IBM® DB2® for Linux®, UNIX®, and Windows®

Best Practices

Writing and Tuning Queries for Optimal Performance

John Hornibrook

DB2 Query Optimization Development

Nina Kolunovsky

DB2 Information Development

Executive summary 4
Introduction 5
Writing SQL statements 6
Avoid complex expressions in search conditions 6
Avoid join predicates on expressions 6
Avoid expressions over columns in local predicates 6
Avoid data type mismatches on join columns 8
Do not use no-op expressions in predicates to change the optimizer estimate 8
Avoid non-equality join predicates 9
Avoid multiple aggregations with the DISTINCT keyword 10
Avoid unnecessary outer joins 12
Use OPTIMIZE FOR N ROWS clause with FETCH FIRST N ROWS ONLY clause 13
If you are using the star schema join, ensure your queries fit the required criteria 13
Avoid redundant predicates 14
Designing and configuring your database 16
Use Constraints to Improve Query Optimization 16
Use the REOPT bind option with input variables in complex queries 17
Choose the best optimization class for your workload 19
Use parameter markers to reduce compilation time for dynamic queries 21
Set DB2_REDUCED_OPTIMIZATION registry variable 21
Gather accurate catalog statistics, including advanced statistics features 22
Column group statistics 22
Sub-element statistics 24
Statistical views 25

Minimize RUNSTATS impact 26
Avoid updating catalog statistics manually 27
Use optimization profiles if other tuning options do not produce acceptable results 28
Tuning SQL statements using the explain facility 29
Analyzing performance changes 29
Evaluating performance tuning efforts 30
Best Practices 31
Conclusion 33
Further reading 35
Contributors 36
Notices 37
Trademarks 38

Executive summary

This document describes the best practices for minimizing the impact of SQL statements on DB2 database performance. You can minimize this impact in several ways:

- By writing SQL statements that the DB2 optimizer can more easily optimize. The DB2 optimizer might not be able to efficiently run SQL statements that contain non-equality join predicates, data type mismatches on join columns, unnecessary outer joins, and other complex search conditions.

- By correctly configuring the DB2 database to take advantage of DB2 optimization functionality. The DB2 optimizer can select the optimal query access plan if you have accurate catalog statistics and choose the best optimization class for your workload.

- By using the DB2 explain functionality to review potential query access plans and determine how to tune queries for best performance.

This document includes best practices that apply to general workloads, warehouse workloads, and SAP workloads.

There are a number of ways to deal with specific query performance issues after an application is written. However, this document focuses on good fundamental writing and tuning practices that can be widely applied to help improve DB2 database performance.

If you follow the recommendations discussed in this document and still experience poor query performance, there are a number of techniques available to understand why. The "Tuning and Monitoring Database System Performance" best practices paper describes a number of techniques for identifying performance problems and ways the system can be configured to help prevent them. The "Physical Database Design" best practices paper describes how to use DB2 database system features such as multi-dimensional clustering (MDC), materialized query tables (MQTs), and the DB2 Design Advisor to achieve optimal query performance. A subsequent best practices paper will describe techniques to analyze performance problems with a specific query.

For suggestions on improving XQuery performance, see the "Managing XML Data" best practices paper.

Introduction

Query performance is not a one-time consideration. You should consider it throughout the design, development, and production phases of the application development life cycle.

SQL is a very flexible language, which means that there are many ways to get the same correct result. This flexibility also means that some queries are better than others in taking advantage of the DB2 optimizer's strengths.

During query execution, the DB2 optimizer chooses a query access plan for each SQL statement. The optimizer models the execution cost of many alternative access plans and chooses the one with the minimal estimated cost. If a query contains many complex search conditions, the DB2 optimizer can rewrite the predicate in some cases, but there are some cases where it cannot.

The time to prepare or compile an SQL statement can be long for complex queries, such as those used in BI applications. You can help minimize statement compilation time by correctly designing and configuring your database. This includes choosing the correct optimization class and setting other registry variables correctly.

The optimizer also requires accurate inputs to make accurate access plan decisions. This means that you need to gather accurate statistics, and potentially use advanced statistical features, such as statistical views and column group statistics.

You can use the DB2 tools, especially the DB2 explain facility, to tune queries. The DB2 compiler can capture information about the access plans and environments of static or dynamic queries. Use this captured information to understand how individual statements are run so that you can tune them and your database manager configuration to improve performance.

Writing SQL statements

SQL is a powerful language that enables you to specify relational expressions in syntactically different but semantically equivalent ways. However, some semantically equivalent variations are easier to optimize than others. Although the DB2 optimizer has a powerful query rewrite capability, it might not always be able to rewrite an SQL statement into the most optimal form. There are also certain SQL constructs that can limit the access plans considered by the query optimizer. The following sections describe certain SQL constructs that should be avoided and provide suggestions for how to replace or avoid them.

Avoid complex expressions in search conditions

Avoid using complex expressions in search conditions where the expressions prevent the optimizer from using the catalog statistics to estimate an accurate selectivity. The expressions might also limit the choices of access plans that can be used to apply the predicate. During the query rewrite phase of optimization, the optimizer can rewrite a number of expressions to allow the optimizer to estimate an accurate selectivity; it cannot handle all possibilities.

Avoid join predicates on expressions

Using join predicates on expressions limits the join method to nested loops. Additionally, the cardinality estimate might be inaccurate. Some examples of joins with expressions are as follows:

```
WHERE SALES.PRICE * SALES.DISCOUNT = TRANS.FINAL_PRICE

WHERE UPPER(CUST.LASTNAME) = TRANS.NAME
```

Avoid expressions over columns in local predicates

Instead of applying an expression over columns in a local predicate, use the inverse of the expression. Consider the following examples:

```
XPRESSN(C) = 'constant'

INTEGER(TRANS_DATE)/100 = 200802
```

You can rewrite these statements as follows:

```
C = INVERSEXPRESSN('constant')

TRANS_DATE BETWEEN 20080201 AND 20080229
```

Applying expressions over columns prevents the use of index start and stop keys, leads to inaccurate selectivity estimates, and requires extra processing at query execution time.

These expressions also prevent query rewrite optimizations such as recognizing when columns are equivalent, replacing columns with constants, and recognizing when at most one row will be returned. Further optimizations are possible after it can be proven that at most one row will be returned, so the lost optimization opportunities are further compounded. Consider the following query:

```
SELECT LASTNAME, CUST_ID, CUST_CODE FROM CUST

WHERE (CUST_ID * 100) + INT(CUST_CODE) = 123456 ORDER BY 1,2,3
```

You can rewrite it as follows:

```
SELECT LASTNAME, CUST_ID, CUST_CODE FROM CUST

WHERE CUST_ID = 1234 AND CUST_CODE = '56' ORDER BY 1,2,3
```

If there is a unique index defined on CUST_ID, the rewritten version of the query enables the query optimizer to recognize that at most one row will be returned. This avoids introducing an unnecessary SORT operation. It also enables the CUST_ID and CUST_CODE columns to be replaced by 1234 and '56', avoiding copying values from the data or index pages. Finally, it enables the predicate on CUST_ID to be applied as an index start or stop key.

It might not always be apparent when an expression is present in a predicate. This can often occur with queries that reference views when the view columns are defined by expressions. For example, consider the following view definition and query:

```
CREATE VIEW CUST_V AS

(SELECT LASTNAME, (CUST_ID * 100) + INT(CUST_CODE) AS CUST_KEY
FROM CUST)
```

```
SELECT LASTNAME FROM CUST_V WHERE CUST_KEY = 123456
```

The query optimizer merges the query with the view definition, resulting in the following query:

```
SELECT LASTNAME FROM CUST

WHERE (CUST_ID * 100) + INT(CUST_CODE) = 123456
```

This is the same problematic predicate described in a previous example. You can observe the result of view merging by using the explain facility to display the optimized SQL.

If the inverse function is difficult to express, consider using a generated column. For example, if you want to find a last name that fits the criteria expressed by LASTNAME

IN ('Woo', 'woo', 'WOO', 'WOo', and so on), you can create a generated column UCASE(LASTNAME) = 'WOO'[1] , as follows:

```
CREATE TABLE CUSTOMER

(LASTNAME VARCHAR(100),

U_LASTNAME VARCHAR(100) GENERATED ALWAYS AS (UCASE(LASTNAME)))

CREATE INDEX CUST_U_LASTNAME ON CUSTOMER(U_LASTNAME)
```

Avoid data type mismatches on join columns

In some cases, data type mismatches prevent the use of hash joins. Hash join has some extra restrictions on the join predicates beyond other join methods. In particular, the data types of the join columns must be exactly the same. For example, if one join column is FLOAT and the other is REAL, hash join is not supported. Additionally, if the join column data type is CHAR, GRAPHIC, DECIMAL or DECFLOAT the lengths must be the same.

Do not use no-op expressions in predicates to change the optimizer estimate

A "no-op" coalesce() predicate of the form "COALESCE(X, X) = X" introduces an estimation error into the planning of any query using it. Currently the DB2 query compiler does not have the capability of dissecting that predicate and determining that all rows actually satisfy it. As a result, the predicate artificially reduces the estimated number of rows coming from some part of a query plan. This smaller row estimate usually reduces the row and cost estimates for the rest of query planning, and sometimes results in a different plan being chosen because relative estimates between different candidate plans have changed.

Why can this do-nothing predicate sometimes improve query performance? The addition of the "no-op" coalesce() predicate introduces an error that masks something else that is preventing optimal performance.

What some performance enhancement tools do is a brute-force test: the tool repeatedly introduces the predicate into different places in a query, operating on different columns, to try to find a case where, by introducing an error, it stumbles onto a better-performing plan. This is also true of a query developer hand-coding the "no-op" predicate into a query. Typically, the developer will have some insight on the data to guide the placement of the predicate.

---

[1] Support for case-insensitive search in DB2 Database for Linux, UNIX, and Windows V9.5 Fix Pack 1 is designed to resolve the situation in this particular example. You can use _S*x* attribute on the UCA500R1 collation name to control the strength of the collations. For example, UCA500R1_LFR_S1 is a French collation that ignores case and accent.
See the “Tuning and Monitoring Database System Performance” best practices paper for more details.

Using this method to improve query performance is a short-term solution which does not address root cause and might have the following implications:

- Potential areas for performance improvements are hidden.
- There are no guarantees that this workaround will provide permanent performance improvements as the DB2 query compiler might eventually handle the predicate better or other random factors might affect it.
- There might be other queries that are affected by the same root cause and the performance of your system in general might suffer as a result.

Avoid non-equality join predicates

Join predicates that use comparison operators other than equality should be avoided because the join method is limited to nested loop. Additionally, the optimizer might not be able to compute an accurate selectivity estimate for the join predicate. However, non-equality join predicates cannot always be avoided. When they are necessary, ensure that an appropriate index exists on either table because the join predicates will be applied on the nested loop join inner.

One common example of non-equality join predicates is when dimension data in a star schema must be versioned, in order to accurately reflect the state of a dimension at different points in time. This is often referred to as a 'slowly changing dimension'. One type of slowly changing dimension involves including effective start and end dates for each dimension row. A join between the fact table and the dimension table requires checking that a date associated with the fact, falls within the dimension's start and end date, in addition to joining on the dimension primary key. This is often referred to as a 'type 6 slowly changing dimension'. The range join back to the fact table to further qualify the dimension version by some fact transaction date, can be expensive. For example,

```
SELECT ... FROM

PRODUCT P, SALES F

WHERE

P.PROD_KEY = F.PROD_KEY AND F.SALE_DATE BETWEEN P.START_DATE AND
P.END_DATE
```

In this situation, ensure there is an index on (F.PROD_KEY, F.SALE_DATE).

Consider creating a statistical view to help the optimizer compute a better selectivity estimate for this scenario. For example,

```
CREATE STATISTICAL VIEW V_PROD_FACT AS

SELECT P.*
```

```
FROM PRODUCT P, SALES F

WHERE

P.PROD_KEY=F.PROD_KEY and

F.SALE_DATE BETWEEN P.START_DATE AND P.END_DATE


ALTER VIEW V_PROD_FACT ENABLE QUERY OPTIMIZATION


RUNSTATS ON TABLE DB2USER.V_PROD_FACT WITH DISTRIBUTION
```

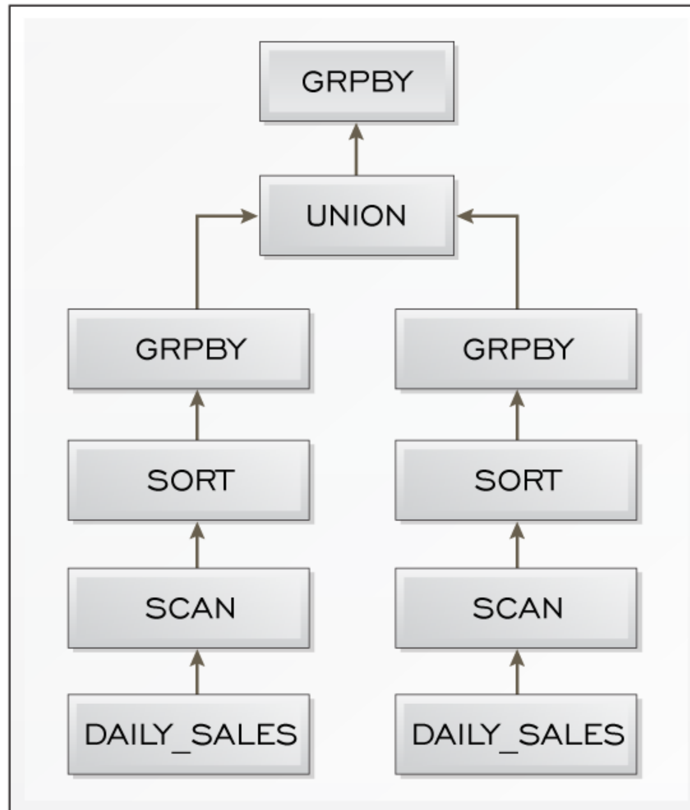
Specialized star schema joins such as star join with index ANDing and hub joins are not considered if there are any non-equality join predicates in the query block. (Refer to "If you are using the star schema join, ensure your queries fit the required criteria" on page 13.)

Avoid multiple aggregations with the DISTINCT keyword

Avoid using queries that perform multiple DISTINCT aggregations in the same subselect, which are expensive to run. Consider the following example:

```
SELECT SUM(DISTINCT REBATE), AVG(DISTINCT DISCOUNT) FROM
DAILY_SALES GROUP BY PROD_KEY;
```

To determine the set of distinct REBATE values and distinct DISCOUNT values, the input stream from the PROD_KEY table might need to be sorted twice. The query access plan for this query might look like this:



The optimizer rewrites the original query into separate aggregations, for each specifying DISTINCT keyword, and then combines the multiple aggregations using a UNION keyword. The internally rewritten statement is:

```
SELECT Q8.MAXC0, (Q8.MAXC1 / Q8.MAXC2)

FROM

   (SELECT MAX(Q7.C0) AS MAXC0, MAX(Q7.C1) AS MAXC1, MAX(Q7.C2)
AS MAXC2

   FROM

      (SELECT SUM(DISTINCT Q2.REBATE) as C0 , cast(NULL as
integer) AS C1, 0 AS C2, Q2.PROD_KEY

      FROM

         (SELECT Q1.PROD_KEY, Q1.REBATE

         FROM DB2USER.DAILY_SALES AS Q1) AS Q2

      GROUP BY Q2.PROD_KEY

      UNION ALL
```

```
      SELECT cast (NULL as integer) AS C0, SUM(DISTINCT
Q5.DISCOUNT) AS C1, COUNT(DISTINCT Q5.DISCOUNT) AS C2,
Q5.PROD_KEY

      FROM

         (SELECT Q4.PROD_KEY, Q4.DISCOUNT

         FROM DB2USER.DAILY_SALES AS Q4) AS Q5

      GROUP BY Q5.PROD_KEY) AS Q7

   GROUP BY Q7.PROD_KEY) AS Q8
```

If you cannot avoid multiple DISTINCT aggregations, consider using the **DB2_EXTENDED_OPTIMIZATION** registry variable with the **ENHANCED_MULTIPLE_DISTINCT** option. This option will result in the input stream to the multiple distinct aggregates being read once and then reused for each arm of the UNION. This option might improve the performance of these types of queries where the ratio of processors to the number of database partitions is low (for example, the ratio is less than or equal to 1). This setting should be used in DPF (Database Partitioning Feature) environments without symmetric multiprocessors (SMPs). This optimization extension might not improve query performance in all environments. Testing should be done to determine individual query performance improvements.

Avoid unnecessary outer joins

The semantics of certain queries require outer joins (either left, right or full). However, if the query semantics do not require an outer join and it is being used to deal with inconsistent data, it is best to deal with the inconsistent data problems at their root cause. For example, in a data mart with a star schema, the fact table might contain rows for transactions but no matching parent dimension rows for some dimensions, due to data consistency problems. This could occur because the extract, transform, and load (ETL) process could not reconcile some business keys for some reason. In this scenario, the fact table rows are left outer joined with the dimensions to ensure they get returned, even when they do not have a parent. For example:

```
SELECT ... FROM DAILY_SALES F

LEFT OUTER JOIN CUSTOMER C ON F.CUST_KEY = C.CUST_KEY

LEFT OUTER JOIN STORE S ON F.STORE_KEY = S.STORE_KEY

WHERE

C.CUST_NAME = 'SMITH'
```

The left outer join can prevent a number of optimizations, including the use of specialized star-schema join access methods. However, in some cases the left outer join can be automatically rewritten to an inner join by the query optimizer. In this example,

the left outer join between CUSTOMER and DAILY_SALES can be converted to an inner join because the predicate C.CUST_NAME = 'SMITH' will remove any rows with NULL values in this column, making a left outer join semantically unnecessary. So the loss of some optimizations due to the presence of outer joins might not adversely affect all queries. However, it is important to be aware of these limitations and avoid outer joins unless they are absolutely required.

Use OPTIMIZE FOR N ROWS clause with FETCH FIRST N ROWS ONLY clause

OPTIMIZE FOR N ROWS clause indicates to the optimizer that the application intends to only retrieve N rows, but the query will return the complete result set. FETCH FIRST N ROWS ONLY clause indicates that the query should only return N rows.

The DB2 data server does not automatically assume OPTIMIZE FOR N ROWS when FETCH FIRST N ROWS ONLY is specified for the outer subselect. Try specifying OPTIMIZE FOR N ROWS along with FETCH FIRST N ROWS ONLY, to encourage query access plans that return rows directly from the referenced tables, without first performing a buffering operation such as inserting into a temporary table, sorting or inserting into a hash join hash table.

Applications that specify OPTIMIZE FOR N ROWS to encourage query access plans that avoid buffering operations, yet retrieve the entire result set, might experience poor performance. This is because the query access plan that returns the first N rows fastest might not be the best query access plan if the entire result set is being retrieved.

If you are using the star schema join, ensure your queries fit the required criteria

The optimizer considers two specialized join methods for star schemas, called a star join or a hub join, which can help to significantly improve performance. However, the query must meet the following criteria.

- For each query block
  - At least 3 different tables must be joined
  - All join predicates must be equality predicates
  - No subqueries exist
  - No correlations or dependencies exist between tables or to outside the query block
  - In addition, for index ANDing, there must be no non-deterministic functions, because fact table predicates must be applied by indexes to facilitate semi-joins.

- A fact table
  - Is the largest table in the query block
  - Is at least 10000 rows
  - Is considered to be only one table
  - Must be joined to at least two dimension tables or to groups called snowflakes
- A dimension table
  - Is not the fact table
  - Can be joined individually to the fact table or in snowflakes
- A dimension table or a snowflake
  - Must filter the fact table (Filtering is based on the optimizer's estimates.)
  - Must have a join predicate to the fact table that uses a leading column in a fact table index. This criterion must be met in order for either star join or hub join to be considered, although hub join will only need to use a single fact table index.

A query block representing a left or right outer join can reference only two tables, so a star-schema join does not qualify.

Explicitly declaring referential integrity is not required for the optimizer to recognize a star-schema join.

Avoid redundant predicates

Avoid redundant predicates, especially when they occur across different tables. In some cases, the optimizer cannot detect that the predicates are redundant. This might result in cardinality underestimation.

For example, within SAP BI applications the snowflake schema with fact and dimension tables is used as query optimized data structure. In some cases, there is a redundant time characteristic column ("SID_0CALMONTH" for month or “"SID_0FISCPER" for year) defined on the fact and dimension table.

The SAP BI OLAP processor generates redundant predicates on the time characteristics column of the dimension and the fact table.

These redundant predicates might result in longer query runtime.

The following section provides an example with two redundant predicates that are defined in the WHERE condition of a SAP BI query. Identical predicates are defined on the time dimension (DT) and fact (F) table:

```
    AND  (      "DT"."SID_0CALMONTH" = 199605

           AND "F". "SID_0CALMONTH" = 199605

            OR "DT"."SID_0CALMONTH" = 199705

           AND "F". "SID_0CALMONTH" = 199705 )

 AND NOT (      "DT"."SID_0CALMONTH" = 199803

           AND "F". "SID_0CALMONTH" = 199803 )
```

The DB2 optimizer does not recognize the predicates as identical, and treats them as independent. This leads to underestimation of cardinalities, suboptimal query access plans and longer query run times.

For that reason the redundant predicates are removed by the DB2 database platform-specific software layer.

The above predicates are transferred to the ones shown below. Only the predicates on the fact table column "SID_0CALMONTH" remain:

```
    AND  (      "F". "SID_0CALMONTH" = 199605

            OR "F". "SID_0CALMONTH" = 199705 )

AND NOT (       "F". "SID_0CALMONTH" = 199803 )
```

Apply the instructions in SAP notes 957070 and 1144883 to remove the redundant predicates.

Designing and configuring your database

There are a number of database design and configuration options that can affect query performance. For additional database design suggestions, see the "Physical Database Design" best practice paper.

Use Constraints to Improve Query Optimization

Consider defining unique, check and referential integrity constraints. These constraints provide semantic information that allows the DB2 optimizer to rewrite queries to eliminate joins, push aggregation down through joins, push FETCH FIRST N ROWS down through joins, remove unnecessary DISTINCT operations and a number of other optimizations. Informational constraints can also be used for check and referential integrity constraints, when the application can guarantee the relationships itself. The same optimizations are possible. Constraints that are enforced by the database manager when rows are updated (or inserted or deleted) can lead to high system overhead, especially when updating a large number of rows that have referential integrity constraints. If an application has already verified information before updating a row, it might be more efficient to use informational constraints, rather than normal constraints.

For example, consider 2 tables DAILY_SALES and CUSTOMER. Each row in the CUSTOMER table has a unique customer key (CUST_KEY). DAILY_SALES contains a CUST_KEY column and each row references a customer key in the CUSTOMER table. A referential integrity constraint could be created to represent this 1:N relationship between CUSTOMER and DAILY_SALES. If the application enforced the relationship, the constraint could be defined as informational. The query below could then avoid performing the join between CUSTOMER and DAILY_SALES because no columns are retrieved from CUSTOMER and every row from DAILY_SALES will find a match in CUSTOMER. The query optimizer will automatically remove the join.

```
SELECT AMT_SOLD, SALE PRICE, PROD_DESC

FROM DAILY_SALES, PRODUCT, CUSTOMER

WHERE

DAILY_SALES.PROD_KEY = PRODUCT.PRODKEY AND

DAILY_SALES.CUST_KEY = CUSTOMER.CUST_KEY
```

The application must enforce informational constraints, otherwise queries might return incorrect results. In the above example, if rows existed in DAILY_SALES that did not have a corresponding customer key in the CUSTOMER table, the above query would incorrectly return these rows.

Use the REOPT bind option with input variables in complex queries

Input variables are essential for good statement preparation time in an online transaction processing (OLTP) environment, where statements tend to be simpler and query access plan selection is more straightforward. Multiple executions of the same query with different input variable values can reuse the compiled access section in the dynamic statement cache, avoiding expensive SQL statement compilations whenever the input values change.

However, input variables can cause problems for complex query workloads where query access plan selection is more complex and the optimizer needs more information to make good decisions. Moreover, statement compilation time is usually a small component of total execution time and BI queries do not tend to be repeated so they do not benefit from the dynamic statement cache.

If input variables need to be used in a complex query workload, consider using the REOPT(ALWAYS) BIND option. The REOPT BIND option defers statement compilation from PREPARE to OPEN or EXECUTE time, when the input variable values are known. The values are passed to the SQL Compiler so the optimizer can use the values to compute a more accurate selectivity estimate. REOPT(ALWAYS) indicates that the statement should be recompiled for every execution. REOPT(ALWAYS) can also be used for complex queries that reference special registers, such as "WHERE TRANS_DATE = CURRENT DATE - 30 DAYS". If input variables lead to poor access plan selection for OLTP workloads and REOPT(ALWAYS) results in excessive overhead due to statement compilation, consider using REOPT(ONCE) for selected queries. REOPT(ONCE) defers statement compilation until the first input variable value is bound in. The SQL statement is compiled and optimized using this first input variable value. Subsequent executions of the statement with different values will reuse the access section compiled based on the first input value. This can be a good approach if the first input variable value is representative of subsequent values and it provides a better query access plan than one that is optimized using the default estimates used by the optimizer when the input variable values are unknown.

There a number of ways that REOPT can be specified:

- For embedded SQL in C/C++ applications, use the REOPT BIND option. This BIND option affects re-optimization behavior for both static and dynamic SQL.

- For CLP packages, rebind the CLP package with the REOPT bind option. For example, to rebind the CLP package used for isolation level CS with REOPT ALWAYS, specify the following command:

```
rebind nullid.SQLC2G13 reopt always;
```

- For CLI applications or JDBC applications using the legacy JDBC driver, use the REOPT keyword setting in the db2cli.ini configuration file. The values and their options are:

    - 2 - NONE
    - 3 - ONCE
    - 4 - ALWAYS

- For JDBC applications using the JCC Universal Driver, use one of the following approaches:
    - Use the SQL_ATTR_REOPT connection or statement attribute.
    - Use the SQL_ATTR_CURRENT_PACKAGE_SET connection or statement attribute to specify either the NULLID, NULLIDR1 or NULLIDRA package sets. The NULLIDR1 and NULLIDRA are reserved package set names. When used, REOPT ONCE and REOPT ALWAYS are implied respectively. These package sets have to be explicitly created with the following commands:

```
db2 bind db2clipk.bnd collection NULLIDR1

db2 bind db2clipk.bnd collection NULLIDRA
```

- For SQL PL procedures, use one of the following approaches:
    - Use the SET_ROUTINE_OPTS stored procedure to set the bind options to be used for the created of SQL PL procedures within the current session. For example, call

```
sysproc.set_routine_opts('reopt always');
```

    - Use the **DB2_SQLROUTINE_PREPOPTS** registry variable to set the SQL PL procedure options at the instance level. Values set using the SET_ROUTINE_OPTS stored procedure will override those specified with **DB2_SQLROUTINE_PREPOPTS**.

You can also use optimization profiles to set REOPT for static and dynamic statements, as shown in the following example:

```
<STMTPROFILE ID="REOPT example ">

    <STMTKEY>

      <![CDATA[select acct_no from customer where name = ? ]]>

    </STMTKEY>

    <OPTGUIDELINES>

      <REOPT VALUE='ALWAYS'/>
```

```
  </OPTGUIDELINES>

</STMTPROFILE>
```

Choose the best optimization class for your workload

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 statements are extremely simple.

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 classes 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 with 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.

  Optimization 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 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 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 queries.

For SAP applications, always use optimization class 5. This optimization class enables many DB2 features optimized for SAP, such as setting the **DB2_REDUCED_OPTIMIZATION** registry variable.

Use parameter markers to reduce compilation time for dynamic queries

The DB2 data server can avoid recompiling a dynamic SQL statement that has been previously run by storing the access section and statement text in the dynamic statement cache. A subsequent PREPARE request for this statement will attempt to find the access section in the dynamic statement cache, avoiding compilation. However, statements that only differ because of literals used in predicates will not match. For example, the following 2 statements are considered different by the dynamic statement cache:

```
SELECT AGE FROM EMPLOYEE WHERE EMP_ID = 26790

SELECT AGE FROM EMPLOYEE WHERE EMP_ID = 77543
```

Even relatively simple SQL statements can result in excessive system CPU usage due to statement compilation, if they are run very frequently. The “Tuning and Monitoring Database System Performance” best practices paper describes how to detect this type of performance problem. If your system experiences this type of performance problem, consider changing the application to use parameter markers to pass the predicate value to the DB2 compiler, rather than explicitly including it in the SQL statement. However, the access plan might not be optimal for complex queries that use parameter markers in predicates. For more information, see “Use the REOPT bind option with input variables in complex queries” on page 17.

Set DB2_REDUCED_OPTIMIZATION registry variable

If setting the optimization class does not reduce the compilation time sufficiently for your application, try setting the **DB2_REDUCED_OPTIMIZATION** registry variable. This registry variable provides more control over the optimizer's search space than setting the optimization class. This registry variable lets you request either reduced optimization features or rigid use of optimization features at the specified optimization class. If you reduce the number of optimization techniques used, you also reduce time and resource use during optimization.

Although optimization time and resource use might be reduced, there is increased risk of producing a less than optimal query access plan.

First, try setting the registry variable to YES. If the optimization class is 5 (the default) or lower, the optimizer disables some optimization techniques that might consume significant prepare time and resources but do not usually produce a better query access plan. If the optimization class is exactly 5, the optimizer reduces or disables some additional techniques, which might further reduce optimization time and resource use, but also further increase the risk of a less than optimal query access plan. For optimization classes lower than 5, some of these techniques might not be in effect in any case. If they are, however, they remain in effect.

If the YES setting does not provide a sufficient reduction in compilation time, try setting the registry variable to an integer. The effect is the same as YES, with the following

additional behavior for dynamically prepared queries optimized at class 5. If the total number of joins in any query block exceeds the setting, then the optimizer switches to greedy join enumeration instead of disabling additional optimization techniques. The resulting effect is that the query will be optimized at a level similar to optimization class 2.

Gather accurate catalog statistics, including advanced statistics features

Accurate database statistics are critical for query optimization. Perform RUNSTATS regularly on any tables critical to query performance. You might also want to collect statistics on system catalog tables, if an application queries these tables directly and if there is significant catalog update activity such as DDL statements. Automatic statistics collection can be enabled to allow the DB2 data server to automatically perform RUNSTATS. Real time statistics collection can be enabled to allow the DB2 data server to provide even more timely statistics by collecting them immediately before queries are optimized.

If collecting statistics manually using RUNSTATS, you should use the following options at a minimum:

```
RUNSTATS ON TABLE DB2USER.DAILY_SALES WITH DISTRIBUTION AND
SAMPLED DETAILED INDEXES ALL
```

Distribution statistics make the optimizer aware of data skew. Detailed index statistics provide more details about the I/O required to fetch data pages when the table is accessed using a particular index. Collecting detailed index statistics consumes considerable CPU and memory for large tables. The SAMPLED option provides detailed index statistics with nearly the same accuracy but requires a fraction of the CPU and memory. These defaults are also used by automatic statistics collection when a statistical profile has not been provided for a table.

To improve query performance, consider collecting more advanced statistics such as column group statistics, LIKE statistics or creating statistical views.

Column group statistics

If your query has more than one join predicate joining two tables, the DB2 optimizer calculates how selective each of the predicates is, before choosing a plan for executing the query.

For example, consider a manufacturer who makes products from raw material of various colors, elasticities, and qualities. The finished product has the same color and elasticity as the raw material from which it is made. The manufacturer issues the query:

```
SELECT PRODUCT.NAME, RAWMATERIAL.QUALITY

FROM PRODUCT, RAWMATERIAL

     WHERE PRODUCT.COLOR        =  RAWMATERIAL.COLOR

     AND PRODUCT.ELASTICITY     =  RAWMATERIAL.ELASTICITY
```

This query returns the names and raw material quality of all products. There are two join predicates:

```
   PRODUCT.COLOR        =  RAWMATERIAL.COLOR

   PRODUCT.ELASTICITY   =  RAWMATERIAL.ELASTICITY
```

The optimizer assumes that the two predicates are independent, which means that all variations of elasticity occur for each color. It then estimates the overall selectivity of the pair of predicates by using catalog statistic information for each table based on the number of levels of elasticity and the number of different colors. Based on this estimate, it might choose, for example, a nested loop join in preference to a merge join, or vice versa.

However, these two predicates might not be independent. For example, highly elastic materials might be available in only a few colors, and the very inelastic materials might be available in a few other colors that are different from the elastic ones. Then the combined selectivity of the predicates eliminates fewer rows so the query will return more rows. Without this information, the optimizer might no longer choose the best plan.

To collect the column group statistics on PRODUCT.COLOR and PRODUCT.ELASTICITY, issue the following RUNSTATS command:

```
RUNSTATS ON TABLE product ON COLUMNS ((color, elasticity))
```

The optimizer uses these statistics to detect cases of correlation, and dynamically adjust combined selectivities of the correlated predicates, thus obtaining a more accurate estimate of the join size and cost.

When a query groups data by using keywords such as GROUP BY or DISTINCT, column group statistics also enable the optimizer to compute the number of distinct groupings.

Consider the following query:

```
SELECT DEPTNO, YEARS, AVG(SALARY)

  FROM EMPLOYEE

 GROUP BY DEPTNO, MGR, YEAR_HIRED
```

Without any index or column group statistics, the 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.

Column group statistics collected on DEPTNO, MGR, and YEAR_HIRED will provide the optimizer with the exact number of distinct groupings for the query above:

```
RUNSTATS ON TABLE EMPLOYEE ON COLUMNS ((DEPTNO, MGR, YEAR_HIRED))
```

In addition to JOIN predicate correlation, the optimizer also manages correlation with simple equal predicates, such as

```
DEPTNO = ”Sales” AND  MGR = “John”
```

In the EMPLOYEE table above, predicates on DEPTNO are likely to be independent from predicates on YEAR. However, the predicates on DEPTNO and MGR are certainly not independent, since each single department would usually be managed by one manager at a time. The optimizer uses the statistical information about the columns to determine the combined number of distinct values and adjusts the cardinality estimate to account for correlation between the two columns.

Sub-element statistics

If you specify LIKE predicates using the % wildcard character in any position other than at the end of the pattern, you should collect basic information about the sub-element structure.

As well as the wildcard LIKE predicate (for example, SELECT .... FROM DOCUMENTS WHERE KEYWORDS LIKE '%simulation%'), the columns and the query must fit certain criteria to benefit from sub-element statistics.

Table columns should contain sub-fields or sub-elements separated by blanks. For example, a four-row table DOCUMENTS contains a KEYWORDS column with lists of relevant keywords for text retrieval purposes. The values in KEYWORDS are:

```
'database simulation analytical business intelligence'

'simulation model fruit fly reproduction temperature'

'forestry spruce soil erosion rainfall'

'forest temperature soil precipitation fire'
```

In this example, each column value consists of 5 sub-elements, each of which is a word (the keyword), separated from the others by one blank.

The query should reference these columns in WHERE clauses.

The optimizer always estimates how many rows match each predicate. For these wildcard LIKE predicates, the optimizer assumes that the COLUMN being matched contains a series of elements concatenated together, and it estimates the length of each element based on the length of the string, excluding leading and trailing % characters. If you collect sub-element statistics, the optimizer will have information about the length or each sub-element and the delimiter. It can use this additional information to more accurately estimate how many rows will match the predicate.

To collect sub-element statistics, run RUNSTATS with the LIKE STATISTICS clause.

Statistical views

The DB2 cost-based optimizer uses an estimate of the number of rows – or cardinality – processed by an access plan operator to accurately cost that operator. This cardinality estimate is the single most important input to the optimizer's cost model, and its accuracy largely depends upon the statistics that the RUNSTATS utility collects from the database. The statistics described above are all important for computing an accurate cardinality estimate, however there are some situations where more sophisticated statistics are required. In particular, more sophisticated statistics are required to represent more complex relationships, such as comparisons involving expressions (for example, price > MSRP + Dealer_markup), relationships spanning multiple tables (for example, product.name = 'Alloy wheels' and product.key = sales.product_key), or anything other than predicates involving independent attributes and simple comparison operations. Statistical views are able to represent these types of complex relationships because statistics are collected on the result set returned by the view, rather than the base tables referenced by the view.

When a query is compiled, the optimizer matches the query to the available statistical views. When the optimizer computes cardinality estimates for intermediate result sets, it uses the statistics from the view to compute a better estimate.

Queries do not need to reference the statistical view directly in order for the optimizer to use the statistical view. The optimizer uses the same matching mechanism used for materialized query tables (MQTs) to match queries to statistical views. In this respect, statistical views are very similar to MQTs, except they are not stored permanently, so they do not consume disk space and do not have to be maintained.

A statistical view is created by first creating a view and then enabling it for optimization using the ALTER VIEW statement. RUNSTATS is then run on the statistical view, populating the system catalog tables with statistics for the view. For example, in order to create a statistical view to represent the join between the time dimension table and the fact table in a star schema, do the following:

```
CREATE VIEW SV_TIME_FACT AS (

            SELECT T.* FROM TIME T, SALES S
```

```
            WHERE T.TIME_KEY = S.TIME_KEY)


ALTER VIEW SV_TIME_FACT ENABLE QUERY OPTIMIZATION


RUNSTATS ON TABLE DB2DBA.SV_TIME_FACT WITH DISTRIBUTION
```

This statistical view can be used to improve the cardinality estimate -and consequently the access plan and query performance - for queries such as:

```
SELECT SUM(S.PRICE)

  FROM SALES S, TIME T,  PRODUCT P

  WHERE

  T.TIME_KEY = S.TIME_KEY AND T.YEAR_MON = 200712 AND

  P.PROD_KEY = S.PROD_KEY AND P.PROD_DESC = 'Power drill'
```

Without the statistical view, the optimizer assumes that all fact table TIME_KEY values corresponding to a particular time dimension YEAR_MON value occur uniformly within the fact table. However, sales might have been particularly strong in December, resulting in many more sales transactions than other months.

There are many situations where statistical views can improve query performance and there are some straightforward best practices available to help determine which statistical views to create. Refer to the links in the "Further reading" section on page 35.

Automatic statistics collection is not currently available for statistical views. Collect statistical view statistics whenever base tables referenced by the statistical view have been updated significantly.

Minimize RUNSTATS impact

To further improve RUNSTATS time:

- Limit the columns for which statistics should be collected by using the COLUMNS clause. Often, a number of columns are never referenced by predicates in the query workload so they do not require statistics.

- Limit the columns for which distribution statistics are collected if the data tends to be uniform. Gathering distribution statistics requires more CPU and memory than basic column statistics. However, determining whether or not a column's values are uniform requires either having existing statistics or querying the data. This approach also assumes that the data will remain uniform, as the table is modified.

- Limit the number of pages and rows processed by using page or row level sampling by specifying the TABLESAMPLE SYSTEM or BERNOULLI clause. Start with a 10% page level sample, by specifying TABLESAMPLE SYSTEM(10). Check the accuracy of the statistics and whether system performance has degraded due to changes in access plan. If they have, try a 10% row level sample instead, by specifying TABLESAMPLE BERNOULLI(10). If the statistics accuracy is insufficient, increase the sampling amount. When using RUNSTATS page or row level sampling, use the same sampling rate for tables that are joined. This is important to ensure the join column statistics have the same level of accuracy.

- Collect index statistics during index creation by specifying the COLLECT STATISTICS option on the CREATE INDEX statement. This approach is faster than performing a separate RUNSTATS after the index has been created. It also ensures the new index has statistics immediately after creation, to allow the optimizer to accurately estimate the cost of using the index.

- Collect statistics when performing LOAD with the REPLACE option. This approach is faster than performing a separate RUNSTATS after the LOAD has completed. It also ensures the table has the most current statistics immediately after the data has been loaded, to allow the optimizer to accurately estimate the cost of using the table.

For a data server configured to use the Database Partitioning Feature (DPF), RUNSTATS collects statistics from a single database partition. If RUNSTATS is issued on a database partition on which the table resides, statistics will be collected there. If not, statistics will be collected on the first database partition in the database partition group for the table. Ensure statistics for tables that are joined are collected from the same database partition in order to ensure consistent statistics.

Avoid updating catalog statistics manually

The DB2 data server supports manually updating catalog statistics by issuing UPDATE statements against views in the SYSSTAT schema. This feature can be useful when mimicking a production database on a test system in order to examine query access plans. The db2look tool is very helpful for capturing the DDL and UPDATE SYSSTAT statements for playback on another system.

However, avoid manually updating statistics as a means to influence the query optimizer by providing incorrect statistics in order to force a particular query access plan. While this practice might result in improved performance for some queries, it might result in degradations for others. Consider other tuning approaches before resorting to this approach. When this approach does become necessary, be sure to record the original statistics in case they need to be restored if the updated statistics lead to performance degradations.

Use optimization profiles if other tuning options do not produce acceptable results

If you have followed the best practices recommended in this paper, but you believe that you are still getting less than optimal performance, you can provide explicit optimization guidelines to the DB2 optimizer.

The optimization guidelines are contained in an XML document called an optimization profile. The profile defines SQL statements and their associated optimization guidelines.

If you use optimization profiles extensively, they require a lot of effort to maintain. More importantly, you can only use the optimization profiles to improve performance for existing SQL statements. Following best practices consistently can help improve query performance stability for all queries, including future ones.

Tuning SQL statements using the explain facility

The explain facility is used to display the query access plan chosen by the query optimizer to run an SQL statement. It contains extensive details about the relational operations used to run the SQL statement such as the plan operators, their arguments, order of execution, and costs. Since the query access plan is one of the most critical factors in query performance, it is important to be able to understand the explain facility output in order to diagnose query performance problems.

Explain information is typically used to:

- understand why application performance has changed
- evaluate performance tuning efforts

Analyzing performance changes

To help you understand the reasons for changes in query performance, you need the before and after explain information which you can obtain by performing the following steps:

1. Capture explain information for the query before you make any changes and save the resulting explain tables. Alternatively, you might save the output from the db2exfmt explain tool. However, having the explain information in the explain tables allows easy querying with SQL, in order to perform more sophisticated analysis. As well, it provides all the obvious maintenance benefits of having data in a relational DBMS. Moreover, the db2exfmt tool can be run any time.

2. 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. Alternatively, if using DB2 9.5, collect the explain snapshot at the same time as the explain table population. The explain snapshot contains all the relevant statistics at the time the statement is explained. The db2exfmt tool will automatically format the statistics contained in the snapshot. This is especially important when using automatic or real time statistics collection, because the statistics used for query optimization might not yet be in the system catalog tables or they might have changed between the time the statement was explained and when they were retrieved from the system catalogs.

3. Save or print the data definition language (DDL) statements, including those for CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE TABLESPACE. The db2look tool will also perform this task.

The information that you collect in this way, provides a reference point for future analysis. For dynamic SQL statements, you can collect this information when you run your application for the first time. For static SQL statements, you can also collect this

information at bind time. It is especially important to collect this information before a major system change such as the installation of a new service level or DB2 release or a significant configuration change, such as adding or dropping database partitions and redistributing data. This is because these types of system changes might result in an adverse change in access plan. Although access plan regressions should be a rare occurrence, having this information available will allow you to resolve performance regressions faster. 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 plan. Using the catalog statistics information displayed by Visual Explain or db2exfmt, you might notice that the number of index levels (NLEVELS column) is now substantially higher than when the query was first bound to the database. You might then choose to perform one of 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.

Evaluating performance tuning efforts

You can take a number of actions to help improve query performance, such as adjusting configuration parameters, adding containers, and collecting fresh catalog statistics.

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.

Best Practices

- Avoid complex expressions in search conditions
- Avoid join predicates on expressions
- Avoid applying expressions over columns in local predicates
- Avoid data type mismatches on join columns
- Avoid non-equality join predicates
- Avoid multiple aggregations with the DISTINCT keyword
- Avoid unnecessary outer joins
- Use OPTIMIZE FOR N ROWS clause with FETCH FIRST N ROWS ONLY clause
- If you are using the star schema join, ensure your queries fit the required criteria
- Avoid redundant query restrictions
- Use constraints to improve query optimization
- Use the REOPT bind option with input variables in complex queries
- Choose the best optimization class for your workload
- Set DB2_REDUCED_OPTIMIZATION registry variable
- Gather accurate catalog statistics, including advanced statistics features
- Minimize RUNSTATS impact

- Avoid updating catalog statistics manually
- Use optimization profiles if other tuning options do not produce acceptable results

Conclusion

The best practices in this white paper are intended to help you minimize the use of system resources and the time required to run SQL statements. To avoid potential performance issues, use these best practices when you write your SQL statements and configure your settings.

To write SQL statements that can be easily optimized by the DB2 optimizer, avoid the following:

- complex expressions in search conditions
- applying expressions over columns in local predicates
- data type mismatches on join columns
- non-equality join predicates
- multiple aggregations with the DISTINCT keyword
- unnecessary outer joins
- redundant query restrictions

If you are using the star schema join, ensure your queries fit the required criteria, in order for the DB2 data server to be able to use specialized star join methods to improve query performance.

If you specify the FETCH FIRST n ROWS ONLY clause, you should also specify the OPTIMIZE FOR n ROWS clause to avoid buffering operations.

When you are designing and configuring your DB2 database, consider the following best practices:

- use constraints
- use the REOPT bind option with SQL statements that have input variables used in predicates
- choose the best optimization class for your workload
- set **DB2_REDUCED_OPTIMIZATION** registry variable
- ensure that the catalog statistics are accurate
- consider collecting advanced statistics, such as column group statistics, sub-element statistics, and statistical views

Finally, use the DB2 explain functionality to review potential query access plans and determine how to tune the queries for best performance.

Further reading

- DB2 Best Practices website
  http://www.ibm.com/developerworks/db2/bestpractices/

- IBM DB2 Database for Linux, UNIX, and Windows Version 9.5 Information Center http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/

- "Designing informational constraints"
  http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/topic/com.ibm.db2.luw.admin.dbobj.doc/doc/c0020160.html

- "Automatic statistics collection"
  http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0011762.html

- "Statistical views"
  http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0021713.html

- "Explain facility"
  https://publib.boulder.ibm.com/infocenter/db2luw/v9r5/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0005134.html

- "Parameter markers"
  http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/topic/com.ibm.db2.luw.apdv.routines.doc/doc/c0020295.html

- "Recreate optimizer access plans using db2look"
  http://www.ibm.com/developerworks/db2/library/techarticle/dm-0508kapoor/

- "Vital Statistics II: Using Statistical Views to Improve Query Performance"
  http://www.idug.org/wps/portal/idug/kcxml/04_Sj9SPykssy0xPLMnMz0vM0Y_QjzKLN4o38rAESYGYnoH6kehCAQghX4_83FT9IKBUpDlQyMzDRz8qJzU9MblSP1jfWz9AvyA3NKLc29ERAP8BqUk!/delta/base64xml/L0lJSk03dWlDU1EhIS9JRGpBQU15QUJFUkVSRUlnLzRGR2dkWW5LSjBGUm9YZmcvN18yXzZITA!!?PC_7_2_6HL_WCM_CONTEXT=/wps/wcm/connect/IDUG+Site/Solutions+Journal/Solutions+Journal+Online+Magazine/Volume+13%2C+Number+2/ISJ+V13+N2-Intelligent-Optimizer

- "Influence query optimization with optimization profiles and statistical views in DB2 9" http://www.ibm.com/developerworks/db2/library/techarticle/dm-0612chen/

- SAP Service Marketplace http://sap.service.com/notes

  You need additional SAP authorization to access SAP notes at the SAP Service Marketplace.

Contributors

Brigitte Blaeser
SAP NetWeaver BI on DB2 for Linux, UNIX, and Windows

Ian Maione
DB2 Advanced Problem Diagnostics

Kaarel Truuvert
DB2 Query Optimizer Development

Karl Fleckenstein
SAP/DB2 Solutions

Qi Cheng
DB2 Query Rewriter Development

Sonia Ang
Information Management Technical Sales

Tim Vincent
Chief Architect DB2 LUW

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 grant 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.

The following paragraph does not apply to the United Kingdom or any other country 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.

Without limiting the above disclaimers, IBM provides no representations or warranties regarding the accuracy, reliability or serviceability of any information or recommendations provided in this publication, or with respect to any results that may be obtained by the use of the information or observance of any recommendations provided herein. The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any recommendations or techniques herein is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Anyone attempting to adapt these techniques to their own environment do so at their own risk.

This document and the information contained herein may be used solely in connection with the IBM products discussed in this document.

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.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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 contains 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: © Copyright IBM Corporation 2008. All Rights Reserved.

This information contains 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml

Windows is a trademark of Microsoft Corporation 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.

Other company, product, or service names may be trademarks or service marks of others.