?
         |  |  |  |  |  |  3: ?
         |  |  |  |  |  |  4: ?
         |  |  |  |  Stop Key: Inclusive Value
         |  |  |  |  |  |  1: ?
         |  |  |  |  |  |  2: ?
         |  |  |  Index-Only Access
         |  |  |  Index Prefetch: None
         |  |  |  Isolation Level: Uncommitted Read
```

```
        |  |  |  Lock Intents
        |  |  |  |  Table: Intent None
        |  |  |  |  Row  : None
        |  |  |  StopKey = StartKey
        |  |  |  Value Start Key = ?
        |  |  Index Search over XML 2
        |  |  |  Access Table Name = ATTALURI.XISCANTABLE
        |  |  |  |  Index Scan over XML:  Name = ATTALURI.IDX2  ID = 4
        |  |  |  |  Physical Index over XML
        |  |  |  |  Index Columns:
        |  |  |  |  |  1: XMLCOL (Ascending)
        |  |  |  |  #Key Columns = 4
        |  |  |  |  Start Key: Inclusive Value
        |  |  |  |  |  |  1: ?
        |  |  |  |  |  |  2: ?
        |  |  |  |  |  |  3: ?
        |  |  |  |  |  |  4: ?
        |  |  |  |  Stop Key: Inclusive Value
        |  |  |  |  |  |  1: ?
        |  |  |  |  |  |  2: ?
        |  |  |  |  Index-Only Access
        |  |  |  |  Index Prefetch: None
        |  |  |  |  Isolation Level: Uncommitted Read
        |  |  |  |  Lock Intents
        |  |  |  |  |  Table: Intent None
        |  |  |  |  |  Row  : None
        |  |  |  |  StopKey = StartKey
        |  |  |  |  Value Start Key = ?
(    5) Insert Into Sorted Temp Table  ID = t1
        |  #Columns = 1
        |  #Sort Key Columns = 1
        |  |  Key 1: (Ascending)
        |  Sortheap Allocation Parameters:
        |  |  #Rows     = 2
        |  |  Row Width = 16
        |  Piped
        |  Duplicate Elimination
(    4) List Prefetch Preparation
(    4) |  Access Table Name = ATTALURI.XISCANTABLE  ID = 2,16
        |  |  #Columns = 1
        |  |  Fetch Using Prefetched List
        |  |  |  Prefetch: Eligible
        |  |  Lock Intents
        |  |  |  Table: Intent Share
        |  |  |  Row  : Next Key Share
(    2) Nested Loop Join
        |  Piped Inner
(    9) |  XML Doc Navigation
        |  |  Navigator is
        |  |  |  /fn:root($CONTEXT_NODE$())/child::element(a)(:#Xpath Predicates = 1:)
        |  |  |  [./child::element(b)(:Output nodeSeqRef :)
        |  |  |  (:#Xpath Predicates = 1:)
        |  |  |  /attribute::attribute(y) and
        |  |  |  ./attribute::attribute(x)]
(    1) Iterate over XML sequence for Xquery bindout
(    1) Return Data to Application
        |  #Columns = 1

End of section


Optimizer Plan:

                                    RETURN
                                    (   1)
                                      |
                                    NLJOIN
                                    (   2)
                                   /      \
                                FETCH    XSCAN
```



Example seven: XSCAN operator: This example shows how the XSCAN operator may appear in an access plan. This operator processes node references passed by a nested-loop join operator (NLJOIN). It is not represented with a direct input in the access plan.

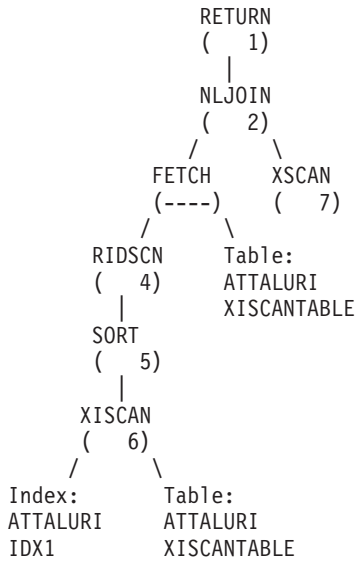
```
IBM DB2 Database SQL Explain Tool

******************** DYNAMIC ***************************************

=================== STATEMENT ============================================

        Isolation Level          = Cursor Stability
        Blocking                 = Block Unambiguous Cursors
        Query Optimization Class = 5

        Partition Parallel       = No
        Intra-Partition Parallel = No

        SQL Path                 = "SYSIBM", "SYSFUN", "SYSPROC", "ATTALURI"


Query Statement:
  xquery
  for $b in db2-fn:xmlcolumn("XISCANTABLE.XMLCOL" )//book[position()<=
          2] return $b

Section Code Page = 819

Estimated Cost = 779592.625000
Estimated Cardinality = 540000000.000000

(    4) Access Table Name = ATTALURI.XISCANTABLE  ID = 2,16
         |  #Columns = 1
         |  Relation Scan
         |  |  Prefetch: Eligible
         |  Lock Intents
         |  |  Table: Intent Share
         |  |  Row  : Next Key Share
(    3) Nested Loop Join
         |  Piped Inner
(    6)  |  XML Doc Navigation
         |  |  Navigator is
         |  |  |  /fn:root($CONTEXT_NODE$())/descendant-or-self::node()(:Output nodeSeqRef :)
         |  |  |
(    5)  |  Nested Loop Join
         |  |  Piped Inner
(   11)  |  |  XML Doc Navigation
         |  |  |  Navigator is
         |  |  |  |  /fn:root($CONTEXT_NODE$())/child::element(book)(:Output nodeSeqRef :)
```

```
        |  |  |  |
(   10) |  |  Aggregation
        |  |  |  Column Function(s)
(    9) |  |  Nested Loop Join
        |  |  |  Piped Inner
(   12) |  |  |  Unnest input XML sequence into stream of items with item number
(    8) |  |  Nested Loop Join
        |  |  |  Piped Inner
(   14) |  |  |  Table Constructor
        |  |  |  |  1-Row(s)
(    2) Nested Loop Join
        |  Piped Inner
(   16) |  Unnest input XML sequence into stream of items
(    1) Iterate over XML sequence for Xquery bindout
(    1) Return Data to Application
        |  #Columns = 1

End of section

Optimizer Plan:

               RETURN
               (   1)
                 |
               NLJOIN
               (   2)
             /-/      \-----\
          NLJOIN            *
          (   3)            *
        /       \           |
    TBSCAN      NLJOIN      TBSCAN
    (   4)      (   5)      (  16)
     |          /     \       |
   Table:     XSCAN   *      TFunc:
   ATTALURI   (   6)  *      SYSIBM
   XISCANTABLE        |      GENROW
                    NLJOIN
                    (   8)
                   /      \
               NLJOIN     FILTER
              (   9)      (  13)
             /      \       |
          GRPBY   TBSCAN  TBSCAN
          (  10)  (  12)  (  14)
            |       |       |
          XSCAN   TFunc:  TFunc:
          (  11)  SYSIBM  SYSIBM
                  GENROW  GENROW
```

Example eight: XISCAN operator: This example shows how the XISCAN operator scans the index over XML data *IDX1* defined on the table *XISCANTABLE*.

```
IBM DB2 Database SQL Explain Tool

****************** DYNAMIC ***************************************

=================== STATEMENT ============================================

        Isolation Level          = Cursor Stability
        Blocking                 = Block Unambiguous Cursors
        Query Optimization Class = 5

        Partition Parallel       = No
        Intra-Partition Parallel = No

        SQL Path                 = "SYSIBM", "SYSFUN", "SYSPROC", "ATTALURI"
```

```
Query Statement:
  xquery
  for $c in db2-fn:xmlcolumn("XISCANTABLE.XMLCOL ")/a[@x="1" ] return
          $c


Section Code Page = 819

Estimated Cost = 1666.833862
Estimated Cardinality = 18.000000

(    6) Access Table Name = ATTALURI.XISCANTABLE
         |  Index Scan over XML:  Name = ATTALURI.IDX1  ID = 4
         |  |  Physical Index over XML
         |  |  Index Columns:
         |  |  |  1: XMLCOL (Ascending)
         |  #Key Columns = 2
         |  |  Start Key: Inclusive Value
         |  |  |  |  1: ?
         |  |  |  |  2: ?
         |  |  Stop Key: Inclusive Value
         |  |  |  |  1: ?
         |  |  |  |  2: ?
         |  Index-Only Access
         |  Index Prefetch: None
         |  Isolation Level: Uncommitted Read
         |  Lock Intents
         |  |  Table: Intent None
         |  |  Row  : None
         |  StopKey = StartKey
         |  Value Start Key = ?
         |  Xpath is
         |  |  /child::element(a)/attribute::attribute(x)
(    5) Insert Into Sorted Temp Table  ID = t1
         |  #Columns = 1
         |  #Sort Key Columns = 1
         |  |  Key 1: (Ascending)
         |  Sortheap Allocation Parameters:
         |  |  #Rows     = 18
         |  |  Row Width = 16
         |  Piped
         |  Duplicate Elimination
(    4) List Prefetch Preparation
(    4) |  Access Table Name = ATTALURI.XISCANTABLE  ID = 2,16
         |  |  #Columns = 1
         |  |  Fetch Using Prefetched List
         |  |  |  Prefetch: Eligible
         |  |  Lock Intents
         |  |  |  Table: Intent Share
         |  |  |  Row  : Next Key Share
(    2) Nested Loop Join
         |  Piped Inner
(    7)  |  XML Doc Navigation
         |  |  Navigator is
         |  |  |  /fn:root($CONTEXT_NODE$())/child::element(a)(:Output nodeSeqRef :)
         |  |  |  (:#Xpath Predicates = 1:)
         |  |  |  /attribute::attribute(x)
(    1) Iterate over XML sequence for Xquery bindout
(    1) Return Data to Application
         |  #Columns = 1

End of section


Optimizer Plan:
```



The explain tables and organization of explain information

All explain information is organized around the concept of an explain instance. An explain instance represents one invocation of the explain facility for one or more SQL or XQuery statements. The explain information captured in one explain instance includes the compilation environment as well as the access plan chosen to satisfy the SQL or XQuery statement being compiled. For example, an explain instance might consist of any one of the following:

- 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, SELECT, SET INTEGRITY, SELECT INTO, UPDATE, VALUES, and VALUES INTO statements. For 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 not compiled statically, but only dynamically.
- One particular SQL statement for incremental bind SQL statements
- One particular SQL statement for dynamic SQL statements
- Each EXPLAIN SQL statement (whether dynamic or static)

Explain table information reflects the relationships between operators and data objects in the access plan. The following diagram shows the relationships between these tables.

Explain information is stored in the following tables:

Table 65. Relational tables that store explain data

Table Name	Description
EXPLAIN_ARGUMENT	Contains information about the unique characteristics for each individual operator, if any.
EXPLAIN_INSTANCE	The main control table for all Explain information. Each row of data in the Explain tables is explicitly linked to one unique row in this table. Basic information about the source of the SQL or XQuery statements being explained and environment information is kept in this table.

Table 65. Relational tables that store explain data (continued)

Table Name	Description
EXPLAIN_OBJECT	Identifies those data objects required by the access plan generated to satisfy the SQL or XQuery statement.
EXPLAIN_OPERATOR	Contains all the operators needed to satisfy the SQL or XQuery statement by the query compiler.
EXPLAIN_PREDICATE	Identifies the predicates that are applied by a specific operator.
EXPLAIN_STATEMENT	Contains the text of the SQL or XQuery statement as it exists for the different levels of explain information. The original SQL or XQuery statement as entered by the user is stored in this table with the version used by the optimizer to choose an access plan. When an explain snapshot is requested, additional explain information is recorded to describe the access plan selected by the query optimizer. This information is stored in the SNAPSHOT column of the EXPLAIN_STATEMENT table in the format required by Visual Explain. This format is not usable by other applications.
EXPLAIN_STREAM	Represents the input and output data streams between individual operators and data objects. The data objects themselves are represented in the EXPLAIN_OBJECT table. The operators involved in a data stream are represented in the EXPLAIN_OPERATOR table.
EXPLAIN_DIAGNOSTIC	Contains an entry for each diagnostic message produced for a particular instance of an explained statement in the EXPLAIN_STATEMENT table.
EXPLAIN_DIAGNOSTIC_DATA	Contains message tokens for specific 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.
ADVISE_WORKLOAD	Allows users to describe a workload to the database. Each row in the table represents an SQL or XQuery statement in the workload and is described by an associated frequency. The `db2advis` tool uses this table to collect and store workload information.
ADVISE_INSTANCE	Contains information about `db2advis` execution, including start time. Contains one row for each execution of `db2advis`.
ADVISE_INDEX	Stores information about recommended indexes. The table can be populated by the query compiler, the `db2advis` utility or a user. This table is used in two ways: • To get recommended indexes. • To evaluate indexes based on input about proposed indexes.
ADVISE_MQT	Contains the CREATE DDL, the query defining each recommended MQT, the statistics for each MQT such as COLSTATS (per column) in XML form, NUMROWS, and so on,, as well as the sampling query to obtain sampled statistics for each MQT.
ADVISE_TABLE	Stores the DDL for table creation using the final Design Advisor recommendations for recommended MQTs, MDCs and database partitions, depending on the options specified and the recommendations generated.
ADVISE_PARTITION	Stores virtual database partitions generated and evaluated by `db2advis`.

Note: Not all of the tables above are created by default. To create them, run the EXPLAIN.DDL script found in the *misc* subdirectory of the *sqllib* subdirectory.

In DB2 Version 9.5, you now have the ability to create, drop and validate explain tables using the SYSPROC.SYSINSTALLOBJECTS procedure. This procedure allows

the explain tables to be created under a specific schema and tablespace. An example can be found in the EXPLAIN.DLL file.

Explain tables might be common to more than one user. However, the explain tables can be defined for one user, and then aliases can be defined for each additional user using the same name to point to the defined tables. Alternatively, the explain tables can be defined under the SYSTOOLS schema. The Explain facility will default to the SYSTOOLS schema if no other explain tables or aliases are found under the user's session ID for dynamic SQL or XQuery statements, or the statement authorization ID for static SQL or XQuery statements. Each user sharing the common explain tables must have insert permission on those tables. Read permission for the common explain tables should also be limited, typically to users who analyze the explain information.

Explain information for data objects

A single access plan may use one or more data objects to satisfy the SQL or XQuery statement.

Object Statistics: The explain facility records information about the object, such as the following:

- The creation time
- The last time that statistics were collected for the object
- An indication of whether or not the data in the object is ordered (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 this object is stored
- The estimated transfer rate, in milliseconds, to read a 4K page from the specified table space
- Prefetch and extent sizes, in 4K pages
- The degree of data clustering with the index
- The number of leaf pages 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 required, such as a table scan, an index scan, a nested loop join, or a group-by operator.

In addition to showing the operators used in an access plan and information about each operator, explain information also shows the cumulative effects of the access plan.

Estimated Cost Information: The following estimated cumulative costs can be displayed for the operators. 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 CPU instructions
- The cost (in timerons) of fetching the first row, including any initial overhead required
- The communication cost (in frames).

Timerons are an invented relative unit of measure. Timerons are determined by the optimizer based on internal values such as statistics that change as the database is used. As a result, the timerons measure for a SQL or XQuery statement are not guaranteed to be the same every time the estimated cost in timerons is determined.

Operator Properties: The following information is recorded by the explain facility to describe the properties of each operator:

- 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 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 information provided by the following items helps you to uniquely identify each explain instance and to correlate the information for the query statements with a given invocation of the facility:

- The user who requested the explain information
- When the explain request began
- The name of the package that contains the explained query statement
- The SQL schema of the package that contains the explained query 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 for the level of DB2
- The degree of parallelism for which the query was compiled

  The CURRENT DEGREE special register, the DEGREE bind option, the SET RUNTIME DEGREE API, and the *dft_degree* configuration parameter determine the degree of parallelism for which a particular query is compiled.
- Whether the query statement is dynamic or static
- The query optimization class used to compile the query
- The type of row blocking for cursors specified when compiling the query
- The isolation level in which the query runs
- The values of various configuration parameters when the query was compiled. 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 query statement might have been explained for each explain instance. In addition to information that uniquely identifies the explain instance, the following information helps identify individual query statements:

- The type of statement: SELECT, DELETE, INSERT, UPDATE, positioned DELETE, positioned UPDATE, SET INTEGRITY
- The statement and section number of the package issuing the query statement, as recorded in 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. For dynamic explain query statements submitted during a CLP or CLI session, when EXPLAIN MODE or EXPLAIN SNAPSHOT is active, the QUERYTAG is set to "CLP" or "CLI". 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 query statements, which are not from CLP, CLI, or do not use the EXPLAIN query statement, QUERYTAG 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 text of the query statement are recorded for each statement explained. 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 allow you to understand the context in which the SQL and XQuery optimizer chose the access plan. To understand how the SQL and XQuery 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 elements in the environment affecting your statement, such as triggers and constraints. Some keywords used by this "optimized" text are:

$C*n* The name of a derived column, where *n* represents an integer value.

$CONSTRAINT$
The tag used to indicate the name of a constraint added to the original query statement during compilation. Seen in conjunction with the $WITH_CONTEXT$ prefix.

$DERIVED.T*n*
The name of a derived table, where *n* represents an integer value.

$INTERNAL_FUNC$
The tag indicates the presence of a function used by the SQL and XQuery compiler for the explained query but not available for general use.

$INTERNAL_PRED$
The tag indicates the presence of a predicate added by the SQL and XQuery compiler during compilation of the explained query but not available for general use. An internal predicate is used by the compiler to satisfy additional context added to the original query statement because of triggers and constraints.

$INTERNAL_XPATH$
Shows an internal table function which takes a single input annotated XPath pattern as a parameter and returns a table with one or more columns that match the pattern.

RID The tag used to identify the row identifier (RID) column for a particular row.

$TRIGGER$
The tag used to indicate the name of a trigger added to the original query statement during compilation. Seen in conjunction with the $WITH_CONTEXT$ prefix.

$WITH_CONTEXT$(...)
This prefix appears at the start of the text when additional triggers or constraints have been added into the original query statement. A list of the names of any triggers or constraints affecting the compilation and resolution of the query statement appears after this prefix.

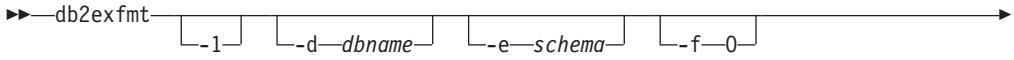
db2exfmt - Explain table format

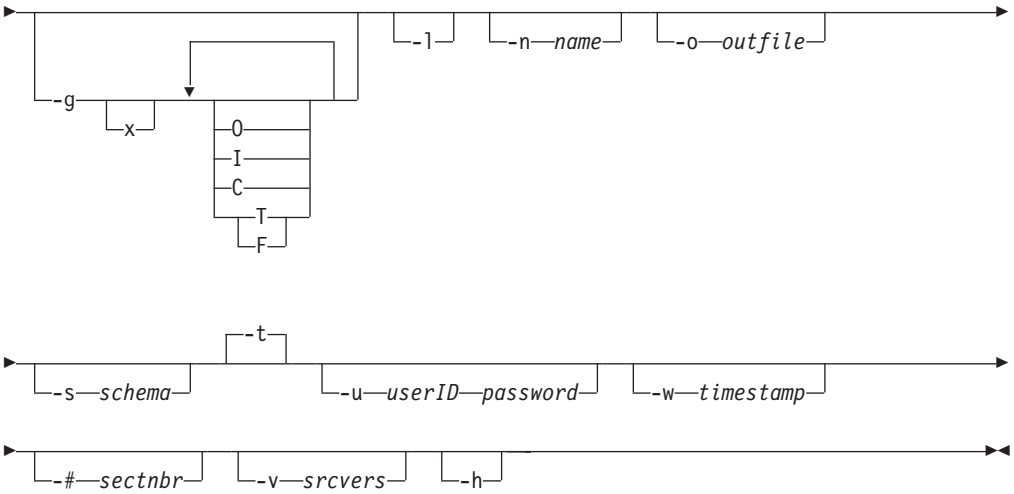
You use the db2exfmt tool to format the contents of the EXPLAIN tables. This tool is located in the `misc` subdirectory of the instance `sqllib` directory. This tool uses the statistics from the EXPLAIN snapshot, if the snapshot is available.

Authorization

To use the tool, you require read access to the explain tables being formatted.

Command syntax





Command parameters

db2exfmt
If no options are specified, then the command enters interactive mode and you will be prompted to make entries.

-1 Use defaults `-e % -n % -s % -v % -w -1 -# 0`

If Explain schema is not supplied, the contents of the environment variable $USER, or $USERNAME will be used as a default. If this variable is not found, the user will be prompted for an Explain schema.

-d *dbname*
Name of the database containing packages.

-e *schema*
Explain table SQL schema.

-f Formatting flags. In this release, the only supported value is O (operator summary).

-g Graph plan.

x Turn OFF options (default is to turn them ON).

If only -g is specified, a graph, followed by formatted information for all of the tables, is generated. Otherwise, any combination of the following valid values can be specified:

O Generate a graph only. Do not format the table contents.

T Include total cost under each operator in the graph.

F Include first tuple cost in graph.

I Include I/O cost under each operator in the graph.

C Include the expected output cardinality (number of tuples) of each operator in the graph.

Any combination of these options is allowed, except F and T, which are mutually exclusive.

-l Respect case when processing package names.

-n *name*
Name of the source of the explain request (SOURCE_NAME).

-s *schema*
SQL schema or qualifier of the source of the explain request (SOURCE_SCHEMA).

-o *outfile*
Output file name.

-t Direct the output to the terminal.

-u *userID password*
When connecting to a database, use the provided user ID and password.

Both the user ID and password must be valid according to naming conventions and be recognized by the database.

-w *timestamp*
Explain time stamp. Specify `-1` to obtain the latest explain request.

-# *sectnbr*
Section number in the source. To request all sections, specify zero.

-v *srcvers*
Source version of source of Explain request (default %)

-h Display help information. When this option is specified, all other options are ignored, and only the help information is displayed.

Usage notes

You will be prompted for any parameter values that are not supplied, or that are incompletely specified, except in the case of the -h and the `-l` options.

If an explain table SQL schema is not provided, the value of the environment variable USER is used as the default. If this variable is not found, the user is prompted for an explain table SQL schema.

Source name, source SQL schema, and explain time stamp can be supplied in LIKE predicate form, which allows the percent sign (%) and the underscore (_) to be used as pattern matching characters to select multiple sources with one invocation. For the latest explained statement, the explain time can be specified as `-1`.

If -o is specified without a file name, and -t is not specified, the user is prompted for a file name (the default name is `db2exfmt.out`). If neither -o nor -t is specified, the user is prompted for a file name (the default option is terminal output). If -o and -t are both specified, the output is directed to the terminal.

The db2exfmt command displays the statistics from the EXPLAIN snapshot, if the snapshot is available. Otherwise, db2exfmt displays statistics stored in the EXPLAIN_OBJECT table and also displays some statistics retrieved directly from the system catalog.

The following are EXPLAIN snapshot examples.

```
db2 explain plan with snapshot for query
db2exfmt
```

or,

```
db2 set current explain mode yes
db2 set current explain snapshot yes
run the query
db2exfmt
```

Optimizing query access plans

Optimization classes

When you compile an SQL or XQuery query, you can specify an optimization class that determines how the optimizer chooses the most efficient access plan for that query. The optimization classes are differentiated by the number and type of optimization strategies considered in the compilation of the query. Although you can specify optimization techniques individually to improve runtime performance for the query, the more optimization techniques you specify, the more time and system resources query compilation will require.

You can specify one of the following optimizer classes when you compile an SQL or XQuery query:

0 - This class directs the optimizer to use minimal optimization to generate an access plan. This optimization class has the following characteristics:

- Non-uniform distribution statistics are not considered by the optimizer.
- Only basic query rewrite rules are applied.
- Greedy join enumeration occurs.
- 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 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:

- Non-uniform distribution statistics are not considered by the optimizer.
- Only a subset of the query rewrite rules are applied.
- Greedy join enumeration occurs.
- 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 significantly higher than class 1, while keeping the compilation cost significantly lower than classes 3 and above for complex queries. This optimization class has the following characteristics:

- All available statistics, including both frequency and quantile non-uniform distribution statistics, are used.
- All query rewrite rules are applied, including routing queries to materialized query tables, except computationally intensive rules that are applicable only in very rare cases.
- Greedy join enumeration is used.
- A wide range of access methods are 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. 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 classes 3 and above. Class 2 is recommended for very complex queries in a decision support or online analytic processing (OLAP) environment. In such environments, specific queries are rarely repeated exactly, so that a query access plan is unlikely to remain in the cache until the next occurrence of the query.

3 - This class requests a moderate amount of optimization. This class comes closest to matching the query optimization characteristics of DB2 for MVS/ESA™, OS/390, or z/OS. This optimization class has the following characteristics:

- Non-uniform distribution statistics, which track frequently occurring values, are used if available.
- Most query rewrite rules are applied, including subquery-to-join transformations.
- Dynamic programming join enumeration, as follows:
  - Limited use of composite inner tables
  - Limited use of Cartesian products for star schemas involving look-up tables
- A wide range of access methods are considered, including list prefetch, index ANDing, and star joins.

This class is suitable for a broad range of applications. This class improves access plans for queries with four or more joins. However, the optimizer might fail to consider a better plan that might be chosen with the default optimization class.

5 - This class directs the optimizer to use a significant amount of optimization to generate an access plan. This optimization class has the following characteristics:

- All available statistics are used, including both frequency and quantile distribution statistics.
- All of the query rewrite rules are applied, including the routing of queries to materialized query tables, except for those computationally intensive rules which are applicable only in very rare cases.
- Dynamic programming join enumeration, as follows:
  - Limited use of composite inner tables
  - Limited use of Cartesian products for star schemas involving look-up tables
- A wide range of access methods are considered, including list prefetch, index ANDing, and materialized query table routing.

When the optimizer detects that the additional resources and processing time are not warranted for complex dynamic SQL or XQuery queries, optimization is reduced. The extent or size of the reduction depends on the machine size and the number of predicates.

When the query optimizer reduces the amount of query optimization, it continues to apply all the query rewrite rules that would normally be applied. However, it does use the Greedy join enumeration method and reduces the number of access plan combinations that are considered.

Query optimization class 5 is an excellent choice for a mixed environment consisting of both transactions and complex queries. This optimization class is designed to apply the most valuable query transformations and other query optimization techniques in an efficient manner.

7 - This class directs the optimizer to use a significant amount of optimization to generate an access plan. It is the same as query optimization class 5 except that it does not reduce the amount of query optimization for complex dynamic SQL or XQuery queries.

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 enumerations, including Cartesian products and unlimited composite inners
- All access methods

This class can greatly expand the number of possible access plans that are considered by the optimizer. You might use this class to find out whether more comprehensive optimization would generate a better access plan for very complex and 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, particularly for the following reasons:

- To manage very small databases or very simple dynamic queries
- To accommodate memory limitations at compile time on your database server
- To reduce the query compilation time, such as PREPARE.

Most statements can be adequately optimized with a reasonable amount of resources by using optimization class 5, which is the default query optimization class. At a given optimization class, the query compilation time and resource consumption is primarily influenced by the complexity of the query, particularly the number of joins and subqueries. However, compilation time and resource usage are also affected by the amount of optimization performed.

Query optimization classes 1, 2, 3, 5, and 7 are all suitable for general-purpose use. Consider class 0 only if you require further reductions in query compilation time and you know that the SQL and XQuery statements are extremely simple.

Tip: To analyze queries that run a long time, run the query with db2batch to find out how much time is spent in compilation and how much is spent in execution.If compilation requires more time, reduce the optimization class. If execution requires more time, consider a higher optimization class.

When you select an optimization class, consider the following general guidelines:

- Start by using the default query optimization class, class 5.
- To use 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 of the join predicates that are on the same column, and if compilation time is a concern.

- Use a low optimization class (0 or 1) for queries having very short run-times of less than one second. Such queries tend to have the following characteristics:
  - Access to a single or only a few tables
  - Fetch a single or only a few rows
  - Use fully qualified, unique indexes.

Online transaction processing (OLTP) transactions are good examples of this kind of query.

- Use a higher optimization class (3, 5, or 7) for longer running queries that take more than 30 seconds.
- Classes 3 and above use the Dynamic Programming join enumeration algorithm. This algorithm considers many more alternative plans, and might incur significantly more compilation time than classes 0, 1, and 2, especially as the number of tables increases.
- Use optimization class 9 only if you have specific 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 the following characteristics:

- Access to large tables
- A large number of predicates
- Many subqueries
- Many joins
- Many set operators, such as UNION and INTERSECT
- Many qualifying rows
- GROUP BY and HAVING operations
- Nested table expressions
- A large number of views.

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, including those produced by a query generator, require additional optimization to select a good access plan. Using query optimization class 2 and higher can improve such SQL and XQuery queries.

Note: In a federated database query, 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. For static SQL and XQuery statements, the query compilation time and resources are expended just 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. 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 think that a query that might benefit from additional optimization, but you are not sure, or you are concerned about compilation time and resource usage, you might perform some benchmark testing.

To specify a query optimization class, follow these steps:

1. Analyze the performance factors either informally or with formal tests as follows:
   - For **dynamic** query statements, tests should compare the average run time for the statement. Use the following formula to estimate an average run time:

     ```
     compile time + sum of execution times for all iterations
     -------------------------------------------------------
                    number of iterations
     ```

     In this formula, the `number of iterations` represents the number of times that you expect that the query statement might be executed each time it is compiled.

     Note: After the initial compilation, dynamic SQL and XQuery statements are recompiled when a change to the environment requires it. If the environment does not change after a query statement is cached, it does not need to be compiled again because subsequent PREPARE statements re-use the cached statement.
   - For **static** SQL and XQuery statements, compare the statement run times.

     Although you might also be interested in the compile time of static SQL and XQuery statements, the total compile and run time for the statement is difficult to assess in any meaningful context. Comparing the total time does not recognize the fact that a static query statement can be run many times for each time it is bound and that it is generally not bound during run time.
2. Specify the optimization class as follows:
   - **Dynamic** SQL and XQuery statements use the optimization class specified by the CURRENT QUERY OPTIMIZATION special register that you set with the SQL statement SET. 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, you might include a SET statement in the application program.

     If the CURRENT QUERY OPTIMIZATION 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 value of the database configuration parameter *dft_queryopt*. Class 5 is the default value of this parameter. 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 specified on the PREP and BIND commands. The QUERYOPT column in the SYSCAT.PACKAGES catalog table records the optimization class used to bind the package. If the package is rebound either implicitly or using the REBIND PACKAGE command, this same optimization class is used for the static query 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, DB2 uses the default optimization as specified by *dft_queryopt* database configuration parameter.

Optimizer profiles and guidelines overview

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 facilities such as db2advis, RUNSTATS, db2expln and the optimization class setting to help you tune the optimizer for better database performance. If you do not receive the desired 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 relevant information needed to unambiguously identify an SQL statement. Figure 31 illustrates 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 TPCD Q9">
  <STMTKEY SCHEMA="TPCD">
    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>
```

Figure 31. Passing guidelines using an optimization profile

Each STMTPROFILE element provides a set of optimization guidelines for one application statement. The targeted statement is identified by the STMTKEY sub-element. The optimization profile is then given a schema qualified name and inserted into the database. The optimization profile is put into effect for the optimization of the statement by specifying this name on the BIND or PREPARE 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 in the database, and provide the name of the optimization profile on the BIND or PREPARE 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 works with the selection of other possible access plans using the existing cost-based methods. Optimization guidelines targeting specific table references cannot override general optimization parameter settings. Thus, the optimization guideline specifying the merge join between tables A and B is not valid at optimization class 0.

The optimizer ignores invalid or inapplicable optimization guidelines. If any optimization guidelines are ignored, an execution plan is produced and a SQL0437W warning with reason code 13 is returned. You can then use the EXPLAIN statement to get detailed diagnostic information regarding optimization guidelines processing.

Optimization guidelines

Types of optimization guidelines and processing overview: The DB2 optimizer optimizes a statement in two phases. 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 considered during the two optimization phases are governed by various DB2 parameters, such as CURRENT QUERY OPTIMIZATION (a special register), REOPT (a bind option), and *DB2_REDUCED_OPTIMIZATION* (a registry variable). The set of optimization alternatives considered during the optimization of a statement is called the search space.

The following types of statement optimization guidelines are supported:

- General optimization guidelines
- Query rewrite guidelines
- Plan optimization guidelines

Optimization guidelines are applied in a specific order. General optimization guidelines are applied first, since they can affect the search space. Query rewrite guidelines are applied next, since they can affect the statement optimized during the plan optimization phase. Plan optimization guidelines are applied last.

General optimization guidelines can be used to affect the setting of general optimization parameters. *Query rewrite guidelines* can be used to affect the transformations considered during the query rewrite optimization phase. *Plan optimization guidelines* can be used to affect the access methods, join methods, and join orders considered during the plan optimization phase.

General optimization guidelines: General optimization guidelines can be used to set general optimization parameters. Each of these guidelines have statement level scope.

Query rewrite optimization guidelines: Query rewrite guidelines can be used to affect the transformations considered during the query rewrite optimization phase. The query rewrite optimization phase transforms the original statement into a semantically equivalent *optimized statement*. The optimal execution plan for the optimized statement is then 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 with 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, some query rewrite rules enforced before a subsequent rule can affect the query rewrite optimization guideline associated with that rule. Sometimes the environment configuration can affect the behavior of some rewrite rules, which will affect the applicability of the query rewrite optimization guideline for a specific rule. In order to get the same result each time, query rewrite rules have some conditions before being enforced. If the conditions associated with the 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 will be ignored. If the query rewrite optimization guideline is not applicable and the guideline is an enabling guideline, an SQL0437W error message with reason code 13 is returned. If the query rewrite optimization guideline is not applicable and the guideline is a disabling guideline, no error message is returned. The query rewrite rule is not applied in this case because the rule is treated as if it was disabled.

The query rewrite optimization guidelines can be divided into two categories: statement level and predicate level. 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 guideline applies 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 will override 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.

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 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 a desired access method for satisfying a table reference in a statement.
- *joinRequest* – A join request specifies a desired method and sequence for performing a join operation. Join requests are composed of other access or 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. These methods are described in the *Performance Guide*.

Forming table references in optimization guidelines: The term *table reference* is used in this document to mean any table, view, table expression, or alias in an SQL statement or view definition. An optimization guideline can identify a table

reference either using its exposed name in the original statement or using the unique correlation name associated with the table reference in the *optimized statement*. Extended names, which are sequences of exposed names, can be used to help uniquely identify table references embedded in views. Optimization guidelines that identify exposed or extended names which 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. This following sections describe these particular aspects of the optimization guidelines technology in greater detail. Due to possible query transformations, there is no guarantee that an exposed or extended name will still exist during optimization, in such a case, any guideline targeting the table reference will be ignored.

Using exposed names of the original statement to identify table references

A table reference is identified using its exposed name. 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 apply to the TABLE attribute value.

The TABLE attribute value of an optimization guideline is compared to each exposed name of the statement. Only a single match is permitted in this release of DB2. 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 (thus the TABLE attribute value is considered implicitly qualified by the default schema in effect for the statement). These concepts are illustrated by the example statement in Figure 32. Assume the statement is optimized using the default schema "Tpcd".

```
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';
```

Figure 32. Using TABLE attribute value of an optimization guideline to compare to each exposed name of a statement

TABLE attribute values that would properly identify table references in the statement include '"Tpcd".PARTS', 'PARTS', 'Parts' (since the identifier is not-delimited, it will be converted to upper case). TABLE attribute values that fail to identify a table reference in the statement include '"Tpcd2".SUPPLIERS', 'PARTSUPP' (not an exposed name), and 'Tpcd.PARTS' (the identifier Tpcd must be delimited or it will be converted to upper case).

The exposed name can be used to target any table reference in the original statement, view, SQL function, or trigger.

Using exposed names of the original statement to identify table references in views

Optimization guidelines can use extended syntax to identify table references embedded in views.

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 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.

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 DB2.

Ambiguous table references

An optimization guideline is considered invalid and is not applied if it matches multiple exposed or extended names.

To disambiguate, the view can be rewritten to use unique correlation names or the TABID attribute can be used.

Note: Table references identified by the TABID field are never ambiguous since all correlation names in the optimized statement are unique.

Conflicting optimization guidelines

Multiple optimization guidelines cannot identify the same table reference.

When two or more guidelines refer to the same table, only the first is applied; all other guidelines are ignored and an error message is issued.

There is a restriction with multiple query rewrite INLIST2JOIN rewrite request elements specifying the enable OPTION at a predicate level. Only one such INLIST2JOIN rewrite request element can be specified at a predicate level in one query.

Verifying that optimization guidelines have been used:

The optimizer makes every attempt to adhere to the optimization guidelines specified in an optimization profile or through SQL embedded optimization guidelines, also known as a statement profile; however, the optimizer can reject invalid or inapplicable guidelines.

To use the EXPLAIN facility, the EXPLAIN tables must exist. The DDL to create the EXPLAIN tables is EXPLAIN.DDL and can be found in the misc subdirectory of the sqllib directory.

To verify that a valid optimization guideline has been used:

1. Run the EXPLAIN command on the statement. If an optimization guideline was in effect for the statement using either an optimization profile or SQL comment, the optimization profile name will appear as a RETURN operator argument in the EXPLAIN_ARGUMENTS 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 will appear as a RETURN operator argument. The types of the 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 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
```

Figure 33. Using a query to return results from an explained statement

If the optimization guideline is active and the explained statement matches the statement contained in the STMTKEY element of the optimization guideline, then a query similar to that in Figure 33 produces the output in Figure 34. The value for the STMTPROF argument is the same as the ID attribute in the STMTPROFILE element.

```
TYPE      VALUE
--------- -------------------------
OPT_PROF  NEWTON.PROFILE1
STMTPROF  Guidelines for TPCD Q9
```

Figure 34. Output from a query to return results from an explained statement

Optimization profiles

Anatomy of an optimization profile: This section introduces you to the contents of an optimization profile. The valid optimization profile contents for a given DB2 release is described by an XML schema called the current optimization profile schema (COPS).

The COPS for the current release of DB2 is listed in "Current optimization profile schema" on page 377.

An optimization profile can contain global guidelines, which apply to all DML statements while the profile is in effect, and it can contain specific guidelines that each apply to a single DML statement in a package. For example:

- You could write a global optimization guideline that requests that the optimizer refer to the materialized query tables Test.SumSales and Test.AvgSales for every statement encountered while the current optimization profile is active.
- You could write a statement optimization guideline that requests that the I_SUPPKEY index be used to access the SUPPLIERS table whenever the optimizer encounters the target statement.

An optimization profile, therefore, contains two elements that define the major sections where you can specify the two types of guidelines: a global *OPTGUIDELINES* element and any number of *STMTPROFILE* elements. An optimization profile must also contain a *OPTPROFILE* element, which defines the section that contains metadata and processing directives.

Figure 35 shows an example of a valid optimization profile for DB2 Version 9.1. This optimization profile has a global optimization guidelines section, and one statement profile section.

```
<?xml version="1.0" encoding="UTF-8"?>
<OPTPROFILE VERSION="9.1.0.0">

  <!--
       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 TPCD Q9">
    <STMTKEY SCHEMA="TPCD">
      <![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>
```

Figure 35. A valid optimization profile

The OPTPROFILE element

The optimization profile begins with the OPTPROFILE element. In the example, Figure 35, this element consists of a version attribute indicating that the optimization profile version is 9.1.

The global optimization guidelines section

Global optimization guidelines specify optimization guidelines that apply to all statements for which the optimization profile is in effect.The global optimization guidelines section is represented by the global *OPTGUIDELINES* element. In the example, shown in Figure 35 on page 371, this section contains a single global optimization guideline specifying that the materialized query tables (MQTs) Test.AvgSales and Test.SumSales should be considered for answering 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. Statement profile sections are represented by the *STMTPROFILE* element. In the example, shown in Figure 35 on page 371, 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.

The statement key identifies the statement to which the statement level optimization guidelines apply. In the example, shown again in Figure 35 on page 371, the *STMTKEY* element contains the original statement text and other information needed to unambiguously identify the targeted statement.

The optimizer automatically matches a statement profile with the appropriate statement using the statement key. This relationship allows you to provide optimization guidelines for a statement in an application without having to modify the application.

The statement level optimization guidelines section of the statement profile is represented by the *OPTGUIDELINES* element. Upon a successful match of the statement key in a statement profile, the optimizer refers to the associated statement optimization guidelines when optimizing the statement.

The statement optimization guidelines section

In the example in Figure 35 on page 371, the statement profile section includes a statement optimization guidelines section, represented by the *OPTGUIDELINES* element

This section specifies aspects of the desired query execution plan for the referenced statement. It is made up of one or more *access* and *join* requests, which specify the desired methods for accessing and joining tables in the statement. The guidelines in Figure 35 on page 371 contain one access request, which specifies that the SUPPLIERS table referenced in the nested subselect use an index named I_SUPPKEY.

In Figure 35 on page 371, the *indexScan* element indicates an index access request. The *tableReference* element of the indexScan element specifies that the SUPPLIERS table be targeted for the access request. The *index* element specifies that the I_SUPPKEY index be used.

Guidelines need only specify parts of the desired query execution plan. The optimizer is able to apply its cost-based model to choose the remainder of the plan.

Creating an optimization profile:

An optimization profile must be valid according to the current optimization profile schema.

Since an optimization profile can contain many combinations of guidelines, this task specifies only those steps common in creating any optimization profile.

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.
2. Create a new XML document 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 will apply, for example, inventory_db.xml.
3. Add the XML declaration to the document. If you do not specify an encoding format, then UTF-8 is assumed. Save the document with UTF-16 encoding, if possible. DB2 is more efficient when processing this encoding.

   ```
   <?xml version="1.0" encoding="UTF-16"?>
   ```

4. Add the optimization profile section to the document.

   ```
   <OPTPROFILE VERSION="9.1.0.0">
   </OPTPROFILE>
   ```

5. Within the optimization profile element, create global or statement level guidelines according to your needs and save the file.

Creating statement optimization guidelines:

These steps describe how to create statement optimization guidelines.

"Creating an optimization profile" in which you want to insert the statement guidelines.

To create a statement optimization guideline:

1. Exhaust all other tuning options. Refer to the *Performance Guide*. For example:
   a. Ensure that the data distribution statistics have been recently updated by the RUNSTATS utility.
   b. Ensure that DB2 is running with the proper optimization class setting for the workload.
   c. Ensure the optimizer has the appropriate indexes to access the tables referenced in the query.
2. Run the EXPLAIN facility on the problem statement and analyze the output to determine if any guidelines would be appropriate. If you determine that a statement optimization guideline would help, proceed.
3. Obtain the *original* statement by running a query 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 in the statement key. For example:

```
<STMTPROFILE ID="Guidelines for TPCD Q9">
   <STMTKEY SCHEMA="TPCD"><![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. Below the statement key, compose the statement optimization guidelines. Use the exposed names to identify the objects used in the 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.

To test the results, follow the procedures in "Configuring DB2 to use an optimization profile," "Specifying which optimization profile the optimizer is to use," and "Verifying that optimization guidelines have been used" on page 369. If you do not achieve the desired results, make changes to the guidelines (or specify more aspects of the execution plan) and update the optimization profile as described in "Modifying an optimization profile" on page 376. Repeat as necessary.

Configuring DB2 to use an optimization profile:

After an optimization profile file has been composed and its contents 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.

To configure DB2 to use an optimization profile:

1. Create the optimization profile table as shown in "SYSTOOLS.OPT_PROFILE table" on page 400. Each row of the table can contain one optimization profile: the SCHEMA and NAME column identify the optimization profile and the PROFILE column contains the text of the optimization profile.
2. Optional: You can grant any authority that satisfies your database security requirements. The optimizer will be able to read the table regardless of the authority set.
3. Insert into the table the optimization profiles that you want to use.

Specifying which optimization profile the optimizer is to use: To specify that an optimization profile be used at the package level, you can use the OPTPROFILE bind option. To specify that an optimization profile be used at the statement level, you can use the CURRENT OPTIMIZATION PROFILE special register. 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 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

Binding an optimization profile to a package:

When you prepare a package by using a BIND or PRECOMPILE command, you can use the OPTPROFILE option to specify the optimization profile to use for the package.

This method is the only way to apply an optimization profile to static statements, and the profile specified 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.

You can bind an optimization profile in SQLJ and in embedded SQL using the APIs (for example, sqlaprep) or from the CLP.

For example, to bind the inventory database optimization profile to the 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 option is used as the implicit qualifier.

Setting an optimization profile within an application:

In an application, you can control the setting of the current optimization profile for dynamic statements by using the SET CURRENT OPTIMIZATION PROFILE statement. The optimization profile name that you provide in the statement must be a schema qualified name. If you do not provide a schema name, then 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.

To set an optimization profile within an application:

- You can use the SET CURRENT OPTIMIZATION PROFILE statement throughout your application. For example, the final 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 ... ;
```

Figure 36. Using the SET CURRENT OPTIMIZATION PROFILE statement to change a profile

- If you want the optimizer to use the default optimization profile that was in effect when the application started, specify the value null. For example:

  ```
  EXEC SQL SET CURRENT OPTIMIZATION PROFILE = NULL;
  ```

- If you want the optimizer to not use optimization profiles, specify the empty string (''). For example:

  ```
  EXEC SQL SET CURRENT OPTIMIZATION PROFILE = '';
  ```

- If it is a CLI application, you can add the CURRENTOPTIMIZATIONPROFILE parameter to the db2cli.ini file. To add the entry to the file, use the configuration assistant or the UPDATE CLI CONFIGURATION command. For example:

  ```
  DB2 UPDATE CLI CFG FOR SECTION SANFRAN USING CURRENTOPTIMIZATIONPROFILE JON.SALES
  ```

  This would result 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.

Modifying an optimization profile:

You can modify an optimization profile by editing the document, validating it against the current optimization profile schema (COPS), and replacing the original document in the SYSTOOLS.OPT_PROFILE table with the new version. However, when an optimization profile is referenced, it is compiled and cached in memory; these references must also be removed. Use the FLUSH OPTIMIZATION PROFILE CACHE statement to both remove the old optimization profile from the optimization profile cache and to invalidate from the dynamic plan cache any statement prepared using the old profile.

To modify an optimization profile:

1. Edit the optimization profile file with the necessary changes and validate the XML.
2. Update the row in the SYSTOOLS.OPT_PROFILE table with the new profile.
3. If you did not create "Triggers to flush the optimization profile cache" on page 400, issue the statement FLUSH OPTIMIZATION PROFILE CACHE.

   Note: When you flush the optimization profile cache, any dynamic statements prepared with the old optimization profile are also invalidated in the dynamic plan cache.

Any subsequent reference to the optimization profile will cause the optimizer to read the new profile and reload it into the optimization profile cache. Also, because of the soft invalidation of statements prepared under the old optimization profile, any calls made to those statements will be prepared under the new optimization profile and re-cached 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, then you must also flush the deleted optimization profile from the optimization profile cache.

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" on page 400, flush any versions of the optimization profile that might be contained in the optimization profile cache by running the FLUSH OPTIMIZATION PROFILE CACHE command from the CLP.

   Note: This action also causes any statements prepared using the old optimization profile to be invalidated in the dynamic plan cache.

Any subsequent reference to the optimization profile will cause the optimizer to issue a warning SQL0437W with reason code 13.

XML schema for optimization profiles and guidelines

Current optimization profile schema:

This is the schema for optimization profiles.

```
<?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 2007. 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 V9.5.0.0                     -->
<!--                                                                                        -->
<!-- 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:simpleType>
            <xs:restriction base="xs:NMTOKEN">
               <xs:enumeration value="9.5.0.0"/>
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
   </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.                                                     -->
<!-- + The REOPT element can affect how statements involving variables are optimized.       -->
<!-- ************************************************************************************** -->
<xs:complexType name="globalOptimizationGuidelinesType">
   <xs:sequence>
      <xs:group ref="MQTOptimizationChoices" />
      <xs:group ref="computationalPartitionGroupOptimizationChoices" />
      <xs:group ref="generalRequest"/>
   </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.  -->
<!--                                                                                  -->
<!--********************************************************************************-->
<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:choice>
   </xs:sequence>
</xs:complexType>
<!--******************************************************************************** -->
<!-- Choices of general request elements.                                           -->
<!-- REOPT can be used to override the setting of the REOPT bind option.            -->
<!--******************************************************************************** -->
<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: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                                     -->
<!-- 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="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>
<!--******************************************************************************** -->
<!-- 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.                                        -->
<!--***************************************************************************************** -->
<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: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 index ANDing access request element.                                   -->
<!-- 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.                                                            -->
<!--***************************************************************************************** -->
<xs:complexType name="indexAndingType">
   <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: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>
<!--*****************************************************************************************-->
<!-- Use for derived table access or other cases where the access method is not of        -->
<!-- consequence.                                                                         -->
<!--*****************************************************************************************-->
<xs:complexType name="anyAccessType">
   <xs:complexContent>
      <xs:extension base="accessType"/>
   </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>
<!--*****************************************************************************************-->
<!-- 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>
<!--*****************************************************************************************-->
<!-- 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>
</xs:schema>
```

XML schema for the OPTPROFILE element: The OPTPROFILE element is the root of an optimization profile. The OPTPROFILE element is defined as follows:

XML Schema

```
<xs:element name="OPTPROFILE">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="OPTGUIDELINES" type="globalOptimizationGuidelinesType"
           minOccurs="0"/>
         <xs:element name="STMTPROFILE" type="statementProfileType" minOccurs="0"
           maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="VERSION" use="optional">
         <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
               <xs:enumeration value="9.1.0.0"/>
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
   </xs:complexType>
</xs:element>
```

Description

The optional OPTGUIDELINES sub-element defines the global optimization guidelines section of the optimization profile. Each STMTPROFILE sub-element defines a statement profile section. The VERSION attribute identifies the current

optimization profile schema that a given optimization profile was created and validated against.

XML schema for the global OPTGUIDELINES element: The type *globalOptimizationGuidelinesType* defines the format of a global OPTGUIDELINES element.

XML Schema

```
<xs:complexType name="globalOptimizationGuidelinesType">
   <xs:sequence>
      <xs:group ref="MQTOptimizationChoices"/>
      <xs:group ref="computationalPartitionGroupOptimizationChoices"/>
      <xs:group ref="generalRequest"/>
   </xs:sequence>
</xs:complexType>
```

Description

Global optimization guidelines can be defined with elements from the group MQTOptimizationChoices, with elements from the group computationalPartitionGroupChoices, or with the REOPT element.

The MQTOptimizationChoices group elements, defined in "MQT optimization choices," can be used to influence MQT substitution. The computationalPartitionGroupOptimizationChoices group elements, defined in "Computational partition group optimization choices" on page 385, can be used to influence computational partition group optimization.

Computational partition group optimization involves the dynamic redistribution of data read from remote data sources. It applies only in partitioned federated database configurations.

The REOPT element, defined in "REOPT global optimization guidelines" on page 386, can be used to influence when optimization occurs for statements referencing variables.

MQT optimization choices: The group *MQTOptimizationChoices* defines a set of elements that can be used to influence materialized query table (MQT) optimization. In particular, the elements can be used to either enable or disable consideration of MQT substitution, or to specify the complete set of MQTs 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 specify the values ENABLE or DISABLE. The default value of the OPTION attribute is ENABLE.

Alternately, zero or more MQT elements can be provided. The NAME attribute of an MQT element identifies an MQT 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 these MQTs are considered by the optimizer. The decision to perform MQT substitution using one or more of the specified MQTs is still a cost-based decision.

Examples

The following example illustrates how to disable MQT optimization.

```
<OPTGUIDELINES>
   <MQTOPT OPTION='DISABLE'/>
</OPTGUIDELINES>
```

This following example illustrates how to limit MQT optimization to the MQTs ″Tpcd.PARTSMQT″ and ″COLLEGE.STUDENTS″.

```
<OPTGUIDELINES>
   <MQT NAME='Tpcd.PARTSMQT'/>
   <MQT NAME='COLLEGE.STUDENTS'/>
<OPTGUIDELINES>
```

Computational partition group optimization choices: The group *computationalPartitionGroupOptimizationChoices* defines a set of elements that can be used to influence computational partition group optimization. In particular, the elements can be used to either enable or disable computational group optimization, or to specify the particular partition group 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 specify the values ENABLE or DISABLE. The default value of the OPTION attribute is ENABLE.

Alternately, a PART element can be used to specify the partition group to be used for computational partition group optimization. The NAME attribute of the PART

element identifies the partition group and must identify an existing partition group. The decision to perform dynamic redistribution using the specified partition group is a cost-based decision.

Examples

The following example illustrates how to disable computational partition group optimization.

```
<OPTGUIDELINES>
   <PARTOPT OPTION='DISABLE'/>
</OPTGUIDELINES>
```

This next example shows how to indicate that the partition group WORKPART be used for computational partition group optimization:

```
<OPTGUIDELINES>
   <MQT NAME='Tpcd.PARTSMQT'/>
   <PART NAME='WORKPART'/>
<OPTGUIDELINES>
```

REOPT global optimization guidelines: The REOPT global optimization guideline controls when optimization occurs for DML statements that contain variables (host variables, parameter markers, global variables or special registers). The description and syntax of the REOPT element is the same for both global optimization guidelines and statement optimization guidelines. Refer to "REOPT requests" on page 390.

XML schema for the STMTPROFILE element: The type *statementProfileType* defines the format of a STMTPROFILE element.

```
XML Schema

<xs:complexType name="statementProfileType">
   <xs:sequence>
      <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 for which the optimization profile is in effect. A statement profile has the following parts:

- Statement Key - An optimization profile can be in effect for more than one statement in an application. The optimizer automatically matches each statement profile to the proper statement of the application using the statement key. This technique allows the user to provide optimization guidelines for a statement without editing the application.

  The statement key is comprised of the text of the statement as written in the application, as well as other pieces of information needed to unambiguously identify the proper application statement. The STMTKEY sub-element, described in "XML schema for the STMTKEY element" on page 387, represents the statement key.
- Statement Level Optimization Guidelines - This section of the statement profile specifies the optimization guidelines in effect for the statement identified by the statement key. Refer to "XML schema for the statement-level OPTGUIDELINES element" on page 388.

- Statement Profile Name - A user supplied name used in diagnostic output to indicate the particular statement profile used to optimize a statement.

XML schema for the STMTKEY element: The type *statementKeyType* defines the format of a STMTKEY element.

```
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 statement text part of the statement key is included as data between the start and end STMTKEY element tags.

This 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 function paths can be included in the function path by separating each path with a comma. The specified function paths must match exactly the function paths specified in the compilation key.

Examples

The following example statement key will match 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>
```

Note: A CDATA section (starting with <![CDATA[and ending with]]>) is used around the statement text because the statement text contains the special XML character '>'.

Statement key and compilation key matching: The statement key is used to identify the particular application statement to which the statement level optimization guidelines apply. When DB2 compiles any static or dynamic SQL statement, the setting of certain parameters (set by special registers, and either the bind or precompile options) influence how the statement is interpreted semantically by the compiler. The SQL statement and the settings of these particular SQL compiler parameters together form what is called the *compilation key*. Each part of the statement key corresponds to a part of the compilation key.

A *statement key* is comprised of the following parts:

- Statement Text - The text of the statement as written in the application.
- Default Schema - The schema name used as the implicit quantifier for unqualified table names. This part is optional but should be provided if there are unqualified table names in the statement text.
- Function Path - The function path 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 DB2 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. A statement key and compilation key are said to 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. In effect, each unspecified part of the statement key is treated as a wild card, and matches the corresponding part of any compilation key.

Once 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 (based on document order). Moreover, no error or warning is issued in this case.

XML schema for the statement-level OPTGUIDELINES element: The OPTGUIDELINES element of a statement profile defines the optimization guidelines in effect for the statement identified by the associated statement key. The OPTGUIDELINES element is defined of type *optGuidelinesType* as follows:

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:choice>
  </xs:sequence>
</xs:complexType>
```

Description

The optGuidelinesType group defines the set of valid sub-elements of the OPTGUIDELINES element. Each sub-element is understood as an optimization guideline by the DB2 optimizer. Sub-elements can be categorized as either a general request element, a rewrite request element, an access request element, or a join request element.

- General request elements are used to specify general optimization guidelines. General optimization guidelines can be used to change the optimizer's search space.
- Rewrite request elements are used to specify query rewrite optimization guidelines. Query rewrite guidelines can be used to affect the query transformations applied in determining the optimized statement.
- Access and join request elements are plan optimization guidelines. Plan optimization guidelines can be used to affect the access methods, join methods, and join orders used in the execution plan for the optimized statement.

Note: Optimization guidelines specified in a statement profile take precedence over those specified in the global section of an optimization profile.

XML schema for general optimization guidelines: General optimization guidelines are used to specify guidelines that are not specific to a particular phase of the optimization process. General optimization guidelines can be used to change the optimizer's search space. They consist of the *generalRequest* group.

```
<!--****************************************************************************** --> \
<!-- Choices of general request elements.                                         --> \
<!-- REOPT can be used to override the setting of the REOPT bind option.          --> \
<!--****************************************************************************** --> \
<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:sequence>
</xs:group>
```

Description

General request elements can be used to define general optimization guidelines. General optimization guidelines affect the optimization search space and hence, 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 bind option, the dft_degree database configuration parameter or a previous SET CURRENT DEGREE statement. The DEGREE general request will only be considered if the database manager is configured for intra-partition parallelism and a warning will be returned when the database manager is not configured for intra-partition parallelism. The DEGREE general request element is defined by degreeType as follows:

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 which specifies the setting of the DEGREE option. The attribute can take an integer value from 1 to 32767 or the string values ″-1″ or ″ANY″. The value -1 (or equivalently ″ANY″) indicates that the degree of parallelism used is determined by DB2. The value 1 indicates that the query should not use intra-partition parallelism.

QRYOPT requests: The QRYOPT general request element can be used to override the setting of the bind option, the dft_qryopt database configuration parameter or a previous SET CURRENT QUERY OPTIMIZATION statement. The QRYOPT general request element is defined by qryoptType as follows:

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 which specifies the setting of the DEGREE option. The attribute can take any value from the list: 0, 1, 2, 3, 5, 7 and 9. The *Performance Guide* has detailed information on the behavior of the each of the possible settings.

REOPT requests: The REOPT general request element can be used to override the setting of the REOPT bind option. The REOPT bind option affects the optimization of statements with parameter markers or host variables. The REOPT general request element is defined by reoptType as follows:

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 which specifies the setting of the REOPT option. The attribute can take either the value ONCE or to the value ALWAYS. The value ONCE indicates that the statement should be optimized for the first set of host variable or parameter marker values. The value ALWAYS indicates that the statement should be optimized for each set of host variable or parameter marker values. The *Performance Guide* has detailed information on the behavior of the each of the possible REOPT bind option settings.

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 may be desirable to disable or limit the time spent on real-time statistics collection to avoid extra overhead at statement compilation time.

```
<!--****************************************************************************************--> \
<!-- 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 values ENABLE or DISABLE. ENABLE is the default if no option is specified.
- The TIME attribute specifies the maximum amount of time in milliseconds to be spent on real-time statistics collection at statement compilation time, for a single statement.

If ENABLE is specified for the OPTION attribute, automatic statistics collection and real-time statistics must be enabled by their corresponding configuration parameters. Otherwise, the optimization guideline will not be applied, and you will get warning message SQL0437W (reason code 13).

For example, the following RTS request enables real-time statistics collection and limits real-time statistics collection time to 3.5 seconds.

```
<RTS OPTION="ENABLE" TIME="350" />
```

XML schema for query rewrite guidelines: Query rewrite optimization guidelines impact the query rewrite optimization phase. It consists of the *rewriteRequest* group.

The *rewriteRequest* group defines the set of valid rewrite request element choices. The rewrite requests are *INLIST2JOIN*, *SUBQ2JOIN*, *NOTEX2AJ* and *NOTIN2AJ*:

```
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

- INLIST2JOIN: Enables or disables IN-LIST to join rewrite transformation. It can be used as a statement level guideline to enable or disable all IN-LIST predicates in a query to be considered for IN-LIST to join transformation. It can also be used as a predicate level guideline to enable or disable a specified IN-LIST predicate to be considered for IN-LIST to join transformation. If both statement level and predicate level guidelines are specified, the predicate guideline will override the statement level guideline.
- SUBQ2JOIN: Enables or disables subquery to join rewrite transformation. It can only be used as a statement level guideline to enable or disable all subqueries in a query to be considered for subquery to join rewrite transformation.
- NOTEX2AJ: Enables or disables NOT-EXISTS to anti-join rewrite transformation. It can only be used as a statement level guideline to enable or disable all NOT-EXISTS subqueries in a query to be considered for NOT-EXISTS to anti-join rewrite transformation.

- NOTIN2AJ: Enables or disables NOT-IN to anti-join rewrite transformation. It can only be used as statement level guideline to enable or disable all NOT-IN subqueries in a query to be considered for NOT-IN to anti-join rewrite transformation.

IN-LIST-to-join rewrite request: A *INLIST2JOIN* request element can be used to enable or disable IN-LIST predicate to join rewrite transformation. It can be specified as statement level guideline or predicate level guideline. For predicate level guideline, only one optimization guideline with option ENABLE is permitted in a query.

It is defined by the complex type *inListToJoinType* as follows:

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

A *INLIST2JOIN* element has three optional attributes and no sub-element. The OPTION attribute has type *optionType*, which has the value “ENABLE” or “DISABLE”. If no OPTION attribute is specified, the default value is “ENABLE”. The table name attribute and column name attribute are used to specify the IN-LIST predicate which involves the specified table and column. If table name attribute and column name attribute are not specified or both table name and column name attributes are specified as empty string “”, it will be handled as a statement level guideline. If table name is specified or both table name and column name are specified, it will be handled as predicate level guideline. If table name is not specified or specified as empty string “” but column name is specified, SQLO437W with reason code 13 will be issued and the optimization guideline will be ignored.

NOT-EXISTS-anti-join rewrite requests: A *NOTEX2AJ* request element can be used to enable or disable NOT-EXISTS to anti-join rewrite transformation. It can be specified as a statement level guideline only.

It is defined by the complex type *notExisitsToAntiJoinType* as follows:

XML Schema

```
<xs:complexType name="notExistsToAntiJoinType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
</xs:complexType>
```

Description

A *NOTEX2AJ* element has one optional attribute and no sub-element. The option attribute has type *optionType*, which has the value “ENABLE” or “DISABLE”. If no option attribute is specified, the default value is “ENABLE”.

NOT-IN-to-anti-join rewrite requests: A *NOTIN2AJ* request element can be used to enable or disable NOT-IN to anti-join rewrite transformation. It can be specified as a statement level guideline only.

It is defined by the complex type *notInToAntiJoinType* as follows:

XML Schema

```
<xs:complexType name="notInToAntiJoinType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
</xs:complexType>
```

Description

A *NOTIN2AJ* element has one optional attribute and no sub-element. The option attribute has type *optionType*, which has the value “ENABLE” or “DISABLE”. If no option attribute is specified, the default value is “ENABLE”.

Subquery-to-join rewrite requests: A *SUBQ2JOIN* request element can be used to enable or disable subquery to join rewrite transformation. It can be specified as a statement level guideline only.

It is defined by the complex type *subqueryToJoinType* as follows:

XML Schema

```
<xs:complexType name="subqueryToJoinType">
   <xs:attribute name="OPTION" type="optionType" use="optional" default="ENABLE"/>
</xs:complexType>
```

Description

A *SUBQ2JOIN* element has one optional attribute and no sub-element. The option attribute has type *optionType*, which has the value “ENABLE” or “DISABLE”. If no option attribute is specified, the default value is “ENABLE”.

XML schema for plan optimization guidelines: Plan optimization guidelines can be broken down into access requests and join requests:

- Access request - An access request specifies a desired access method for a table reference.
- Join request - A join request specifies a desired method and sequence for performing a join operation. Join requests are composed of other access or join requests.

Most of the access request choices 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. These methods are described in the *Performance Guide*. This section details each of the access request elements and join request elements that can be used to influence plan optimization.

Access requests: The *accessRequest* group defines the set of valid access request element choices. An access request specifies a desired method for satisfying a table reference in a statement.

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="ACCESS" type="anyAccessType"/>
   </xs:choice>
</xs:group>
```

Description

- TBSCAN, IXSCAN, LPREFETCH, IXAND, and IXOR

  These elements correspond to DB2 data access methods, and can only be applied to local tables referenced in the statement. They cannot refer to nicknames (remote tables) or derived tables (the result of a subselect).
- ACCESS

  This element is used in cases where the join order, as opposed to the access method, is the primary concern. This element must be used when the target table reference is a derived table. The optimizer will choose the access method for the target table reference using cost.

Access types: Common aspects of the TBSCAN, IXSCAN, LPREFETCH, IXAND, IXOR, and ACCESS elements are defined by the abstract type accessType shown below.

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: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. "Forming table references in optimization guidelines" on page 367 describes how to form proper table references from an access request element. They 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 you want an execution plan that specifies that the table targeted by the access request appears as the first table 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 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 issued.

Any access requests: The *ACCESS* access request element can be used to request that the optimizer choose the appropriate method for accessing a table in a cost-based fashion. This choice is typically used to specify an access request that only indicates how the local table is joined with other tables in the statement. This access request element must be used when making reference to a derived table. A derived table is the result of another subselect. An ACCESS access request element is defined by the complex type *anyAccessType* as follows:

XML Schema

```
<xs:complexType name="anyAccessType">
   <xs:complexContent>
      <xs:extension base="accessType"/>
   </xs:complexContent>
</xs:complexType>
```

Description

The complex type *anyAccessType* is a simple extension of the abstract type *accessType*. No new elements or attributes are added.

Index ANDing access requests: An *indexAnding* access request element can be used to request that the optimizer use the index ANDing data access method to access a local table. It is defined by the complex type *indexAndingType* as follows:

XML Schema

```
<xs:complexType name="indexAndingType">
   <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:extension>
   </xs:complexContent>
</xs:complexType>

<xs:complexType name="indexType">
   <xs:attribute name="IXNAME" type="xs:string" use="optional"/>
</xs:complexType>
```

Description

The complex type *indexAndingType* extends the abstract type *localAccessType* by adding an optional *INDEX* attribute and optional INDEX sub-elements. The INDEX attribute can be used to specify the first index used in the index ANDing operation. If the INDEX attribute is used, the optimizer will choose the additional indexes and access sequence in a cost-based fashion. The INDEX 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 will choose both the indexes and the access sequence in a cost-based fashion.
- If indexes are specified either using the attribute or sub-elements, they must identify indexes defined on the table identified by the TABLE or TABID attribute.
- If there are no indexes defined on the table, the access request is ignored and a warning SQL0437W with reason code 13 is issued.

Block indexes must appear before record indexes in an index ANDing access request. A SQL0437W warning with reason code 13 is issued if this requirement is not satisfied. The index ANDing access method requires at least one predicate 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 a warning SQL0437W with reason code 13 is issued. If the index ANDing data access method is not in the search space in effect for the statement, then the access request is ignored and a warning SQL0437W with reason code 13 is issued.

Index ORing access requests: An *IXOR* access request element can be used to request that the optimizer use the index ORing data access method to access a local table. It is defined by the complex type *indexOringType* as follows:

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 in effect for the statement, the access request is ignored and a warning SQL0437W with reason code 13 is issued. The optimizer will choose the predicates and indexes used in the index ORing operation in a cost-based fashion. The index ORing access method requires at least one IN predicate able to be indexed or a predicate with terms able to be indexed and connected by a logical OR operation. If index ORing is not eligible because the required predicate or indexes does not exist, the request is ignored and a warning SQL0437W with reason code 13 is issued.

Index scan access requests: A *IXSCAN* access request element can be used to request that the optimizer use an index scan to access a local table. It is defined by the complex type *indexScanType* as follows:

```
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 that is to be used to access the table.

- If the index scan access method is not in the search space in effect for the statement, the access request is ignored and a warning SQL0437W with reason code 13 is issued.
- If the INDEX attribute is specified, it must identify an index defined on the table identified by the TABLE or TABID attribute.
- If the index does not exist, the access request is ignored and a warning SQL0437W with reason code 13 is issued.
- If the INDEX attribute is not specified, the optimizer will choose an index in a cost-based fashion.
- If no indexes are defined on the target table, the access request is ignored and a warning SQL0437W with reason code 13 is issued.

List prefetch access requests: The *listPrefetch* element can be used to request that the optimizer use a list prefetch index scan to access a local table. It is defined by the complex type *listPrefetchType* as follows:

```
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 in effect for the statement, the access request is ignored and a warning SQL0437W with reason code 13 is issued.
- The list prefetch access method requires at least one predicate able to be indexed.
- If the list prefetch is not eligible because the required predicate does not exist, the access request is ignored and a warning SQL0437W with reason code 13 is issued.
- If the INDEX attribute is specified, it must identify an index defined on the table specified by the TABLE or TABID attribute.
- If the index does not exist, the access request is ignored and a warning SQL0437W with reason code 13 is issued.
- If the INDEX attribute is not specified, the optimizer will choose an index in a cost-based fashion.
- If there are no appropriate indexes defined on the targeted table, the access request is ignored and a warning SQL0437W with reason code 13 is issued.

Table scan access requests: A *TBSCAN* access request element can be used to request that the optimizer use a sequential table scan to access a local table. It is defined by the complex type *tableScanType* as follows:

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* shown above. No new elements or attributes are added. If the table scan access method is not in the search space in effect for the statement, the access request is ignored and a warning SQL0437W with reason code 13 is issued.

Join requests: The *joinRequest* group defines the set of valid join request element choices. A join request specifies a desired 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

The *NLJOIN*, *MSJOIN*, and *HSJOIN* join request elements correspond to the nested-loop, merge, and hash join methods respectively. The *Performance Guide*

describes the eligibility requirements and execution characteristics of these join methods in more detail. The *JOIN* join request element indicates that the optimizer is free to choose the join method. This choice is used in cases where specifying a particular join order is the 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.

Any join requests: A *JOIN* join request element can be used to request that the optimizer join two tables in a particular order but with any join method of the optimizer's choosing. Either of the input tables can be a local or derived table, as specified by an access request sub-element, or they can be the result of a join operation, as specified by a join request sub-element. A *JOIN* join request element is defined by the complex type *anyJoinType* as follows:

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.

Hash join requests: A *HSJOIN* join request element can be used to request that the optimizer join two tables using a hash join method. Either of the input tables can be a local or derived table, as specified by an access request sub-element, or they can be the result of a join operation, as specified by a join request sub-element. A *HSJOIN* join request element is defined by the complex type *hashJoinType* as follows:

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 in effect for the statement, the join request is ignored and a warning SQL0437W with reason code 13 is issued.

Merge join requests: A *MSJOIN* join request element can be used to request that the optimizer join two tables using a merge join method. Either of the input tables can be a local or derived table, as specified by an access request sub-element, or they can be the result of a join operation, as specified by a join request sub-element. A *MSJOIN* join request element is defined by the complex type *mergeJoinType* as follows:

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 in effect for the statement, the join request is ignored and a warning SQL0437W with reason code 13 is issued.

Nested-loop join requests: A *NLJOIN* join request element can be used to request that the optimizer join two tables using a nested-loop join method. Either of the input tables can be a local or derived table, as specified by an access request sub-element, or they can be the result of a join operation, as specified by a join request sub-element. A *NLJOIN* join request element is defined by the complex type *nestedLoopJoinType* as follows:

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 in effect for the statement, the join request is ignored and a warning SQL0437W with reason code 13 is issued.

Types of join requests: Common aspects of all join request elements are defined by the abstract type *joinType* as follows:

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 *joinType* must have exactly two sub-elements. Either of the sub-elements can be an access request element chosen from the group *accessRequest* or another join request element chosen from the group *joinRequest*. 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 where the tables 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, then all but the initial such request is ignored and a warning SQL0437W with reason code 13 is issued.

SYSTOOLS.OPT_PROFILE table

The SYSTOOLS.OPT_PROFILE table contains all of the optimization profile files. There are two methods to create the table:

- Call the SYSINSTALLOBJECTS procedure:

```
db2 "call sysinstallobjects('opt_profiles', 'c', '', '')"
```

- Issue the CREATE TABLE command:

```
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 qualifier of the optimization profile. The schema name can include up to 30 alphanumeric or underscore characters, but define it as VARCHAR(128) as shown.

NAME

- Specifies the base-name of the optimization profile. The name can include up to 128 alphanumeric or underscore characters.

PROFILE

- The XML document defining the optimization profile.

Note:

1. An optimization profile can be referenced in several different contexts. The NAME and SCHEMA columns of the SYSTOOLS.OPT_PROFILE table together designate the two part optimization profile name. The PROFILE column contains the XML document which defines the optimization profile.
2. No table space or partition group restrictions exist.

Triggers to flush the optimization profile cache: The following SQL procedure and triggers should be created to ensure that the profile cache is automatically flushed when an entry in the SYSTOOLS.OPT_PROFILE table is updated or deleted:

```
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 profile files must be associated with a unique schema-qualified name and stored in the SYSTOOLS.OPT_PROFILE table. You can use 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 from the files into the SYSTOOLS.OPT_PROFILE table. The EXPORT command can be used to retrieve a profile from the OPT_PROFILE table into a file.

Here is an example of inserting three new rows from separate input files into the SYSTOOLS.OPT_PROFILE table. This assumes the files are in the current directory.

1. Create an input file (for example, profiledata) with the schema, name and file name for each row 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, you can either delete them first and insert them again as above or you use the INSERT_UPDATE option with IMPORT:

```
IMPORT FROM profiledata OF DEL MODIFIED BY LOBSINFILE
  INSERT_UPDATE INTO SYSTOOLS.OPT_PROFILE
```

To retrieve the profile for ROBERT.PROF1 into ROBERT.PROF1.xml, assuming the profile is less than 32,700 bytes:

```
EXPORT TO ROBERT.PROF1.xml OF DEL SELECT PROFILE FROM SYSTOOLS.OPT_PROFILE
  WHERE SCHEMA='ROBERT' AND NAME='PROF1'
```

To export data greater than 32,700 bytes, or for further information, refer the documentation for the EXPORT command in the *Command Reference*.

Configuration parameters that affect query optimization

Several configuration parameters affect the access plan chosen by the SQL or XQuery compiler. Many of these are appropriate to a single-partition database environment and some are only appropriate to a partitioned database environment. Assuming a homogeneous partitioned database environment, where the hardware is the same, the values used for each parameter should be the same on all database partitions.

Note: When you change a configuration parameter dynamically, the optimizer might not read the changed parameter values immediately because of older access plans in the package cache. To reset the package cache, execute the FLUSH PACKAGE CACHE command.

In a federated system, if the majority of your queries access nicknames, evaluate the types of queries that you send before you change your environment. For example, in a federated database the buffer pool does not cache pages from data

sources, which are the DBMSs and data within the federated system. For this reason, increasing the size of the buffer does not guarantee that the optimizer will consider additional access-plan alternatives when it chooses an access plan for queries that contain nicknames. However, the optimizer might decide that local materialization of data source tables is the least-cost route or a necessary step for a sort operation. In that case, increasing the resources available might improve performance.

The following configuration parameters or factors affect the access plan chosen by the SQL or XQuery compiler:

- The size of the buffer pools that you specified when you created or altered them.

  When the optimizer chooses the access plan, it considers the I/O cost of fetching pages from disk to the buffer pool and estimates the number of I/Os required to satisfy a query. The estimate includes a prediction of buffer-pool usage, because additional physical I/Os are not required to read rows in a page that is already in the buffer pool.

  The optimizer considers the value of the *npages* column in the SYSCAT.BUFFERPOOLS system catalog tables and, in partitioned database environments, the SYSCAT.BUFFERPOOLDBPARTITIONS system catalog tables.

  The I/O costs of reading the tables can have an impact on:

  – How two tables are joined
  – Whether an unclustered index will be used to read the data
- Default Degree (dft_degree)

  The `dft_degree` configuration parameter specifies parallelism by providing a default value for the CURRENT DEGREE special register and the DEGREE bind option. A value of one (1) means no intra-partition parallelism. A value of minus one (-1) means the optimizer determines the degree of intra-partition parallelism based on the number of processors and the type of query.

  Note: Intra-parallel processing does not occur unless you enable it by setting the *intra_parallel* database manager configuration parameter.
- Default Query Optimization Class (dft_queryopt)

  Although you can specify a query optimization class when you compile SQL or XQuery queries, you can also set a default query optimization class.
- Average Number of Active Applications (avg_appls)

  The optimizer uses the *avg_appls* parameter to help estimate how much of the buffer pool might be available at run-time for the access plan chosen. Higher values for this parameter can influence the optimizer to choose access plans that are more conservative in buffer pool usage. If you specify a value of 1, the optimizer considers that the entire buffer pool will be available to the application.
- Sort Heap Size (sortheap)

  If the rows to be sorted occupy more than the space available in the sort heap, several sort passes are performed, where each pass sorts a subset of the entire set of rows. Each sort pass is stored in a system temporary table in the buffer pool, which might be written to disk. When all the sort passes are complete, these sorted subsets are merged into a single sorted set of rows. A sort is considered to be “piped” if it does not require a system temporary table to store the final, sorted list of data. That is, the results of the sort can be read in a single, sequential access. Piped sorts result in better performance than non-piped sorts and will be used if possible.

When choosing an access plan, the optimizer estimates the cost of the sort operations, including evaluating whether a sort can be piped, by:
  - Estimating the amount of data to be sorted
  - Looking at the *sortheap* parameter to determine if there is enough space for the sort to be piped.
- Maximum Storage for Lock List (locklist) and Maximum Percent of Lock List Before Escalation (maxlocks)

  When the isolation level is **repeatable read (RR)**, the optimizer considers the values of the *locklist* and *maxlocks* parameters to determine whether row level locks might be escalated to a table level lock. If the optimizer estimates that lock escalation will occur for a table access, then it chooses a table level lock for the access plan, instead of incurring the overhead of lock escalation during the query execution.
- CPU Speed (cpuspeed)

  The optimizer uses the CPU speed to estimate the cost of performing certain operations. CPU cost estimates and various I/O cost estimates help select the best access plan for a query.

  The CPU speed of a machine can have a significant influence on the access plan chosen. This configuration parameter is automatically set to an appropriate value when the database is installed or migrated. Do not adjust this parameter unless you are modelling a production environment on a test system or assessing the impact of a hardware change. Using this parameter to model a different hardware environment allows you to find out the access plans that might be chosen for that environment. To have the database manager recompute the value of this automatic configuration parameter, set it to -1.
- Statement Heap Size (stmtheap)

  Although the size of the statement heap does not influence the optimizer in choosing different access paths, it can affect the amount of optimization performed for complex SQL or XQuery statements.

  If the *stmtheap* parameter is not set large enough, you might receive a warning indicating that there is not enough memory available to process the statement. For example, SQLCODE +437 (SQLSTATE 01602) might indicate that the amount of optimization that has been used to compile a statement is less than the amount that you requested.
- Communications Bandwidth (comm_bandwidth)

  Communications bandwidth is used by the optimizer to determine access paths. The optimizer uses the value in this parameter to estimate the cost of performing certain operations between the database partition servers in a partitioned database environment.
- Application Heap Size (applheapsz)

  Large schemas require sufficient space in the application heap.

Database database partition group impact on query optimization

In partitioned database environments, the optimizer recognizes collocation of tables and uses this collocation 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 a partitioned database environment so that the rows from each table being joined are located on the same database partition. During the join operation, the collocation of the data from both joined tables prevents moving data from one database partition to another. Place both tables in the same database partition group to ensure that the data from the tables is collocated.

In a partitioned database environment, depending on the size of the table, spreading data over more database partitions reduces the estimated time (or cost) 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.

Column correlation for multiple predicates

Your applications might contain queries that are constructed with joins such that more than one join predicate joins two tables. This is not unusual when queries need to determine relationships between similar, related columns in different tables.

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
```

When the optimizer chooses a plan for executing this query, it calculates how selective each of the two predicates is. It assumes that they are independent, which means that all variations of elasticity occur for each color, and, conversely, that for each level of elasticity there is raw material of every color. It then estimates the overall selectivity of the pair of predicates by using catalog statistic information for each table based on the number of levels of elasticity and the number of different colors. Based on this estimate, it may choose, for example, a nested loop join in preference to a merge join, or vice versa.

However, it may be that these two predicates are not independent. For example, it may be that the highly elastic materials are available in only a few colors, and the very inelastic materials are only available in a few other colors that are different from the elastic ones. Then the combined selectivity of the predicates eliminates fewer rows so the query will return more rows. Consider the extreme case, in which there is just one level of elasticity for each color and vice versa. Now either one of the predicates logically could be omitted entirely since it is implied by the other. The optimizer might no longer choose the best plan. For example, it might choose a nested loop join plan when the merge join would be faster.

The DB2 optimizer attempts to detect and compensate for correlation of join predicates if you define an index on those columns or if you collect and maintain group column statistics on the appropriate columns.

For example, in elasticity example above, you might define one or both of the following indexes:

```
IX1 PRODUCT.COLOR, PRODUCT.ELASTICITY
```

or

```
IX2 RAWMATERIAL.COLOR, RAWMATERIAL.ELASTICITY
```

or both.

For the optimizer to detect correlation, the non-include columns of this index must be only the correlated columns. The index may also contain include columns to allow index-only scans. If there are more than two correlated columns in join predicates, make sure that you define the index to cover all of them.

One condition that must be met for the optimizer to consider an index key cardinality to detect correlation is that the number of distinct values in each column must be higher for each column from the same table. For example, imagine you have defined IX1 and IX2 as above. If the number of distinct values in PRODUCT.COLOR is less than the number of distinct values in RAWMATERIAL.COLOR and the number of distinct values in PRODUCT.ELASTICITY is less than that of RAWMATERIAL.ELASTICITY, then the optimizer will use IX2 to detect the correlation. Comparing column cardinalities makes it likely, but not certain, that the distinct values in the PRODUCT columns are included in the distinct values of the RAWMATERIAL columns. To make it even more likely that one domain assumes another, the optimizer might also compare the HIGH2KEY AND LOW2KEY statistics for these index columns.

After creating appropriate indexes, ensure that statistics on tables are accurate and up to date.

The optimizer uses the information in the FIRSTnKEYCARD and FULLKEYCARD columns of the unique index statistics table to detect cases of correlation, and dynamically adjust combined selectivities of the correlated predicates, thus obtaining a more accurate estimate of the join size and cost.

As an alternative, column group statistics can be collected on a set of columns. In the elasticity example above, you might gather statistics on the columns PRODUCT.COLOR, PRODUCT.ELASTICITY and/or RAWMATERIAL.COLOR, RAWMATERIAL.ELASTCITY.

Column group statistics are collected using the ″ON COLUMNS″ option of RUNSTATS. For example, 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))
```

Correlation of simple equal predicates

In addition to JOIN predicate correlation, the optimizer also manages correlation with simple equal 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, sports-utility vehicle. Because almost every manufacturer makes the same standard colors available on 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, the predicates MAKE and MODEL certainly are not independent since 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 and certainly not wanted by the car makers.

If an index exists on the two columns MAKE and MODEL or column group statistics are gathered, the optimizer uses the statistical information about the index or columns to determine the combined number of distinct values and adjust the

selectivity or cardinality estimation for correlation between the two columns. If the predicates are local equality predicates, the optimizer does not need a unique index to make the adjustment.

Using index and column group statistics to compute grouping keycard

When a query requires data to be grouped in a certain way, the optimizer needs to compute the number of distinct groupings, or *grouping keycard*. A grouping requirement can result from operations such as GROUP BY or DISTINCT.

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 number of groupings (also in this case the number of rows returned) estimated by the optimizer will be the product of the number of distinct values of DEPTNO, MGR, and YEAR_HIRED. This estimate assumes that the grouping key columns are independent. However, this assumption could be erroneous if each manager manages exactly one department. Also, it is unlikely that each department has employees hired 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.

Now, consider an index with the following definition:

```
INDEX IX1:  DEPTNO, MGR, YEAR_HIRED
```

In this case, the FULLKEYCARD of IX1 provides the optimizer with the exact number of distinct groupings for the query above.

Consider another index definition:

```
INDEX IX2:  DEPTNO, MGR, YEAR_HIRED, COMM
```

IX2 could also help to compute the grouping keycard, since its FIRST3KEYCARD indicates how many distinct groups of (DEPTNO,MGR,YEAR_HIRED).

Besides index statistics (FIRST2KEYCARD, FIRST3KEYCARD, FIRST4KEYCARD, and FULLKEYCARD), column group statistics could also be exploited by the optimizer to compute grouping keycard in a similar fashion. Column group statistics collected on DEPTNO, MGR, and YEAR_HIRED will provide the same benefit as IX1 and IX2 above:

```
RUNSTATS ON TABLE EMPLOYEE ON COLUMNS ((DEPTNO, MGR, YEAR_HIRED))
```

Note that if the grouping keys consist of five or more columns, then collecting column group statistics might be preferable. This is because RUNSTATS only collects statistics on the first four columns and the full index key columns of any given index.

Statistical views

Statistical views allow the optimizer to compute more accurate cardinality estimates. Cardinality estimation is the process whereby the optimizer uses statistics to determine the size of partial query results after predicates are applied or aggregation is performed. The accuracy of cardinality estimates depends on the predicates and the available statistics. Statistics are available to represent the

distribution of data values within a column, which can improve cardinality estimates when the data values are unevenly distributed. Statistics are also available to represent the number of distinct values in a set of columns, which can improve cardinality estimates when columns are statistically correlated. However, quite often, these statistics may not be able to represent more complex relationships such as the filtering effect of predicates or aggregation involving correlated and skewed attributes (for example, *make* = 'Honda' AND *model* = 'Accord'), comparisons with expression results (for example, *price* > MSRP + Dealer_markup), relationships spanning multiple tables (for example *product.name* = 'Alloy wheels' and *product.key* = *sales.product_key*), or anything other than predicates or aggregation involving independent attributes and simple comparison operations.

Statistical views are views with associated statistics that can be used to improve cardinality estimates for queries in which the view definition overlaps with the query definition. This is a powerful feature in that it provides the optimizer with accurate statistics for determining cardinality estimates for queries with complex sets of (possibly correlated) predicates involving one or more tables.

A statistical view need not be directly referenced in a query it optimizes; in most cases, a view's statistics can be exploited if its definition overlaps with the query definition. To exploit this new feature, a view must be enabled for optimization using the ALTER VIEW statement and system catalog tables must be populated for statistics on the view.

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 a *statistical view*. A view that is not a statistical view is said to be disabled for optimization. The term *regular view* is used to refer to a view that is disabled for optimization. A view is disabled for optimization when it is first created.

- To enable a view for optimization you must have ALTER privilege on both the view and the table on which the view is defined.
- To invoke RUNSTATS for a view you must have one of the following:
  - SYSADM
  - SYSCTRL
  - SYSMAINT
  - DBADM
  - CONTROL privilege on the view

  In addition, you need to have appropriate privileges to access rows from the view. Specifically, for each table, view or nickname referenced in the view definition, you must have one of the following privileges:
  - SYSADM
  - DBADM
  - CONTROL
  - SELECT

A view cannot be enabled for optimization if any of the following are true:

- The view directly or indirectly references an MQT. (An MQT or statistical view can reference a statistical view.)
- It is an inoperative view.

- It is a typed view.
- There is another view alteration in the same ALTER VIEW statement.

If the definition of a view that is altered to enable optimization meets any of the conditions below, the ALTER VIEW ENABLE OPTIMIZATION will succeed with a warning, but the optimizer will not exploit its statistics:

- It contains aggregation or distinct operations.
- It contains union, except, or intersect operations.
- It contains scalar aggregate (OLAP) functions.

1. Enable the view for optimization.

   A view can be enabled for optimization using the new 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 on the ALTER VIEW statement. For example, to enable the view `myview` for optimization, enter the following:

   ```
   ALTER VIEW myview ENABLE QUERY OPTIMIZATION
   ```

   A view that is enabled for optimization has a 'Y' in character position 13 of the PROPERTY column of its corresponding SYSTABLES entry. A view that is disabled for optimization has a blank in character position 13 of the PROPERTY column of its corresponding SYSTABLES entry.
2. Execute RUNSTATS. For example, to collect statistics on the view `myview`, enter the following:

   ```
   RUNSTATS ON TABLE db2dba.myview
   ```

   To use statistics sampling to collect view statistics, including distribution statistics, on 10 percent of the rows using row-level sampling, enter:

   ```
   RUNSTATS ON TABLE db2dba.myview WITH DISTRIBUTION TABLESAMPLE BERNOULLI (10)
   ```

   Note: Prior to DB2 Version 9.1, executing RUNSTATS on a statistical view only primed the catalog statistics tables for manual updates and did not collect any statistics. In DB2 Version 9.1, executing RUNSTATS on a statistical view will collect statistics. This means that RUNSTATS may take much longer to execute than previously, depending on how much data is returned by the view.
3. Updating view statistics can result in changes to the plans for queries that overlap the view definition. If these queries are part of static SQL packages, these packages must be rebound to take advantages of the plans resulting from the new statistics.

View statistics relevant to optimization

Only statistics that characterize the data distribution of the query defining a statistical view, such as CARD and COLCARD, are considered for optimization by the optimizer. The following statistics associated with view records can be collected and can be exploited by the optimizer.

Table statistics (SYSCAT.TABLES, SYSSTAT.TABLES)

- CARD - The number of rows in the view result.

Column statistics (SYSCAT.COLUMNS, SYSSTAT.COLUMNS)

- COLCARD - The number of distinct values of a column in the view result
- AVGCOLLEN - Average length of 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 NULLs in a view result column
- SUB_COUNT - Average number of sub-elements in a view result column
- SUB_DELIM_LENGTH - Average length of each delimiter separating each sub-element

Column Distribution Statistics (SYSCAT.COLDIST, SYSSTAT.COLDIST)

- DISTCOUNT - If TYPE is Q, the number of distinct values that are less than or equal to COLVALUE statistics
- SEQNO - Frequency ranking of a sequence number to help uniquely identify the row in the table
- COLVALUE - Data value for which frequency or quantile statistic is collected
- VALCOUNT - Frequency with which the data value occurs in view column, or for quantiles, the number of values 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 will be ignored by the optimizer.

Scenario: Improving cardinality estimates using statistical views

In a data warehouse, fact table information often changes quite dynamically, while 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 which 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 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. In order 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. Since 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 represent 1% of the products manufactured historically, but accounts now for 20% of recent sales, the optimizer would likely overestimate the selectivity of *item_desc = 'golf club'*, as 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.

The following is a database from a typical data warehouse, where STORE, CUSTOMER, PRODUCT, PROMOTION, and PERIOD are the dimension tables, and DAILY_SALES the fact table.

Table 66. STORE (63 rows)

Column	storekey	store_number	city	state	district	...
Attribute	integer	char(2)	char(20)	char(5)	char(14)	...
	not null					
	primary key					

Table 67. CUSTOMER (1,000,000 rows)

Column	custkey	name	address	age	gender	...
Attribute	integer	char(30)	char(40)	smallint	char(1)	...
	not null					
	primary key					

Table 68. PRODUCT (19,450 rows)

Column	prodkey	category	item_desc	price	cost	...
Attribute	integer	integer	char(30)	decimal(11)	decimal(11)	...
	not null					
	primary key					

Table 69. PROMOTION (35 rows)

Column	promokey	promotype	promodesc	promovalue	...
Attribute	integer	integer	char(30)	decimal(5)	...
	not null				
	primary key				

Table 70. PERIOD (2922 rows)

Column	perkey	calendar_date	month	period	year	...
Attribute	integer	date	char(3)	smallint	smallint	...
	not null					
	primary key					

Table 71. 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 wish to find out whether or not consumers will buy a product again if they are offered a discount on the same product on a return visit. In addition, suppose the study is only done for store '01', which has 18 locations nationwide. Table 72 shows the different categories of promotion available, annotated by the percentage of promotions.

Table 72. PROMOTION (35 rows)

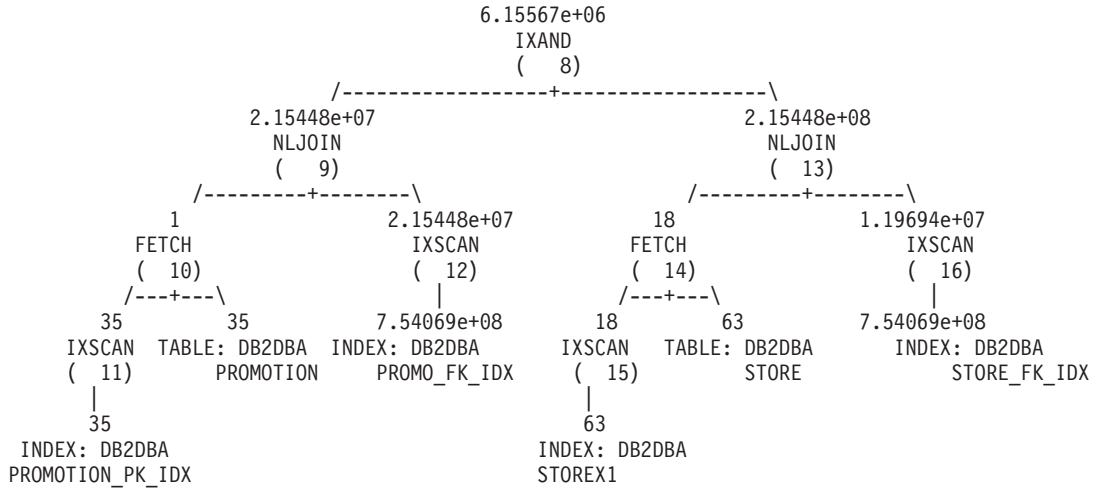
promotype	promodesc	COUNT(promotype)	percentage of promotions
1	Return customers	1	2.86%
2	Coupon	15	42.86%
3	Advertisement	5	14.29%
4	Manager's special	3	8.57%
5	Overstocked items	4	11.43%
6	End aisle display	7	20.00%

The table indicates that only 2.86% of all 35 kinds of promotions offered comes from discounts for return customers (1÷35 ≈ 0.0286) .

The following query is then executed and a count of 12,889,514 is returned:

```
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 is executed according to the following plan generated by the optimizer. In each node of this diagram, the top number represents the cardinality estimate, the second row is the operator type, and 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 same percentage before and after joining the PROMOTION table with the DAILY_SALES

table. Table 73 summarizes the cardinality estimates and their percentage (the filtering effect) before and after the join.

Table 73. Cardinality estimates before and after joining with DAILY_SALES.

	Before Join		After Join	
Predicates	**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 off by a factor of 2.09 (12,889,514 ÷ 6,155,670 ≈ 2.09).

What causes the optimizer to only be able to estimate half of the actual number of records qualifying 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 73 can be misleading. The actual percentage in DAILY_SALES could very well be a different figure than the projected one.

We can use statistical views to help the optimizer correct its estimates and reduce overestimation or underestimation. First, we need to create two statistical views representing each semi-join in the query above. The first statistical view provides the distribution of promotion types 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, we use a 10% sample rate to retrieve the records in DAILY_SALES for the respective views and save them in global temporary tables. We 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)
```

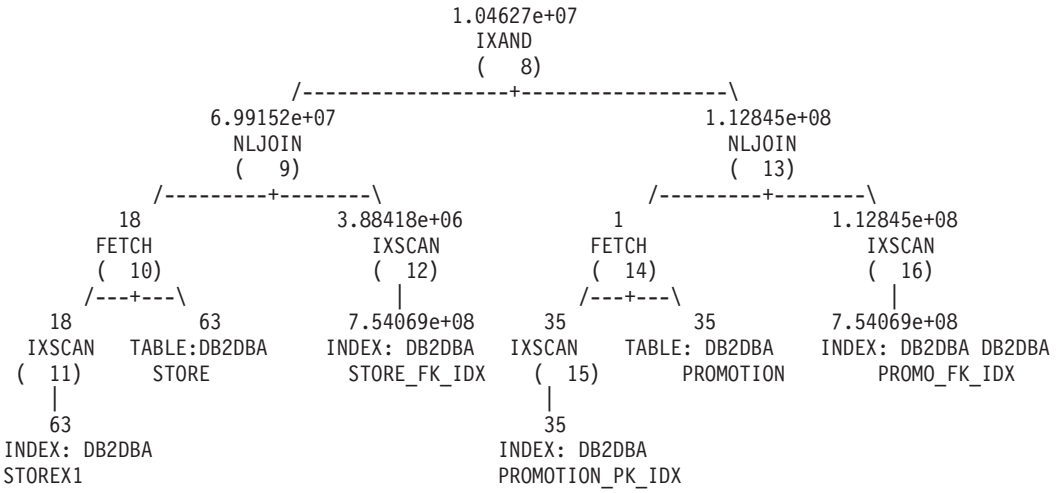
2. Create a view representing the join of PROMOTION with DAILY_SALES.

```
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 RUNSTATS 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 it to reverse the original order of the semi-joins chosen without these statistics. The new plan is as follows:



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. Table 74 summarizes the cardinality estimates and their percentage (filtering effect) before and after the join for each semi-join.

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. Here, 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 in Table 73 on page 412. Table 74 summarizes the cardinality estimates and the filtering effect for each predicate before and after the join.

Table 74. Cardinality estimates before and after joining with DAILY_SALES.

	Before Join		**After Join** *no statistical views*		**After Join** *with statistical views*	
Predicates	**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%

This time, the optimizer estimates that there are approximately 10,462,700 products sold. This estimate is off by a factor of 1.23, which is a significant improvement over the estimate without statistical views.

Part 6. Appendixes

Appendix A. Overview of the DB2 technical information

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics (Task, concept and reference topics)
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF DVD)
  - printed books
- Command line help
  - Command help
  - Message help

Note: The DB2 Information Center topics are updated more frequently than either the PDF or the hard-copy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com®.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks® publications online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how to improve the DB2 documentation, send an email to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this email address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/shop/publications/order. English DB2 Version 9.5 manuals in PDF format and translated versions can be downloaded from www.ibm.com/support/docview.wss?rs=71&uid=swg2700947.

Although the tables identify books available in print, the books might not be available in your country or region.

Table 75. DB2 technical information

Name	Form Number	Available in print
Administrative API Reference	SC23-5842-00	Yes
Administrative Routines and Views	SC23-5843-00	No
Call Level Interface Guide and Reference, Volume 1	SC23-5844-00	Yes
Call Level Interface Guide and Reference, Volume 2	SC23-5845-00	Yes
Command Reference	SC23-5846-00	Yes
Data Movement Utilities Guide and Reference	SC23-5847-00	Yes
Data Recovery and High Availability Guide and Reference	SC23-5848-00	Yes
Data Servers, Databases, and Database Objects Guide	SC23-5849-00	Yes
Database Security Guide	SC23-5850-00	Yes
Developing ADO.NET and OLE DB Applications	SC23-5851-00	Yes
Developing Embedded SQL Applications	SC23-5852-00	Yes
Developing Java Applications	SC23-5853-00	Yes
Developing Perl and PHP Applications	SC23-5854-00	No
Developing User-defined Routines (SQL and External)	SC23-5855-00	Yes
Getting Started with Database Application Development	GC23-5856-00	Yes
Getting Started with DB2 installation and administration on Linux and Windows	GC23-5857-00	Yes
Internationalization Guide	SC23-5858-00	Yes
Message Reference, Volume 1	GI11-7855-00	No
Message Reference, Volume 2	GI11-7856-00	No
Migration Guide	GC23-5859-00	Yes
Net Search Extender Administration and User's Guide **Note:** The content of this document is not included in the DB2 Information Center	SC23-8509-00	Yes
Partitioning and Clustering Guide	SC23-5860-00	Yes
Query Patroller Administration and User's Guide	SC23-8507-00	Yes
Quick Beginnings for IBM Data Server Clients	GC23-5863-00	No
Quick Beginnings for DB2 Servers	GC23-5864-00	Yes

Table 75. DB2 technical information (continued)

Name	Form Number	Available in print
Spatial Extender and Geodetic Data Management Feature User's Guide and Reference	SC23-8508-00	Yes
SQL Reference, Volume 1	SC23-5861-00	Yes
SQL Reference, Volume 2	SC23-5862-00	Yes
System Monitor Guide and Reference	SC23-5865-00	Yes
Text Search Guide	SC23-5866-00	Yes
Troubleshooting Guide	GI11-7857-00	No
Tuning Database Performance	SC23-5867-00	Yes
Visual Explain Tutorial	SC23-5868-00	No
What's New	SC23-5869-00	Yes
Workload Manager Guide and Reference	SC23-5870-00	Yes
pureXML Guide	SC23-5871-00	Yes
XQuery Reference	SC23-5872-00	No

Table 76. DB2 Connect-specific technical information

Name	Form Number	Available in print
Quick Beginnings for DB2 Connect Personal Edition	GC23-5839-00	Yes
Quick Beginnings for DB2 Connect Servers	GC23-5840-00	Yes
DB2 Connect User's Guide	SC23-5841-00	Yes

Table 77. Information Integration technical information

Name	Form Number	Available in print
Information Integration: Administration Guide for Federated Systems	SC19-1020-01	Yes
Information Integration: ASNCLP Program Reference for Replication and Event Publishing	SC19-1018-02	Yes
Information Integration: Configuration Guide for Federated Data Sources	SC19-1034-01	No
Information Integration: SQL Replication Guide and Reference	SC19-1030-01	Yes
Information Integration: Introduction to Replication and Event Publishing	SC19-1028-01	Yes

Ordering printed DB2 books

If you require printed DB2 books, you can buy them online in many but not all countries or regions. You can always order printed DB2 books from your local IBM representative. Keep in mind that some softcopy books on the *DB2 PDF Documentation* DVD are unavailable in print. For example, neither volume of the *DB2 Message Reference* is available as a printed book.

Printed versions of many of the DB2 books available on the DB2 PDF Documentation DVD can be ordered for a fee from IBM. Depending on where you are placing your order from, you may be able to order books online, from the IBM Publications Center. If online ordering is not available in your country or region, you can always order printed DB2 books from your local IBM representative. Note that not all books on the DB2 PDF Documentation DVD are available in print.

Note: The most up-to-date and complete DB2 documentation is maintained in the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2luw/v9r5.

To order printed DB2 books:

- To find out whether you can order printed DB2 books online in your country or region, check the IBM Publications Center at http://www.ibm.com/shop/publications/order. You must select a country, region, or language to access publication ordering information and then follow the ordering instructions for your location.
- To order printed DB2 books from your local IBM representative:
  1. Locate the contact information for your local representative from one of the following Web sites:
     - The IBM directory of world wide contacts at www.ibm.com/planetwide
     - The IBM Publications Web site at http://www.ibm.com/shop/publications/order. You will need to select your country, region, or language to the access appropriate publications home page for your location. From this page, follow the "About this site" link.
  2. When you call, specify that you want to order a DB2 publication.
  3. Provide your representative with the titles and form numbers of the books that you want to order. For titles and form numbers, see "DB2 technical library in hardcopy or PDF format" on page 417.

Displaying SQL state help from the command line processor

DB2 returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

To invoke SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

For example, `? 08003` displays help for the 08003 SQL state, and `? 08` displays help for the 08 class code.

Accessing different versions of the DB2 Information Center

For DB2 Version 9.5 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/

For DB2 Version 9 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9/

For DB2 Version 8 topics, go to the Version 8 Information Center URL at: http://publib.boulder.ibm.com/infocenter/db2luw/v8/

Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

- To display topics in your preferred language in the Internet Explorer browser:
  1. In Internet Explorer, click the **Tools** —> **Internet Options** —> **Languages...** button. The Language Preferences window opens.
  2. Ensure your preferred language is specified as the first entry in the list of languages.
     - To add a new language to the list, click the **Add...** button.

       Note: Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.
     - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
  3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.
- To display topics in your preferred language in a Firefox or Mozilla browser:
  1. Select the button in the **Languages** section of the **Tools** —> **Options** —> **Advanced** dialog. The Languages panel is displayed in the Preferences window.
  2. Ensure your preferred language is specified as the first entry in the list of languages.
     - To add a new language to the list, click the **Add...** button to select a language from the Add Languages window.
     - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
  3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

On some browser and operating system combinations, you might have to also change the regional settings of your operating system to the locale and language of your choice.

Updating the DB2 Information Center installed on your computer or intranet server

If you have installed the DB2 Information Center locally, you can download and install updates that IBM might make available.

Updating your locally-installed DB2 Information Center requires that you:

1. Stop the DB2 Information Center on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to download and apply updates.
2. Use the Update feature to see what updates are available. If there are updates that you would like to install, you can use the Update feature to download and install them

   Note: If your environment requires installing the DB2 Information Center updates on a machine that is not connected to the internet, you have to mirror the update site to a local file system using a machine that is connected to the internet and has the DB2 Information Center installed. If many users on your network will be installing the documentation updates, you can reduce the time required for individuals to perform the updates by also mirroring the update site locally and creating a proxy for the update site.
   If update packages are available, use the Update feature to download the packages. However, the Update feature is only available in stand-alone mode.
3. Stop the stand-alone Information Center, and restart the DB2 Information Center on your computer.

Note: On Windows Vista, the commands listed below must be run as an administrator. To launch a command prompt or graphical tool with full administrator privileges, right-click on the shortcut and then select **Run as administrator**.

To update the DB2 Information Center installed on your computer or intranet server:

1. Stop the DB2 Information Center.
   - On Windows, click **Start → Control Panel → Administrative Tools → Services**. Then right-click on **DB2 Information Center** service and select **Stop**.
   - On Linux, enter the following command:
     ```
     /etc/init.d/db2icdv95 stop
     ```
2. Start the Information Center in stand-alone mode.
   - On Windows:
     a. Open a command window.
     b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the <Program Files>\IBM\DB2 Information Center\Version 9.5 directory, where <Program Files> represents the location of the Program Files directory.
     c. Navigate from the installation directory to the doc\bin directory.
     d. Run the help_start.bat file:
        ```
        help_start.bat
        ```
   - On Linux:

a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the /opt/ibm/db2ic/V9.5 directory.
b. Navigate from the installation directory to the doc/bin directory.
c. Run the help_start script:

```
help_start
```

The systems default Web browser launches to display the stand-alone Information Center.

3. Click the Update button (). On the right hand panel of the Information Center, click Find Updates. A list of updates for existing documentation displays.
4. To initiate the download process, check the selections you want to download, then click Install Updates.
5. After the download and installation process has completed, click Finish.
6. Stop the stand-alone Information Center.
   - On Windows, navigate to the installation directory's doc\bin directory, and run the help_end.bat file:

     ```
     help_end.bat
     ```

     Note: The help_end batch file contains the commands required to safely terminate the processes that were started with the help_start batch file. Do not use `Ctrl-C` or any other method to terminate help_start.bat.
   - On Linux, navigate to the installation directory's doc/bin directory, and run the help_end script:

     ```
     help_end
     ```

     Note: The help_end script contains the commands required to safely terminate the processes that were started with the help_start script. Do not use any other method to terminate the help_start script.
7. Restart the DB2 Information Center.
   - On Windows, click **Start → Control Panel → Administrative Tools → Services**. Then right-click on **DB2 Information Center** service and select **Start**.
   - On Linux, enter the following command:

     ```
     /etc/init.d/db2icdv95 start
     ```

The updated DB2 Information Center displays the new and updated topics.

DB2 tutorials

The DB2 tutorials help you learn about various aspects of DB2 products. Lessons provide step-by-step instructions.

Before you begin

You can view the XHTML version of the tutorial from the Information Center at http://publib.boulder.ibm.com/infocenter/db2help/.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

DB2 tutorials

To view the tutorial, click on the title.

"pureXML™" in *pureXML Guide*
Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

"Visual Explain" in *Visual Explain Tutorial*
Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 products.

DB2 documentation
Troubleshooting information can be found in the DB2 Troubleshooting Guide or the Support and Troubleshooting section of the DB2 Information Center. There you will find information on how to isolate and identify problems using DB2 diagnostic tools and utilities, solutions to some of the most common problems, and other advice on how to solve problems you might encounter with your DB2 products.

DB2 Technical Support Web site
Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.

Access the DB2 Technical Support Web site at http://www.ibm.com/software/data/db2/udb/support.html

Terms and Conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal use: You may reproduce these Publications for your personal, non commercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these Publications, or any portion thereof, without the express consent of IBM.

Commercial use: You may reproduce, distribute and display these Publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these Publications, or reproduce, distribute or display these Publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the Publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the Publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Appendix B. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

This document may provide links or references to non-IBM Web sites and resources. IBM makes no representations, warranties, or other commitments whatsoever about any non-IBM Web sites or third-party resources that may be referenced, accessible from, or linked from this document. A link to a non-IBM Web site does not mean that IBM endorses the content or use of such Web site or

its owner. In addition, IBM is not a party to or responsible for any transactions you may enter into with third parties, even if you learn of such parties (or use a link to such parties) from an IBM site. Accordingly, you acknowledge and agree that IBM is not responsible for the availability of such external sites or resources, and is not responsible or liable for any content, services, products, or other materials on or available from those sites or resources. Any software provided by third parties is subject to the terms and conditions of the license that accompanies that software.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information that has been exchanged, should contact:

IBM Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs, in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

Trademarks

Company, product, or service names identified in the documents of the DB2 Version 9.5 documentation library may be trademarks or service marks of International Business Machines Corporation or other companies. Information on the trademarks of IBM Corporation in the United States, other countries, or both is located at http://www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 documentation library:

Microsoft®, Windows, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel®, Intel logo, Intel Inside® logo, Intel Centrino®, Intel Centrino logo, Celeron®, Intel Xeon®, Intel SpeedStep®, Itanium® and Pentium® are trademarks of Intel Corporation in the United States, other countries, or both.

Java™ and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Adobe®, the Adobe logo, PostScript®, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

A

B

C

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

SC23-5867-00

Spine information:

DB2 Version 9.5 for Linux, UNIX, and Windows

Tuning Database Performance