IBM® Smart Analytics System®

Best practices

Query optimization in a data warehouse

Gregor Meyer

Detlev Kuntze

Toni Bollinger

Sascha Laudien

IBM Data Warehousing Center of Excellence

Published: June 2011

Table of contents

Executive summary

In large data warehouse systems, it is critical to optimize query workloads to maximize system utilization and minimize processing times. This paper describes techniques for the optimization of data warehouse system query workloads. The scenarios focus on IBM Smart Analytics Systems where DB2® software manages multiple database partitions in a cluster. The scenarios describe optimization methods that can help improve performance in a short time. The guidelines in this document might not apply to transactional applications.

The two main tasks for query optimization are as follows:

1. Provide the DB2 optimizer with rich statistical information. The DB2 optimizer uses the statistical information to perform its analysis and find a good execution plan. This paper describes best practices for gathering statistics.
2. Determine the best query execution plan. If a query needs further tuning, you must inspect the explain plan for the query. This paper provides a list of patterns to look for in explain plans. Each pattern is paired with recommended actions.

It is assumed that the reader of this document has a basic understanding of the design and administration of data warehouse databases with DB2 products.

Introduction

The workloads on data warehouse systems can range from complex analytical queries processing terabytes of data to many short tactical queries requiring sub second response time. Updates to large volumes of data might have to be processed in parallel. Queries can be ad hoc, that is, the access pattern is not known in advance and might vary significantly from one query to the next. This paper examines some characteristics of data warehouse workloads in more detail and describes how the IBM Smart Analytics System design is optimized for these workloads.

The DB2 query compiler and its optimizer are a key component for executing each SQL statement efficiently. The best practices in this document help you get optimal execution plans.

This paper first outlines a summary of guidelines for defining the physical data model for the database and partitioning the data. However, the main focus of this document is on the collection of statistics and the analysis of query plans.

The DB2 compiler uses a cost-based optimizer. Cost estimates rely on statistics about the data. Therefore it is important to gather rich statistics. This paper provides guidelines for collecting statistics with the **RUNSTATS** command. These guidelines are complemented by recommendations for defining foreign keys and functional dependency constraints that help the optimizer better understand the data model.

If a query or data modification statement runs slower than expected, the execution plan that the compiler creates can help identify the operations that need further optimization. This paper describes how to extract and analyze execution plans that contain important information. This paper also provides a series of critical patterns that you might see in an execution plan. Each pattern has an associated action that helps in improving response times.

Optimization considerations for data warehouse systems

Use the computing power of database partitions

Transactional workloads typically comprise many short reads and updates, mostly in sub second response time. Although the database might be large, a single transaction usually processes only a small amount of data. Data warehousing query workloads consist of queries that often process and aggregate large amounts of data. Queries tend to be more complex, for example, they can contain more joins.

These differences lead to the following recommendations:

- Distribute data across database partitions to use parallelism for tasks that need a significant amount of resources.
- Collocate data that is joined together, that is, store it in the same database partition. Transmitting data between partitions can be expensive in terms of network and other resources.

Determine how to partition the database

In this section, partitioning refers to database partitioning, that is, distributing data across the database partitions using hashing. Practical advice is to keep small tables on a single database partition, for example, on the administration node of the IBM Smart Analytics System. The bigger tables should be partitioned across the data nodes.

Recommendation: Use the following guidelines as a starting point:

- Partition tables with more than 10,000,000 rows.
- Keep tables with 10, 000 - 10,000,000 rows on the administration node, without partitioning them. If these tables are frequently used, replicate them across the partitions on the data nodes.
- Keep tables with up to 10,000 rows on the administration node, without partitioning them. There is no need to replicate these tables because they can be sent quickly across the network at query run time.

Use collocation as a major driver for choosing distribution keys

What is a good distribution key for partitioning? A distribution key should have the following properties:

- It should distribute data and workload evenly across partitions.
- It should collocate rows that are joined together.

The first property addresses the fact that the run time of a query depends on the slowest partition. This partition is usually the one that has to process more data than others. Even distribution of data ensures that no partition contains much more data than others.

The second property is important because DB2 partitioning uses a shared-nothing architecture to scale out. If possible, collocate joined data on the same partition. Otherwise, a large amount of data might have to be sent across the network, and that might slow down the query and limit scalability. This property is true for virtually any database system that uses a shared-nothing architecture for scalability. Collocation is less critical for smaller databases where all partitions can be kept on the same server and where database communication does not need a network.

In larger data warehouse systems, collocating data from different tables can be more important for performance than finding the optimal even distribution of data within tables. To balance the requirements for even distribution and collocation, follow this procedure for defining distribution keys:

1. Determine the biggest tables.
2. Check the data model or queries and locate equijoins between the large tables. In a simple star schema, an equijoin might be the join between the fact table and a dimension table.
3. Check whether some of the joined columns can be used for distribution, that is, if they distribute data well. Keep the number of columns as small as possible. Using a single column is best in most cases. A primary key must include the distribution key, but there is no need to use all primary key columns for distribution.
4. Optional: Distribute smaller tables if doing so enables collocation. For example, distribute a STORE table by using its primary key if you distribute fact tables by using the STORE key.
5. Optional: Add redundant columns to fact tables. For example, add a STORE column to a table that contains itemized sales transactions even if the market basket identifier alone might be a primary key. Adding redundant columns would enable a collocated join to other tables that are distributed by using the STORE column.
6. For remaining tables, choose a distribution key that minimizes the skew in the distribution of data. Primary keys or columns with many different values are good candidates. A column should not have many rows containing NULL or some other default value.

It is usually not possible to achieve collocation in all cases. In many cases, an inability to collocate data is not a problem. Focus on the biggest tables and the joins that are used most often or include many rows. For these tables, it is acceptable if collocation increases the distribution skew by 5 - 10%.

A collocated join requires tables to be in the same partitioning group, even if the tables are in different table spaces.

In the special case of a multi-dimension model such as a star schema with one fact table and multiple dimensions, the procedure for defining distribution keys is simple:

1. Partition the fact table and the biggest dimension by specifying the dimension key as the distribution key. Choose another large dimension if the distribution is skewed.
2. Replicate the other dimension tables.

The previous procedures require tests to determine whether data is distributed evenly. There are two simple methods for checking this:

- Count the number of rows per partition, as shown in the following example:

```
-- Actual number of row counts per partition
SELECT DBPARTITIONNUM(KEY),COUNT_BIG(*)
FROM THETABLE TABLESAMPLE SYSTEM(10)
GROUP BY ROLLUP(DBPARTITIONNUM(KEY)) ORDER BY 2;
```

- Check the space allocation by using the SYSIBMADM.ADMINTABINFO view, as shown in the following example:

```
 -- Space allocation per partition
SELECT  DBPARTITIONNUM,  SUM(DATA_OBJECT_L_SIZE) SIZE_KB
FROM  SYSIBMADM.ADMINTABINFO
WHERE (TABSCHEMA,TABNAME) = ('THESCHEMA','THETABLE')
GROUP BY ROLLUP(DBPARTITIONNUM) ORDER BY 2;
```

You can use the information that you collect to determine data distribution, as follows:

- The size of partitions, which is based on space allocation, is relevant because DB2 software fetches full pages or extents rather than single rows. The allocated space is an indicator of the I/O load on the partition, and the row count is an indicator of the CPU load.
- Compare the size of the biggest partition with the average size of all partitions. A table is balanced if the size of each partition is close to the average size. The largest partition, however, limits the performance. For example, if a partition is two times larger than the average, a table scan might take twice the time of a table scan on the same data in a perfectly balanced table. A partition with much less data than average does not cause performance problems.
- Partitions using 10 - 20% more space than average and partitions using 10 – 20% more rows that average are normal. The table samples should not be too small because the differences in row counts might become less reliable. For example, with a 5% sample, some partitions might report 10% more rows than other partitions even if the table is balanced.

You can test the distribution of data with a new partitioning key by using a procedure such as the following one:

```
-- Create a table with the new distribution key
CREATE TABLE TEMP.THETABLE_TESTKEY AS
      ( SELECT NEWDISTKEY FROM THETABLE  TABLESAMPLE SYSTEM(10) )
      DATA INITIALLY DEFERRED REFRESH DEFERRED
      DISTRIBUTE BY HASH ( NEWDISTKEY ) IN THETABLESPACE;
COMMIT;

UPDATE COMMAND OPTIONS USING C OFF;
ALTER TABLE TABLE TEMP.THETABLE_TESTKEY ACTIVATE NOT LOGGED
INITIALLY;
REFRESH TABLE TEMP.THETABLE_TESTKEY;

-- Then check row counts per partition.
```

When the number of rows per partition is about the same, the amount of data that queries fetch per partition might still vary because the selectivity of filter conditions might vary by partition. You can check the number of rows that are processed per partition by creating query plans with *section actuals*. For details, see the "Explain plans with actual cardinalities" section.

Make the database queries faster by avoiding indexes

SQL statements in short queries often use filter conditions that are selective, that is, they return few rows. The cost of table scans can be prohibitive. Regular row-based indexes address this workload well.

In data warehouse environments, however, queries tend to process larger portions of tables. Regular indexes are useful in data warehouse databases also, but there are important caveats:

- Index maintenance can take a considerable amount of time in large updates.
- Indexes encourage the query compiler to use nested loop joins with index lookups. Without the index, the optimizer might choose a hash join that uses a table scan, which can be faster when many lookups are needed per query.

If queries filter on a certain column, run the queries without any changes to see whether the queries are fast enough. If they are not fast enough, follow these steps:

1. Consider using table partitioning. If table partitioning is applicable, create no more than 200 hundred partitions per table.

2. Consider using multidimensional clustering (MDC). Do not make the dimensions too fine grained, because this can waste space in MDC blocks that are not full. If more than 20% of space is unused, make the dimensions more coarse grained by using generated columns.

3. If the previous two techniques are not appropriate or do not sufficiently improve queries performance, create a regular index.

Database configurations

IBM Smart Analytics System software comes with many predefined configurations. This section provides information about some additional configurations that are generally useful when you are optimizing you system.

Recommendation: For simplified testing **make MQTs eligible without checking the time of their last REFRESH.**

```
DB2 UPDATE DB CFG FOR THEDB USING DFT_REFRESH_AGE ANY
```

Subsequent sections describe how to use the explain tool and Design Advisor to assist in optimizing queries. These tools need several tables. Create the tables in the SYSTOOLSPACE table space by using the following stored procedure:

```
CALL SYSPROC.SYSINSTALLOBJECTS('EXPLAIN','C','',CURRENT SCHEMA)
-- Uses SYSTOOLSPACE by default because the third parameter is
not specified
```

The explain tool adds rows to the explain tables but does not delete old entries. You can clean up the tables with a simple **DELETE** statement, as shown in the following example:

```
DELETE FROM EXPLAIN_INSTANCE WHERE EXPLAIN_TIME < CURRENT DATE -
2 MONTHS;
-- Cleans up other explain tables automatically by using foreign
keys
```

The compiler might take several seconds to optimize a query before the execution starts. The **db2batch** command with the **-i complete** parameter reports compile times, that is, prepare times. If they are too long, for example, longer than a fraction of the total run time, consider reducing the effort spent by the optimizer by issuing the following command:

```
DB2SET DB2_REDUCED_OPTIMIZATION=YES
```

There are situations where the compile time for complex queries might be very short, for example, less than 0.01 second, and it doesn't always mean that the optimizer reduced its optimizing efforts. This is the case for example if a query is in the package cache.

If there are many SQL statements that are almost identical except for the values of literals, the statement concentrator might reduce overall compile time. This method can work well in transactional databases but it is not recommended for data warehousing,

where distribution statistics are relevant. Queries with different constants might need different query plans.

Recommendation: Do not use the statement concentrator in a data warehouse system.

Physical data modeling

This section highlights some key recommendations that are important for physical modeling in data warehouse databases.

For more general information about physical data modeling, see *Best Practices: Data Life Cycle Management*: http://www.ibm.com/developerworks/data/bestpractices/lifecyclemanagement/

For more details on table partitioning and MDC, see *Database Partitioning, Table Partitioning, and MDC*: http://www.redbooks.ibm.com/abstracts/sg247467.html

For information about data compression, see *Best Practices: Deep Compression*: http://www.ibm.com/developerworks/data/bestpractices/deepcompression/.

Recommendations:

- Define keys of dimension tables as not null, even if you do not formally set the keys as primary keys. It is a good idea to also define all joined columns as not null. You can define columns as not null easily, as shown in the following example:

```
ALTER TABLE THETABLE ALTER COLUMN THECOL SET NOT NULL
```

  If the query compiler recognizes that a column cannot have null values, more optimizations become possible.

- Use the same data type, length, and precision for pairs of columns that you use in equijoins. Otherwise, the compiler does not consider hash joins.

- Define primary keys for all dimension tables. Define primary keys for fact tables only if needed.

- Before defining a primary key, create a corresponding unique index with a declarative name. This avoids lengthy names for system-generated indexes.

Foreign keys

A foreign key relationship is a relationship between two tables where a set of columns in one of the tables is a primary key and all values of the corresponding columns in the other table occur in the key columns. Such constraints that are defined by foreign keys are sometimes called referential constraints. In a data warehousing environment, foreign key relationships exist typically between the primary key columns of dimension tables and the corresponding foreign key columns of the fact tables. In a normalized data

model, the foreign key relationships occur as references to lookup tables. They can also occur between fact tables. For example, a table with itemized sales data might have a foreign key referencing a market basket table that has one entry per transaction.

Recommendation: If the database applications ensure the consistency of foreign keys, declare the foreign keys as NOT ENFORCED ENABLE QUERY OPTIMIZATION.

The main purpose of foreign keys is to guarantee the integrity of a database. In the data warehouse context, foreign key relationships are generally ensured by the applications that write data into the database. Verifying foreign key relationships again in the database causes unnecessary performance overhead. If the process that writes the data guarantees that the foreign keys are valid, define the relationship in the database as NOT ENFORCED.
Foreign keys can give the DB2 optimizer valuable hints as the estimated cardinality gets more precise and redundant joins can be eliminated. Include the ENABLE QUERY OPTIMIZATION option in the SQL declaration of the foreign key. This option is the default.

You must define the referenced column as primary key, or it must have a UNIQUE constraint. You must also define the column as NOT NULL, and it must have a unique index.

The following example shows the ALTER TABLE statement for creating a foreign key with the options described previously. The statement specifies that all values of the PROD_ID column of the SALES_FACT table occur in the PROD_ID column of the PRODUCT_DIM table.

```
CREATE UNIQUE INDEX PRODUCT_DIM__PROD_ID ON PRODUCT_DIM(PROD_ID);
ALTER TABLE PRODUCT_DIM ADD PRIMARY KEY (PROD_ID);

ALTER TABLE SALES_FACT
        ADD CONSTRAINT FK__PROD_ID FOREIGN KEY (PROD_ID)
        REFERENCES PRODUCT_DIM (PROD_ID)
        NOT ENFORCED
        ENABLE QUERY OPTIMIZATION;
```

Foreign keys give the DB2 optimizer valuable information for simplifying queries. Because there is no performance overhead if you specify them with the NOT ENFORCED option, you should define all foreign key relationships in a database. This assumes that the keys are consistent, that is, they have been checked before the database is updated. If the foreign keys are not consistent, queries might produce incorrect results.

Verifying that referential constraints are valid

If you want to check the validity of a referential constraint, you can temporarily change the foreign key constraint to ENFORCED. For example, you can set the foreign key FK__PROD_ID to ENFORCED and then reset it to NOT ENFORCED by issuing the following two statements:

```
ALTER TABLE SALES_FACT ALTER FOREIGN KEY FK__PROD_ID ENFORCED;
-- Check for errors returned by previous statement
ALTER TABLE SALES_FACT ALTER FOREIGN KEY FK__PROD_ID NOT
ENFORCED;
```

If the first statement ends successfully, the foreign key relationship is valid.

You can check keys that violate the constraint by using an SQL query, as shown in the following example. Before executing such a query, you must temporarily disable query optimization for the constraint because the DB2 optimizer might use query optimization to simplify the query in such a way that it does not return any rows.

```
ALTER TABLE SALES_FACT ALTER FOREIGN KEY FK__PROD_ID DISABLE
QUERY OPTIMIZATION;

SELECT DISTINCT ppd.PROD_ID
      FROM SALES_FACT ppd LEFT OUTER JOIN PRODUCT p
      ON ppd.PROD_ID = p.PROD_ID
      WHERE p.PROD_ID IS NULL;

ALTER TABLE SALES_FACT ALTER FOREIGN KEY FK__PROD_ID ENABLE QUERY
OPTIMIZATION;
```

Benefits of foreign keys

Foreign keys help the DB2 database manager to detect and remove redundant joins

Assume that a reporting tool submitted a query that joins a fact table with all its dimension tables and that the SELECT clause contains only columns from a subset of the dimension tables. In such a case, the DB2 database manager might execute joins with tables that do not contribute to the result of the query. With referential constraints between the fact and the dimension tables, however, the DB2 database manager recognizes that the joins with the unused dimension tables do not add, remove, or alter the values of any rows in the result set. Therefore, the joins can be removed from the query.

The following example illustrates how an unnecessary join can be removed. Figure 1 shows a representation of a star schema with PRCHS_PRFL_DTL (Purchase Profile Details) as the fact table and STORE, PRODUCT, and TIME as the dimension tables.

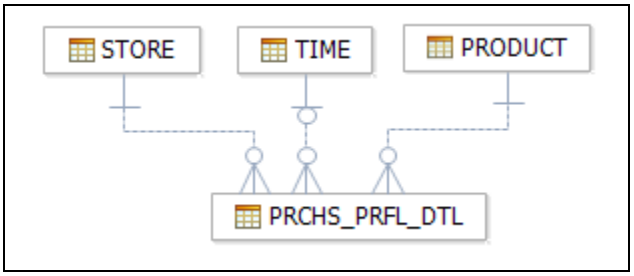

Figure 1. Sample star schema data model

The referential constraints are as follows:

- PRCHS_PRFL_DTL(PD_ID) references PRODUCT(PD_ID)
- PRCHS_PRFL_DTL(TIME_ID) references TIME(TIME_ID)
- PRCHS_PRFL_DTL(STORE_ID) references STORE(STORE_ID)

A query might select columns of the PRCHS_PRFL_DTL fact table and PRODUCT and TIME dimension tables but no columns of the STORE dimension table, as shown in the following example:

```
-- Original query
SELECT  PD_SUB_DEPT_ID,
      COUNT(*) AS NBTRANS,
      SUM(NUMBER_OF_ITEMS) AS SUM_NB_ITEMS,
      SUM(SALES_AMOUNT) AS SUM_SALES_AMOUNT
      FROM TIME T, PRODUCT P, STORE S, PRCHS_PRFL_DTL PPD
      WHERE PPD.PD_ID = P.PD_ID AND PPD.TIME_ID = T.TIME_ID AND
      PPD.STORE_ID = S.STORE_ID AND P.PD_DEPT_ID = 8
      AND T.CDR_YR=2008
      GROUP BY P.PD_SUB_DEPT_ID;
```

The join predicate `ppd.STORE_ID = s.STORE_ID` is true in all cases, and there is exactly one row in the STORE table that matches a row in the PRCHS_PRFL_DTL table. The join can be eliminated because the query does not use any other column from the STORE table. Because the foreign key constraint is declared in the database, the compiler recognizes that the join with the STORE table is redundant and eliminates the table. You can see the resulting optimized query in the explain plan.

```
-- Optimized query as shown in the explain plan
SELECT Q5.PD_SUB_DEPT_ID AS "PD_SUB_DEPT_ID", Q5.$C1
      AS "NBTRANS", Q5.$C2 AS "SUM_NB_ITEMS", Q5.$C3 AS
      "SUM_SALES_AMOUNT"
      FROM
      (SELECT Q4.PD_SUB_DEPT_ID, COUNT(*),
            SUM(Q4.NUMBER_OF_ITEMS),
            SUM(Q4.SALES_AMOUNT)
            FROM
            (SELECT Q2.PD_SUB_DEPT_ID, Q3.NUMBER_OF_ITEMS,
             Q3.SALES_AMOUNT
             FROM TIME AS Q1, PRODUCT AS Q2, PRCHS_PRFL_DTL AS Q3
       WHERE (Q1.CDR_YR = 2008) AND (Q2.PD_DEPT_ID = 8)
       AND (Q3.TIME_ID = Q1.TIME_ID) AND (Q3.PD_ID = Q2.PD_ID)
      ) AS Q4
      GROUP BY Q4.PD_SUB_DEPT_ID) AS Q5
```

Compression

Compression reduces the need for storage. Compression can also optimize performance because the DB2 software reads more rows per disk access and keeps more rows in the

buffer pool. More rows are kept in the buffer pool because the data is compressed. Values are decompressed dynamically when they are used, for example, in filters or joins. Compression saves I/O operations but generally takes more CPU cycles.

Recommendations:

- Consider compression when system monitoring shows many I/O waits. Do not add compression if the system is CPU bound.

- Estimate compression ratios with the ADMIN_GET_TAB_COMPRESS_INFO_V97 administration function. For example, analyze all tables in schema 'EDW' with the following statement:

```
SELECT * FROM
TABLE ( SYSPROC.ADMIN_GET_TAB_COMPRESS_INFO_V97('EDW', '',
'ESTIMATE'))
```

  Execution might take several minutes for large tables, especially if there are many partitions.

- For optimal compression on large tables, use the command **`REORG TABLE ... RESETDICTIONARY`**. The dictionary is based on a sample of the whole table rather than just the first 2 MB used, as with automatic dictionary creation. Also, issue the **`RUNSTATS`** command after issuing the **`REORG TABLE`** command.

A compressed table has one compression dictionary per partition. For example, if there are eight database partitions and a table is defined with 20 table partitions, there can be 160 different compression dictionaries. If there are multiple compression dictionaries, compression rates might differ for each partition.

Table partitioning

Table partitioning increases query performance through partition elimination. You can quickly roll in and roll out data online by using the **`ATTACH`** and **`DETACH`** clauses in the **`ALTER TABLE`** statement.
For overviews, see the following articles:

- http://www.ibm.com/developerworks/data/library/techarticle/dm-0605ahuja2/index.html
- http://www.ibm.com/developerworks/data/library/dmmag/DMMag_2010_Issue3/DistributedDBA/

Recommendations:

- If there would be more than a few hundred partitions in a table, consider using MDC instead of table partitioning.

  Table partitioning is often used with ranges of dates or other types of data that have many different values. Table partitioning can also help in cases when there are only two or three values. For example, there might be a flag that identifies old or logically deleted rows that are stored together with current data in one table. Queries that select current data can run faster because the optimizer can eliminate the partitions that do not contain current data in advance.

- Use partitioned indexes when possible. Create an index as NOT PARTITIONED in the following cases:

  - A partitioned index is sorted locally, not sorted globally. Therefore, there might be another sort operation after individual ranges are scanned.
  - If there are many short queries that select only a few records, all local indexes are evaluated. This can be slower than going through one global index.

- When you want to add a partition, create indexes on the new table that match the partitioned index of the existing partitioned table before the ATTACH operation. The indexes make the SET INTEGRITY statement step faster.

Multidimensional clustering

Multidimensional clustering (MDC) provides an efficient indexing method for large tables. When choosing columns to use as MDC dimensions, try to apply these guidelines equally:

- Choose columns that are useful in filter conditions.
- Use combinations of columns that have many rows per distinct value.

Each MDC cell, that is, every different combination of dimension values, allocates at least one block equivalent to one extent in the table space. If a block is not filled with rows, some space is wasted. Each database partition and each table partition has its own set of blocks per MDC cell.

If the table exists, you can inspect the size of the MDC cells by using a query similar to the following example:

```
-- Count number of cells and number of rows, split by partitions
SELECT COUNT(*) NBR_CELLS, MIN(n) MIN_ROWS, AVG(n) AVG_ROWS,
MAX(n) MAX_ROWS
FROM (
-- dimA, dimB, … dimensions as in the ORGANIZE BY clause
SELECT DP,TP, DIMA, DIMB, …, COUNT(*) AS n
FROM (SELECT DBPARTITIONNUM(DIMA),DATAPARTITIONNUM(DIMA), DIMA,
DIMB, …,
      FROM THETABLE )
GROUP BY DP,TP, DIMA, DIMB, …
)
```

Consider a table that has rows with an average row length of 100 bytes after compression. The corresponding table space might have a page size of 16 KB and 32 pages per extent. The space for an MDC block would be calculated as follows:

32 x 16 KB / 100 bytes = approximately 5000 records

Each MDC cell in this example has many more than 5000 records per database partition and per table partition. Preferably, the cells should be bigger by a factor of 5 or more to reduce wasted space.

An alternative method for checking the space usage of a table with MDC is to call the **REORGCHK** command with the **CURRENT STATISTICS** and **ON TABLE** parameters, as shown in the following example:

```
REORGCHK CURRENT STATISTICS ON TABLE THESCHEMA.THETABLE
```

The number of cells should be much fewer, about 20% of the number calculated by the following formula:

```
 NP / extent size / number of database partitions / number of
table partitions used = number of cells
```

Where the extent size is the number of pages per extent in the corresponding table space and NP is the estimated sum of the number of pages that contain data over all database partitions as reported in the result of REORGCHK.

Recommendations:

- For a table with MDC, check whether the total number of pages (FP) is close to the number of pages that contain data (NP). If FP is much higher than NP, space is likely being wasted because MDC blocks are not filled up.

- If a candidate MDC dimension has many different values yielding too many small cells, increase the granularity of cells with a generated column. For example, use a generated column with INTEGER (Date)/100 instead of Date.

Materialized query tables (MQTs)

The correct design of MQTs depends on the applications. This paper does not describe detailed best practices for designing MQTs.

Recommendations:

- Use compression for MQTs. Update the statistics for MQTs by issuing the **RUNSTATS** command.

- Test the performance impact of a new MQT by rewriting the relevant query to explicitly refer to the MQT table. Consider automatic MQT matching later, after the MQT sufficiently improves performance.

- If the DB2 optimizer does not automatically selects the MQT, inspect the Optimized Statement in the explain plan for both the query and the SELECT statement in the MQT definition. Modify the MQT definition to get closer to the Optimized Statement of the query.

You can force the DB2 optimizer to use a replicated MQT even if its cost estimate is higher than that of the corresponding base table. To force the optimizer to use a replicated MQT, issue the following command:

```
DB2SET DB2_EXTENDED_OPTIMIZATION=FORCE_REPLICATED_MQT_MERGE
```

The MQTENFORCE optimization profile element provides another method for enforcing the use of MQTs, as illustrated in the following example:

```
-- Create the OPT_PROFILE table in the SYSTOOLS schema
CALL SYSPROC.SYSINSTALLOBJECTS('OPT_PROFILES', 'C', '', '');
-- This creates the table SYSTOOLS.OPT_PROFILE.
-- Note that the table name does not have a trailing ‘S’.

INSERT INTO SYSTOOLS.OPT_PROFILE VALUES (
'MYSCHEMA','MQTENFORCE_ALL',
CAST('<OPTPROFILE><OPTGUIDELINES><MQTENFORCE
TYPE="ALL"/></OPTGUIDELINES></OPTPROFILE>'
      AS BLOB(2M))
);

SET CURRENT OPTIMIZATION PROFILE MYSCHEMA.MQTENFORCE_ALL;
```

The definition TYPE="ALL" enforces the use of both regular MQTs and replicated MQTs. It is also possible to list specific MQTs by name. For further details, see http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0024526.html.

Functional dependencies

A functional dependency is a relationship between two columns of a table that can give the DB2 optimizer valuable information. Functional dependencies usually exist in a denormalized table between the ID of entities, such as department ID, and their descriptions, such as department name. Functional dependencies also exist between such IDs and the ID of the entities at the next hierarchy level, for example, between department ID and sub-department ID. Foreign keys in a normalized model usually correspond to functional dependencies in a denormalized model.

You can define functional dependencies as constraints with statements similar for the following, for example. As shown in the statements, you can define functional dependencies as NOT ENFORCED. If you define the dependencies as NOT ENFORCED, you must make sure (for example, through the ETL process) that relationships that are described by the functional dependencies are valid. Furthermore, you must define the column on the right side of the functional dependency as NOT NULL.

```
ALTER TABLE PRODUCT
        ADD CONSTRAINT PD_DEPT__SUB_DEPT_FD CHECK
                ( PD_DEPT_ID DETERMINED BY PD_SUB_DEPT_ID)
        NOT ENFORCED
        ENABLE QUERY OPTIMIZATION;

ALTER TABLE PRODUCT
        ADD CONSTRAINT PD_DEPT_FD CHECK
                ( PD_DEPT_NM DETERMINED BY PD_DEPT_ID)
        NOT ENFORCED
        ENABLE QUERY OPTIMIZATION;
```

You can check the validity of a functional dependency through an SQL query. The following SQL query checks the functional dependency between the PD_SUB_DEPT_ID and PD_DEPT_ID columns:

```
SELECT PD_SUB_DEPT_ID, COUNT(*), MIN(PD_DEPT_ID), MAX(PD_DEPT_ID)
      FROM (SELECT DISTINCT PD_SUB_DEPT_ID, PD_DEPT_ID
      FROM PRODUCT)
      GROUP BY PD_SUB_DEPT_ID
      HAVING COUNT(*) > 1;
```

The SQL statement returns those PD_SUB_DEPT_ID column values that have more than one associated PD_DEPT_ID column value with the number of these PD_DEPT_ID values and two sample values (the maximum and minimum). It is not necessary to disable the constraint for query optimization to ensure that the query returns rows in case of an invalid functional dependency.

Guidelines for defining functional dependencies

Recommendation: Define functional dependency constraints for all functional relationships in a table that can be guaranteed by the process that writes the data. It is not necessary to define functional dependencies for a primary key or a unique index column because the DB2 database manager recognizes that implicit functional relationships exist for the other columns of a table.

Because functional dependencies do not cause overhead at run time, you can define them for all valid functional relationships. The validity must be ensured, for example, by the ETL process.

Benefits of functional dependencies

You can use functional dependencies to define smaller MQTs.

Assume that the PRCHS_PRFL_SUB_DEPT_ID_YEAR_MQT MQT is defined by using the following statements. The MQT definition computes some aggregated values for each combination of sub-department ID and calendar year.

```
CREATE TABLE PRCHS_PRFL_SUB_DEPT_ID_YEAR_MQT AS (
      SELECT PD_SUB_DEPT_ID,CDR_YR,
            COUNT(*) AS NBTRANS,
            SUM(NUMBER_OF_ITEMS) AS SUM_NB_ITEMS,
            SUM(SALES_AMOUNT) AS SUM_SALES_AMOUNT
            FROM PRCHS_PRFL_DTL PPD, PRODUCT P, TIME T
            WHERE PPD.PD_ID = P.PD_ID AND
            PPD.TIME_ID = T.TIME_ID
            GROUP BY P.PD_SUB_DEPT_ID, T.CDR_YR
      )
      DATA INITIALLY DEFERRED REFRESH DEFERRED
      IN TS_SD_001;

REFRESH TABLE PRCHS_PRFL_SUB_DEPT_ID_YEAR_MQT;
RUNSTATS ON TABLE SCHEMA.PRCHS_PRFL_SUB_DEPT_ID_YEAR_MQT WITH
      DISTRIBUTION AND INDEXES ALL;
```

Consider the following query:

```
SELECT PD_SUB_DEPT_ID, COUNT(*) AS NBTRANS,
      SUM(NUMBER_OF_ITEMS) AS
        SUM_NB_ITEMS, SUM(SALES_AMOUNT) AS SUM_SALES_AMOUNT
      FROM PRCHS_PRFL_DTL PPD, PRODUCT P, TIME T
      WHERE PPD.PD_ID = P.PD_ID AND PPD.TIME_ID = T.TIME_ID AND
      P.PD_DEPT_ID = 8 AND T.CDR_YR=2008
      GROUP BY P.PD_SUB_DEPT_ID;
```

The query references the department ID (`PD_DEPT_ID = 8`) in the WHERE clause. Values in PD_DEPT_ID can be inferred from the values in PD_SUB_DEPT_ID because of the functional dependency between the PD_SUB_DEPT_ID and PD_DEPT_ID columns. Therefore, the PRCHS_PRFL_SUB_DEPT_ID_YEAR_MQT table can be used, and the optimized statement of this query is as follows:

```
SELECT Q4.PD_SUB_DEPT_ID AS "PD_SUB_DEPT_ID", Q4.$C2
      AS "NBTRANS", Q4.$C1 AS
        "SUM_NB_ITEMS", Q4.$C0 AS "SUM_SALES_AMOUNT"
      FROM
      (SELECT SUM(Q3.SUM_SALES_AMOUNT),
       SUM(Q3.SUM_NB_ITEMS), SUM(Q3.NBTRANS),
       Q3.PD_SUB_DEPT_ID
      FROM
      (SELECT DISTINCT Q2.PD_SUB_DEPT_ID, Q2.SUM_SALES_AMOUNT,
              Q2.SUM_NB_ITEMS, Q2.NBTRANS, Q1.PD_DEPT_ID,
            Q1.PD_SUB_DEPT_ID
            FROM PRODUCT AS Q1,
            PRCHS_PRFL_SUB_DEPT_ID_YEAR_MQT AS Q2
            WHERE (Q1.PD_DEPT_ID = 8) AND
            (Q2.CDR_YR = 2008) AND (Q2.PD_SUB_DEPT_ID
              = Q1.PD_SUB_DEPT_ID)) AS Q3
   GROUP BY Q3.PD_SUB_DEPT_ID) AS Q4
```

Without the functional dependency, you would have to include the PD_DEPT_ID column in the SELECT and GROUP BY clauses of the MQT.

Consider the following query:

```
SELECT PD_SUB_DEPT_ID, COUNT(*) AS NBTRANS, SUM(NUMBER_OF_ITEMS)
AS
      SUM_NB_ITEMS, SUM(SALES_AMOUNT)
      AS SUM_SALES_AMOUNT
      FROM PRCHS_PRFL_DTL PPD, PRODUCT P, TIME T
      WHERE PPD.PD_ID = P.PD_ID AND PPD.TIME_ID = T.TIME_ID AND
      P.PD_DEPT_NM = 'Giveaways' AND T.CDR_YR=2008
      GROUP BY P.PD_SUB_DEPT_ID;
```

This query is identical to the previous unoptimized one except that it includes `p.PD_DEPT_NM = 'Giveaways'` in the WHERE clause. In this case, the functional dependency between the PD_DEPT_ID and PD_DEPT_NM columns also makes it possible to use the MQT. The optimized statement of the query is as follows:

```
SELECT Q4.PD_SUB_DEPT_ID AS "PD_SUB_DEPT_ID", Q4.$C2
      AS "NBTRANS", Q4.$C1 AS
        "SUM_NB_ITEMS", Q4.$C0 AS "SUM_SALES_AMOUNT"
      FROM
      (SELECT SUM(Q3.SUM_SALES_AMOUNT), SUM(Q3.SUM_NB_ITEMS),
       SUM(Q3.NBTRANS),Q3.PD_SUB_DEPT_ID
      FROM
      (SELECT DISTINCT Q2.PD_SUB_DEPT_ID, Q2.SUM_SALES_AMOUNT,
              Q2.SUM_NB_ITEMS, Q2.NBTRANS, Q1.PD_DEPT_NM,
             Q1.PD_SUB_DEPT_ID
      FROM PRODUCT AS Q1, PRCHS_PRFL_SUB_DEPT_ID_YEAR_MQT AS Q2
      WHERE (Q1.PD_DEPT_NM = 'Giveaways')
            AND (Q2.CDR_YR = 2008) AND
              (Q2.PD_SUB_DEPT_ID = Q1.PD_SUB_DEPT_ID)) AS Q3
      GROUP BY Q3.PD_SUB_DEPT_ID) AS Q4
```

Without the functional dependency, you would have to include the additional PD_DEPT_NM column in the SELECT and GROUP BY clauses of the MQT.

Gathering statistics for optimization

The DB2 compiler needs accurate statistics to select the optimal query execution plan. If the estimated number of rows in the explain plan is considerably different from the actual number of rows, the chosen operators might be far from optimal. The more accurate information you collect with the **RUNSTATS** command, the higher the chances that the DB2 optimizer can find a good execution plan.

The simplest method for collecting statistics is to enable automatic invocations of the **RUNSTATS** command, as shown in the following example. Automatic invocations are the default in DB2 Version 9.7.

```
DB2 UPDATE DB CFG USING AUTO_RUNSTATS ON
```

Execution of the **RUNSTATS** command tends to be CPU intensive. If you prefer to manually control the invocations of the **RUNSTATS** command and its performance impact on the production workload, use the following recommendations.

Recommendations:

- Collect statistics on all tables and keep them current.
- When a **RUNSTATS** command is completed, issue a COMMIT statement to release locks on catalog tables.

- For system tables, issue the following command:

```
REORGCHK UPDATE STATISTICS ON TABLE SYSTEM
```

- Use the **RUNSTATS** command to gather distribution and index statistics:

```
RUNSTATS ON TABLE SCM.TAB WITH DISTRIBUTION AND SAMPLED
DETAILED INDEXES ALL
```

If you want to reduce the run time for the **RUNSTATS** command on a large table, potentially losing some precision, use sampling for the table and for the indexes. You can use a smaller table sample for large tables.

```
RUNSTATS ON TABLE SCM.TAB WITH DISTRIBUTION AND SAMPLED
DETAILED INDEXES ALL TABLESAMPLE SYSTEM(10)
```

You can further reduce run time by collecting statistics on a subset of columns. If you want to minimize the performance impact of running **RUNSTATS**, issue the command with options similar to the following example, at a minimum:

```
RUNSTATS ON TABLE SCM.TAB ON KEY COLUMNS WITH DISTRIBUTION ON
KEY COLUMNS TABLESAMPLE SYSTEM(1)
```

Specify the **ON KEY COLUMNS** option in the **WITH DISTRIBUTION** clause. Otherwise, the **RUNSTATS** command collects distribution statistics on all columns.

Running the **RUNSTATS** command with all details on a large table with 50,000,000 rows per database partition might take several minutes. Sampling would reduce run time to about 1 minute. Using the option shown in the previous example might reduce the run time to 10 seconds.

If you issue the **RUNSTATS** command multiple times, statistics are not accumulated. For example, the first command in the following command sequence collects distribution statistics. The second command collects statistics on the index tmp.i0, and the distribution statistics are not retained.

```
RUNSTATS ON TABLE TMP.TAB0 WITH DISTRIBUTION
RUNSTATS ON TABLE TMP.TAB0 AND INDEXES TMP.I0
```

Recommendations:

- When you create an index for a populated table for which the RUNSTATS command had already collected statistics information, add the index statistics while the new

index is being created. The following example shows fragments of the CREATE INDEX statement with the relevant options:

```
DB2 CREATE INDEX … COLLECT SAMPLED DETAILED STATISTICS
```

- Store the complete list of the RUNSTATS command options in a profile. Use the profile when invoking the RUNSTATS command. Using a profile that contains the options ensures that you apply the options consistently.

```
RUNSTATS ON TABLE SCM.TAB … COMPLETE COMMAND … SET PROFILE
ONLY
-- Any time later …
RUNSTATS ON TABLE SCM.TAB USE PROFILE

-- Check profile
SELECT STATISTICS_PROFILE FROM SYSCAT.TABLES WHERE
(TABSCHEMA,TABNAME) = ('SCM','TAB')
```

For further guidelines on collecting and updating statistics, see http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.admin.perf.doc/doc/c0005083.html.

Changes to the number of frequency values and quantiles

The collection of frequency values and quantiles is important when the distribution of column values is skewed, that is, when there are several values with high frequency and many others with low frequency. For some of these cases, it can help to increase the value of the **NUM_FREQVALUES** or **NUM_QUANTILES** parameter of the **RUNSTATS** command. The default value of the **NUM_FREQVALUES** parameter is 10, and the default value of the **NUM_QUANTILES** parameter is 20. In the following example, the values of the parameters are changed:

```
RUNSTATS ON TABLE tmp.tab WITH DISTRIBUTION ON ALL COLUMNS AND
COLUMNS (skewedcol NUM_FREQVALUES 40 NUM_QUANTILES 40)
```

If a query filters on one of the frequently occurring column values and this value is in the list that the **RUNSTATS** command collects, the optimizer can precisely estimate the cardinality of the results. If the **RUNSTATS** command did not retain the filter value, the estimate might be too small.

The sample SQL query that is in the `eval-num_freqvalues.sql` file in the `ISASBP_Query_Optimization_Scripts.zip` file returns a list of columns that might benefit from increased **NUM_FREQVALUES** parameter values. The query returns candidates that have a skewed value distribution. Pick those columns that you use in filters in the production workload.

Increasing the value of the **NUM_QUANTILES** parameter is less likely to speed up performance. The default value for the **NUM_QUANTILES** parameter gives reasonably accurate estimates in most cases. Cardinality estimates that are based on quantiles tend to be conservative, meaning that the actual number of rows is not much higher than that estimated by the optimizer. You might want to increase the number of quantiles if an explain plan shows significant differences between estimated and actual cardinalities. Increasing the number of quantiles can be relevant for range predicates on datetime columns that contain extreme sentinel values, such as date('9999-12-31').

Recommendation: Specify non-default values for the **NUM_FREQVALUES** parameter or the **NUM_QUANTILES** parameter only for specific tables in the **RUNSTATS** command or profile. Changes to the default values in the database configuration apply to all tables and all columns. In most cases, changing the default values in the database configuration does not improve performance.

The **RUNSTATS** command **LIKE STATISTICS** column option collects sub-element statistics. These statistics can be useful if a column contains lists of values that are separated by blanks. The **RUNSTATS** command computes the average length of the values, but it does not collect frequencies of individual values. For further details, see http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.admin.perf.doc/doc/c0005098.html.

When to issue the RUNSTATS command

Issue the **RUNSTATS** command when you have inserted or changed a substantial amount of data in the table, for example, more than 30%.

The sample script that is in the `runstats_out_of_date.sh` file in the `ISASBP_Query_Optimization_Scripts.zip` file reports tables with outdated **RUNSTATS** command information. Similar logic is used for automatic updates when **AUTO_RUNSTATS** is enabled.

After you reorganize a table or an index, issue the **RUNSTATS** command again because it collects information about the physical organization, such as index clustering ratios.

Statistical views

Statistical views help to improve the cardinality estimates for the results of joins. The **RUNSTATS** command determines the distribution of the values of a column within a table. If a table is joined with other tables, the DB2 optimizer assumes that the distribution does not change significantly. In a data warehouse scenario, this assumption is not always realistic.

Consider the execution plan of the following query, illustrated in Figure 2. The query determines certain aggregated values for the sub-departments of the department with a value of 8 in the PD_DEPT_ID column before 2008. The query does not define a statistical view.

```
SELECT PD_SUB_DEPT_ID,
       COUNT(*) AS NBTRANS,
       SUM(NUMBER_OF_ITEMS) AS SUM_NB_ITEMS,
       SUM(SALES_AMOUNT) AS SUM_SALES_AMOUNT
      FROM PRCHS_PRFL_DTL PPD, PRODUCT P, TIME T
      WHERE PPD.PD_ID = P.PD_ID AND
      PPD.TIME_ID = T.TIME_ID AND
      P.PD_DEPT_ID = 8 AND
      T.CDR_YR<2008
      GROUP BY P.PD_SUB_DEPT_ID;
```

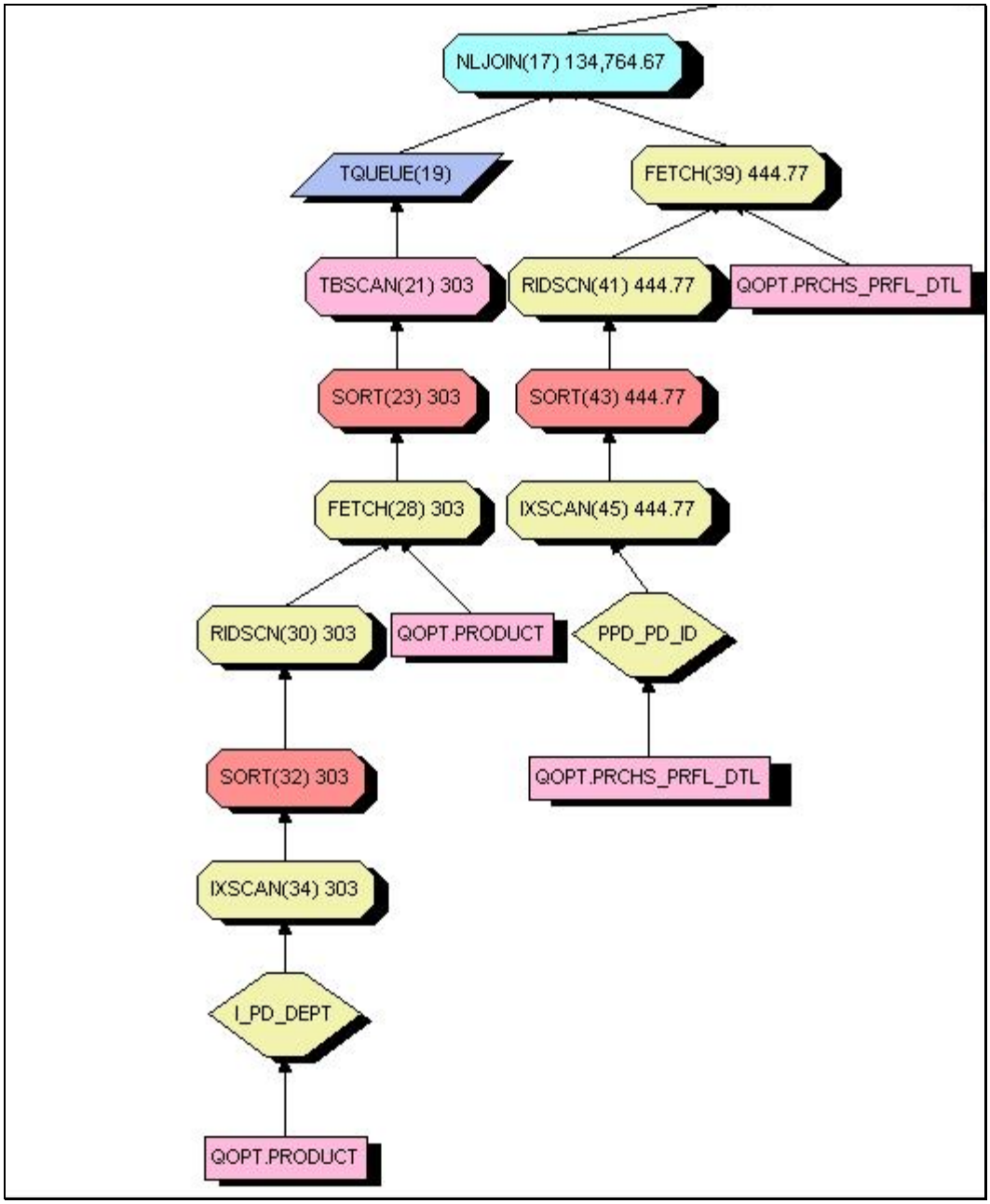

Figure 2. Explain plan without statistical view

Operator 17 at the top of this part of the plan shows that the optimizer estimates the cardinality of the join between the PRODUCT and PRCHS_PRFL_DTL tables when the value of the PD_DEPT_ID column is 8 to be 134,764 rows. This estimate is based on the distribution of values in the PD_DEPT_ID column in the PRODUCT table. The actual number of rows in the result of the join in this example is much higher, around 5,000,000 rows. The query plan has a nested loop join to perform the join, which is appropriate only if the number of rows is not too high.

The plan can be improved by the following statistical view:

```
CREATE OR REPLACE VIEW PRCHS_PRFL_DTL_DEPT_ID_SV AS
SELECT PD_DEPT_ID
      FROM prchs_prfl_dtl ppd, product p
      WHERE ppd.PD_ID = p.PD_ID;

ALTER VIEW PRCHS_PRFL_DTL_DEPT_ID_SV ENABLE QUERY OPTIMIZATION;

RUNSTATS ON TABLE Schema.PRCHS_PRFL_DTL_DEPT_ID_SV WITH
DISTRIBUTION;
```

Important: Ensure that you include the WITH **`DISTRIBUTION`** parameter when you issue the **`RUNSTATS`** command; otherwise, the command does not compute the frequencies of the values of the columns in the select-clause.

A section of the new plan is illustrated in Figure 3:

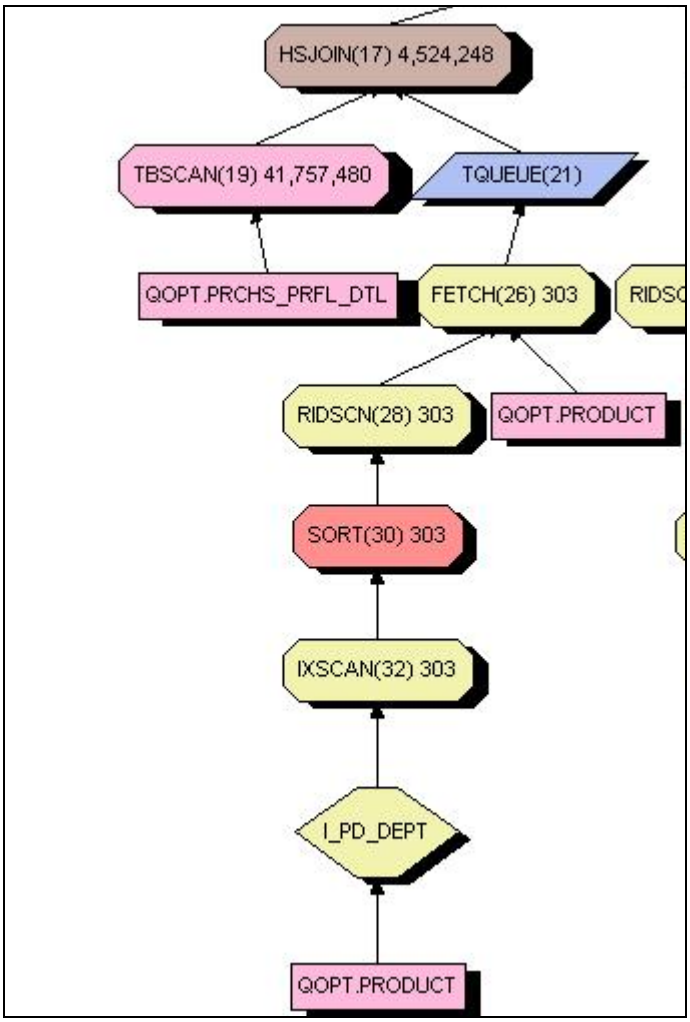

Figure 3. Explain plan optimized with statistical view

The estimated cardinality for this join is now 4,524,248 rows. This is close to the correct cardinality. Because of the higher cardinality, the optimizer now chooses a hash join, which improves performance for this query by approximately 50 %.

Performance of the RUNSTATS command on statistical views

Running the **RUNSTATS** command on a statistical view evaluates the view definition. If any table in the view definition is distributed, the data from all database partitions is used. This behavior is different from that of running the **RUNSTATS** command directly on a table because the latter reads data from only one database partition.

Recommendation: For statistical views that contain distributed tables, issue the **RUNSTATS** command with the **TABLESAMPLE SYSTEM** parameter and a sample rate of (100 / *number of database partitions*)%. In some cases, the DB2 database manager might replace the **TABLESAMPLE SYSTEM** parameter with the **TABLESAMPLE BERNOULLI** parameter. This is fine, and you can ignore corresponding warnings.

You can further improve the performance of the **RUNSTATS** command by limiting the number of columns in the select-clause of the statistical view definition. For example, select only columns that are used in the WHERE clauses of queries.

Recommendation: A good initial approach for defining a statistical view for a join between a fact table and the tables of one of its dimensions is to include in the select-clause at least the columns that contain the IDs and descriptions of the members of the hierarchy levels in the dimension.

Tools

Design Advisor

Using a set of queries, the Design Advisor can recommend the creation of regular indexes, MDC, and MQTs. You invoke the Design Advisor by issuing the **db2advis** command.

The Design Advisor needs detailed statistics because the recommendations are based on query plans. If the estimated and actual numbers of rows in query plans are considerably different, the results from the Design Advisor might not be useful. Collect detailed statistics before running the Design Advisor.

The indexes that the Design Advisor recommends often include many columns. These indexes support fast index-only access to data, but they require additional maintenance if you modify the corresponding table.

For regular indexes that the Design Advisor recommends, check whether you can use a corresponding MDC dimension instead. The Design Advisor can provide recommendations for MDC dimensions. The recommended MDC dimensions might be too fine grained if the data is partitioned.

Evaluating candidate indexes

Creating a new index to try to optimize a certain workload can take a significant amount of time, depending on the size of the table Also, after you create the index, you might want to check how the index affects explain plans, and before running the explain tool, you should collect statistics, which requires even more time.

Recommendation: Before creating an actual index, use the DB2 Design Advisor **db2advis** command to quickly define a virtual index, and then check whether query performance improves.

The **db2advis** command maintains a table called ADVISE_INDEX that describes the recommended indexes. The information in this table includes the names of the indexed columns. The **db2advis** command also creates statistics for this index that are based on the statistics of the underlying table. The explain tool uses the entries in the

ADVISE_INDEX table if the **CURRENT EXPLAIN MODE** special register has the value **EVALUATE INDEXES** set.

Consider a table called **THETABLE** that has two columns, **C1** and **C2**, which you want to use in an index. Issue the **db2advis** command with a single query that filters on these columns:

```
DB2ADVIS -D TESTDB -S "SELECT COUNT(*) FROM THETABLE WHERE C1=0 AND C2=0"
# Will likely recommend an index on THETABLE(C1,C2)
# Check the new entry in the table ADVISE_INDEX
DB2 SELECT SUBSTR(COLNAMES,1,30) FROM ADVISE_INDEX WHERE
TBNAME='THETABLE'
# And generate new explain plan for a query
DB2 SET CURRENT EXPLAIN MODE EVALUATE INDEXES
DB2 ... THE QUERY...
DB2 SET CURRENT EXPLAIN MODE NO
DB2EXFMT -D TESTDB -1
```

The resulting explain plan might use the virtual index that you defined by using the **db2advis** command.

The db2advis command is not guaranteed to come up with the index that you want. You can try different constants in the WHERE clause. If the **db2advis** command does not recommend a particular index, the optimizer would probably not use it either.

For general tips on designing indexes, see http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.dbobj.doc/doc/c0020181.html. For information about virtual indexes, see http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.sql.ref.doc/doc/r0005874.html.

Explain plans with actual cardinalities

Explain plans include the estimated number of rows that an operator returns. You can extend the plans by the actual number of rows that were processed in the query at run time. For an example, see the following documentation:

- http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.admin.perf.doc/doc/c0056362.html
- http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0056464.html

Recommendation: If there are major differences between the estimated and actual numbers of rows, for example, by a factor of 2 or more, the compiler might not choose the optimal plan. Make the statistics more precise so that the number of rows that the optimizer estimates is closer to the actual number.

The explain plan shows how many rows were processed per partition. You can use this information to detect skews in the workload across partitions.

The sample script in the `gen_explain_actuals.sh` file in the `ISASBP_Query_Optimization_Scripts.zip` file combines all relevant commands to run a query and create an access plan with actuals.

Note that the computation of actual cardinalities uses monitor tables. The table space for these tables needs to be set up properly.

Recommendation: Make sure that the monitor tables are available on all partitions, by issuing a **`CREATE TABLESPACE`** statement similar to the one in the following example:

```
-- Setup
UPDATE DATABASE CONFIGURATION USING SECTION_ACTUALS BASE;
CREATE TABLESPACE Monitor IN DATABASE PARTITION GROUP
IBMDEFAULTGROUP … ;
CREATE EVENT MONITOR TheEventName FOR ACTIVITIES WRITE TO TABLE
    ACTIVITY (IN MONITOR),
    ACTIVITYSTMT (IN MONITOR),
    ACTIVITYVALS (IN MONITOR),
    CONTROL (IN MONITOR)
    MANUALSTART;
```

If you do not correctly set up the monitor tables, some results in the explain plan might be wrong or missing.

Using explain plans to identify potential problems

If there is a performance problem with a specific query, the explain plan can reveal the costly operations. This section provides hints and tips on analyzing explain plans and influencing the optimizer to produce better plans. The examples in this section refer to plans that are generated by the **`db2exfmt`** command.

This section requires a basic understanding of how to generate and read query explain plans. For an overview of the explain facility, see http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0005134.html. The explain operators are described in the following documents:

- http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.explain.doc/doc/r0052023.html
- http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?nav=/3_6_4_5_82

You can gather detailed explain information by combining the following example commands:

```
DB2BATCH –D THEDB -O E EXPLAIN -I COMPLETE –F THEQUERY.SQL
DB2EXFMT –D THEDB -1 > THEQUERY.EXFMT.OUT
GEN_EXPLAIN_ACTUALS.SH  THEDB THEQUERY.SQL
```

The explain plan, `TheQuery.exfmt.out`, includes lists of columns for input and output streams and selected distribution statistics. The `gen_explain_actuals.sh` script in the `ISASBP_Query_Optimization_Scripts.zip` file uses the **`db2exfmt`** command to format the explain plan. The output includes the actual number of rows that the operators processed at run time.

In this section, the analysis of explain plans is described with patterns. Each pattern describes combinations of elements that you might find in an explain plan that the **`db2exfmt`** command generates. These patterns indicate potential performance problems. There are corresponding actions that can help you optimize query execution.

The patterns refer to two types of issues: issues with the estimated cardinality for an operator and issues with the execution of an operator. Address the first kind of issues first. They usually require more detailed statistical information.

Explain plans include a section called `Extended Diagnostic Information` immediately following the plan graph. This section might contain warning messages, for example, about missing statistics for tables or indexes. In the following descriptions of patterns, it is assumed that such messages have been addressed. In particular, it is assumed that statistics are current.

Estimated cardinality

The analysis of query plans is effective only if the estimated number of rows that are returned per operator is close to the actual number of rows that are returned at run time. Cost-based optimization directly depends on accurate cardinality estimates. If the cardinality estimates are wrong, the estimated costs are also wrong, which can cause poor performance when you run the query. With more precise cardinality estimates, the compiler can usually generate a better plan. Statistics are used to estimate how many rows a certain join or other operator returns.

The analysis of a particular query might reveal that you must refine certain statistics. For example, the optimizer usually assumes that columns are statistically unrelated. However, real-life data often contains related attributes. You might need detailed statistics, including column groups or statistical views, to adjust the estimates. These refined statistics usually improve not only the performance of a single query but also the performance of other queries that use the same data.

With DB2 V9.7, you can create explain plans that include the actual number of rows that were processed by each operator. For more details, see the “Explain plans with actual cardinalities” section.

Recommendation: Use **section actuals** information in the **`EXPLAIN`** output together with **`db2exfmt`** command output to analyze query plans and check whether the cardinality estimates are in line with the actual number of rows. Focus on operators where the estimated number of rows is very small.

```
        Rows
      Rows Actual
        RETURN
        (   1)
         Cost
          I/O
          |
        3.21948  << The estimated rows used by the optimizer
          301    << The actual rows collected at run time
         DTQ
        (   2)
        75.3961
          NA
```

If the number of actual rows is much higher than estimated number of rows, you might run into various performance problems. An important example is a nested loop join with many rows in the left outer branch. This situation leads to many repeated executions of the operators in the right inner branch. Performance can be much better with a hash or merge join. The right inner branch of a nested loop might have an estimated cardinality of 1 or fewer because these cardinality values are implicitly multiplied by the number of rows in the outer/left branch of the loop.

Recommendation: After you identify an operator where the actual number of rows and the estimated number of rows are different, walk through the graph, from the top down and in order of execution to find the nodes where the numbers start to deviate. Try to adjust the estimate with more detailed statistics. The left outer branches of NLJOIN operators are important.

Pattern: NLJOIN operator. The estimated cardinality in the result of the left outer branch is too low.

```
                      6.6635    << Estimated rows
                   1.17031e+07 << Actual rows at run time
                      NLJOIN
                      (   7)
                      122.716
                        NA
      /------------------+------------------\
0.018402   <<estimated                     362.107
  33176    <<actual                        352.759
 ^NLJOIN                                   FETCH
 (   8)                                    (  26)
 98.9503                                   23.7659
```

Action: Fix the cardinality estimate in the left outer branch. Look for patterns in that branch.

Pattern: The estimated cardinality is wrong for the result of a table scan (represented by the TBSCAN operator) or index scan (represented by the IXSCAN operator).

Action: Check the filter conditions in the detailed description of the operator. If the filter factors that are shown in the explain plan are off for an equality predicate, increasing the value of the **NUM_FREQVALUES** parameter of the **RUNSTATS** command might help. For an inequality or LIKE predicate, increase the value of the **NUM_QUANTILES** parameter of the **RUNSTATS** command. See the example in the "Gathering statistics for optimization" section. If you are filtering the table on multiple columns, you can add the corresponding column group statistics to the **RUNSTATS** command. If the predicate involves an expression such as DATE(ts), a generated column might help. See the pattern "Table scan with many rows on input and few rows on output" later in this section.

Pattern: The estimated cardinality is wrong for the result of a join. The join predicates use more than one column in any of the tables.

The join columns might be correlated. This usually leads to a cardinality estimate that is lower than the actual number of rows that the join returns. The reason for the lower estimate is that the compiler assumes that a match on one column is unrelated to a match on the other column.

Action: You can improve the estimate by using column group statistics. For example, consider a join such as the following one:

```
Txn.Cust_Id = Bal.Cust_Id AND Txn.Contract_id = Bal.Contract_Id
```

Refine the statistics on both joined tables by issuing the **RUNSTATS** command:

```
RUNSTATS ON TABLE MySchema.Txn ON ALL COLUMNS AND COLUMNS
((Cust_Id,Contract_ID))
    WITH DISTRIBUTION AND SAMPLED DETAILED INDEXES ALL;
RUNSTATS ON TABLE MySchema.Bal  ON ALL COLUMNS AND COLUMNS
((Cust_Id,Contract_ID))
    WITH DISTRIBUTION AND SAMPLED DETAILED INDEXES ALL;
```

Pattern: The estimated cardinality is wrong for the result of a join. Some filtering is applied on an input table.

The following example has a filter on each input table, `(Q1.F = 3)` and `(Q2.ATTR = 1001)` respectively

```
Optimized Statement:
-------------------
...
      (SELECT $RID$
      FROM TMP.FACT AS Q1, TMP.DIM AS Q2
      WHERE (Q1.F = 3) AND (Q2.KEY = Q1.KEY) AND (Q2.ATTR =
1001))
...

3) NLJOIN: (Nested Loop Join)
                        Predicate Text:
                        --------------
                        (Q2.KEY = Q1.KEY)
```

Action: Create a statistical view on the joined tables. Look at the filter conditions for both outer and inner in the `Optimized Statement` section in the explain plan. Include the filtered columns in the SELECT clause. The WHERE clause uses the join condition but no filtering.

```
CREATE VIEW ... AS
SELECT OUTER.FILTERCOLS, INNER.FILTERCOLS FROM OUTER, INNER
WHERE ...THEJOIN...
```

In the following example, a statistical view is created on the Fact and Dim tables:

```
CREATE VIEW SV_FACT_DIM AS
SELECT FACT.F, DIM.ATTR FROM FACT,DIM
WHERE FACT.KEY = DIM.KEY
```

Collect statistics as described in the “Statistical views” section.

Optimizing the operators in an execution plan

After you sufficiently improve the statistics, you can look for expensive operators in the query execution plan. In some cases, it might be possible to improve the performance of the operators by changing the physical design of the tables. The following patterns help you find such cases.

Recommendation: After the estimated cardinalities match the actual cardinalities, walk top-down through the explain graph, following the path with the highest cost. Check for the patterns described later in this section. Operators with a high cost compared to the costs of the inputs are critical ones to optimize.

Pattern: A table scan has many input rows but few output rows.

Table scans are common in data warehouse queries. These scans are useful if many rows, for example, more than 50%, match the filter condition. When filter conditions are more restrictive there might be a better access method.

Action: Check whether the filter conditions can be supported by table partitioning, MDC, or a regular index.

If a query uses an expression or a type cast (explicit or implicit) on a column, an index might not be suitable for query execution. In this case, a generated column can help. For example, if you define a table as THETABLE(ts timestamp), a query such as the following one might result in a table scan even if there is an index on the column ts.

```
SELECT COUNT(*) FROM THETABLE WHERE DATE(TS) = '2010-10-31'
```

You can optimize this query by using a generated column:

```
SET INTEGRITY FOR MYTABLE OFF;
ALTER TABLE THETABLE ADD COLUMN DATE_TS GENERATED ALWAYS AS (
DATE(TS) );
SET INTEGRITY FOR THETABLE IMMEDIATE CHECKED FORCE GENERATED;
CREATE INDEX THETABLE_DATE ON MYTABLE(DATE_TS);
```

This method also works for MDC indexing. You can also use generated columns to make MDC cells more coarse grained.

Table queues

Table queues with a small amount of data, for example, up to 10,000 rows, are usually not problematic. A broadcast table queue (BTQ) is used when data from a small dimension table from the administration node is sent to all data partitions. Table queues

that send a significant amount of data often indicate a problem with joins on tables that are not collocated.

First, ensure that joined tables are in the same partitioning group, that is, they use the same partitioning map. Otherwise, you cannot use collocated joins. Standard implementations of IBM Smart Analytics System use only one partitioning group for partitioned user data.

The partitioning columns are shown in the operator details of the explain plan:

```
Partition Map ID: 5
Partitioning: (MULT ) Multiple Partitions

Partition Column Names:
-----------------------
+1: CUST_KEY
```

You can find the partitioning group that corresponds to a particular partition map in the catalog by using a query similar to the following example:

```
SELECT DBPGNAME FROM SYSCAT.DBPARTITIONGROUPS WHERE PMAP_ID = 5
```

You can avoid table queues by taking the following actions:

- By replicating data with MQTs
- By redistributing of a subset of columns with MQTs
- By modifying the distribution key

Pattern: A BTQ on a table has many rows, for example, 100,000 or more rows.

BTQs are often applied for small dimension tables that are stored in a single partition on the administration node. A join with a large fact table on the data nodes needs data from the dimension table in the database partitions on the data nodes. The BTQs usually transfer data from a subset of the columns rather than from the whole table. This behavior is usually not an issue for small tables but can affect performance when the amount of transferred data grows.

Action: Replicate the table with an MQT.

Replication with an MQT creates a new table in a partitioned table space. Replication copies data to each partition and is done once. The query compiler can automatically use the replicated table instead of sending data at run time through a table queue.

Example for creating a replicated dimension table:

```
CREATE TABLE Scm.DimTable__repl  AS (
   SELECT ...Some or all columns...
   FROM DimensionTable
)
DATA INITIALLY DEFERRED REFRESH DEFERRED
DISTRIBUTE BY REPLICATION IN ts_pd_mqt
COMPRESS YES
NOT LOGGED INITIALLY;
REFRESH TABLE Scm.DimTable__repl;
RUNSTATS ON TABLE Scm.DimTable__repl WITH DISTRIBUTION;
```

The replicated table does not need all columns of the base table. You can see the required columns under `Output Streams` in the details for the table queue operator in the explain plan.

Pattern: Directed table queue **(**DTQ) to partitioned table space on a table with 1,000,000 or more rows.

Action: Collocate tables or redistribute the queued table with an MQT.

Redistribution with MQTs is similar to replication. The main difference is that you define the MQT with DISTRIBUTE BY (...NEWHASHKEY...) instead of DISTRIBUTE BY REPLICATION. With redistribution, there is only one copy of the data instead of multiple copies, as there is with replication. In both cases, the DB2 agent in each database partition can read data from local copies in the MQT. There is no need to send the data again via the network.

A common way to collocate tables is to add a redundant join. The Item table in the following example is distributed on the ItemID column, but the join in the query uses the BasketID column. The tables are not collocated.

```
CREATE TABLE CUSTOMER ( CUSTID BIGINT PRIMARY KEY, ….)
DISTRIBUTED BY (CUSTID);
CREATE TABLE BASKET( BASKETID BIGINT PRIMARY KEY, CUSTID BIGINT,
… )
DISTRIBUTED BY (CUSTID);
CREATE TABLE ITEM( ITEMID BIGINT, BASKETID BIGINT, PRODUCTID INT,
AMOUNT DECIMAL(10,2) …)
DISTRIBUTED BY (ITEMID);

-- Sales amount per customer, not collocated
SELECT CUSTID, SUM(AMOUNT) FROM CUSTOMER C, BASKET B, ITEM I
WHERE B.CUSTID = C.CUSTID AND I.BASKETID=B.BASKETID;
```

Collocation can be introduced by adding a redundant column, CustID, to the Item table. A join on the CustID column is also added to the query.

```
-- Collocate tables on CustID
CREATE TABLE Item( ItemID bigint, BasketID bigint,  CustID
bigint, ProductID int, Amount decimal(10,2) …) DISTRIBUTED BY
(CustID);
-- Sales amount per customer, collocated
SELECT CustID, sum(Amount) FROM Customer C, Basket B, Item I
WHERE B.CustID = C.CustID AND I.BasketID=B.BasketID AND I.CustID
= B.CustID
```

Pattern: DTQ to coordinator partition, except for trivial processing of the final result.

Almost all query plans have a DTQ at the top that sends the result sets from the data partitions to the coordinator partition for final sorting or grouping. This process is fine if the amount of transferred data is small, which is true for most analytical queries.

Action: Investigate a DTQ if the amount of data is large or if the cost of the following operators, that is, operators higher in the graph, is substantial. Try to parallelize more sections of the query. For example, check whether you can distribute more tables.

Nested loop joins

Pattern: The cost of a nested loop join is much higher than the cost of its left outer input. This situation usually occurs when many rows come from the outer/left side of the join.

Hash joins or merge joins are often faster than nested loop joins even when there is a suitable regular index on the right/inner table.

Nested loop joins often occur together with index lookups on the inner/right side. The optimizer often chooses indexes with corresponding NLJOIN operator instead of using the HSJOIN operator.

Action: Try removing the index if it is not critical in other queries.

Pattern: In a nested loop join, the data types of the joined columns are different.

A hash join requires both columns in an equijoin to have the same data types. Fixed-length types with different lengths or precisions, such as CHAR(10) and CHAR(12), do not match, so the compiler does not use a hash join. Type casts at run time do not make the hash join eligible. The compiler chooses the NLJOIN or MSJOIN operator instead. Variable-length types such as VARCHAR(10) and VARCHAR(12) can be mixed in hash joins.

Action: If data types in joins are different, make columns use the same data type in the DDL statements of the corresponding tables.

Pattern: In a nested loop join, the join predicate uses an expression. There can be a TBSCAN operator on the inner/right side.

If a join predicate includes an expression, a hash join does not apply. The compiler chooses a nested loop join instead. Existing indexes might not be applicable if the indexed column is part of the expression. For example, the evaluation of the T2.Last = T2.First +1 predicate does not use an index on the T2.First column. However, the T2.Last -1 = T2.First predicate can be evaluated by searching through an index on the T2.First column.

Action: Take one of the following actions:

- Try to rewrite the join predicate, or enable a hash join by precomputing the expression in a generated column. For more information, see the "Table scan with many input rows and few output rows" pattern, earlier in this section.

- Define table partitioning, MDC, or a regular index on the inner table of the join.

Random lookups

Pattern: In the right inner branch of an NLJOIN operator, there is a pair of FETCH and IXSCAN operators where the FETCH operator does not use prefetching. In this situation, index lookups can cause random I/O on a table when pages are fetched in an arbitrary sequence.

Example:

```
            1
          FETCH
          (   3)
          15.2114
           2.01
        /---+----\
       1         1e+007
     IXSCAN   TABLE: TMP
     (   4)      BIGTABLE
     15.1356        Q1
       2
       |
     1e+007
 INDEX: GREGOR
        INDEX1
        Q1

…
…
3) FETCH : (Fetch)
               MAXPAGES: (Maximum pages for prefetch)
                       1
               PREFETCH: (Type of prefetch)
                       NONE
```

If such a combination of FETCH and IXSCAN operators is part of the inner branch of a nested loop join, every index lookup might cause a new physical I/O operation that reads just one page of the BIGTABLE table from disk.

Action: Make sure that the estimated number of rows on the IXSCAN operator is close to the actual number of rows. For example, use column group statistics on the columns that are used in the IXSCAN operator. Check whether you can drop the index or replace it with MDC. Alternatively, reorganize the table with the index.

```
REORG TABLE tmp.bigtable INDEX gregor.index1;
RUNSTATS ON TABLE tmp.bigtable WITH DISTRIBUTION AND SAMPLED
DETAILED INDEXES ALL;
```

After the table is reorganized, the optimizer might extend the index lookup by list prefetching, which avoids random I/O. List prefetching is implemented by a combination of SORT and RIDSCN operators. These operators sort the internal row IDs and then fetch pages from the table in physical order.

Example with list prefetching:

```
            10
           FETCH
           (   3)
           47.0834
           33.5324
          /---+----\
        10         1e+007
      RIDSCN   TABLE: TMP
      (   4)      BIGTABLE
      15.1409        Q1
         2
        |
        10
      SORT
      (   5)
      15.1406
         2
        |
        10
      IXSCAN
      (   6)
      15.1396
         2
        |
      1e+007
  INDEX: GREGOR
         INDEX1
        Q1

…
…
3) FETCH : (Fetch)

               MAX RIDS: (Maximum RIDs per list prefetch request)
                      512
               PREFETCH: (Type of Prefetch)
                      LIST
```

Merge joins

Pattern: An MSJOIN operator is on top of SORT operator in the graph.

Merge joins can perform well when both input branches are already ordered on the join column, for example, when rows are read from an index. If additional sorting is required, a hash join is almost always better than a merge join.

.**Action:** Check whether there is a data type mismatch preventing an HSJOIN operator from being used. For more information, see the pattern with the description "In a nested loop join, the data types of the joined columns are different," earlier in this section. If there is no data type mismatch, consider reorganizing the unsorted table with an index.

Hash joins

Pattern: There is an HSJOIN or MSJOIN operator with one small input table and another large input table. The join returns only a small percentage of the rows from the input tables.

For an HSJOIN operator, the small table is usually in the right branch and is therefore the build table. The large table usually in the left branch is called the probe table.

In the following example of a portion of a plan, only one row is selected from the DIM_WEEK table, and the join returns less than 2% of its input data:

```
                  123346
                  124764    ← Small subset of input data
                  ^HSJOIN
                  (  42)
                  501994
                    NA
                /-------+--------\
          8.0175e+06                1
          8.10293e+06               1
            TBSCAN                BTQ
           (  43)                (  44)
            501700                3.7432
              NA                    NA
              |                     |
          3.74816e+08               1
              NA                    1
       DP-TABLE: TMP             TBSCAN
           BIG_FACT              (  45)
              Q19                 3.71259
                                    NA
                                    |
                                    65
                                    NA
                               TABLE: TMP
                                   DIM_WEEK
```

The TBSCAN operator applies some filtering. It uses table partitioning. You can optimize the table scan further by refining partitioning or by adding MDC. Making table partitioning very fine grained might require thousands of partitions, and that is usually not practical. Organizing the table with MDC is a better approach. A regular index might be less optimal because the base table is large and there are many selected rows.

Action: Consider MDC or table partitioning on the large table. This might turn the join into an NLJOIN operator with the large table in the right inner branch.

Pattern: An index scan (which uses the IXSCAN operator) returns multiple rows. A FETCH operator is used on the result of the index scan.

This IXSCAN operator requires multiple physical I/Os if the rows matching an index key are not located together on the same page or extent. The compiler tries to alleviate this problem by sorting on the page IDs and might insert SORT and RIDSCN operators into the query plan. The sorting can avoid multiple physical reads on the same page, but the rows might still be spread over more pages than necessary.

Action: Consider reorganizing the index.

Index coverage

Pattern: There is a combination of IXSCAN and FETCH operators where the number of rows used by the IXSCAN operator is higher than the number used by the FETCH operator.

Example:

```
            10.35
            FETCH
           (   3)
           39.6932
           5.2372
          /---+----\
        100        1e+006
      IXSCAN   TABLE: TMP
      (   4)        MYTAB
      15.1779         Q1
         2
        |
      1e+006
INDEX: TMP
   MYINDEX
      Q1
```

This situation indicates that the index does not cover all search arguments. The FETCH operator must read every row corresponding to the IDs that are returned from the index scan. Many of these rows do not contribute to the final result because they do not match the additional filter condition in the FETCH operator.

Action: Consider covering more columns in the index.

The Design Advisor is good at recommending indexes that cover more search arguments. In some cases, you can eliminate the access to the table by using an index-only scan. The index must contain not only the search arguments but also other columns that are fetched from the table, for example, in an INCLUDE clause. Wider indexes with more columns

might require more resources when you update the table. You must balance the optimization of queries against the overhead of index maintenance.

Sorts

Sorting can be expensive. Many data warehouse queries require some sorting because they aggregate a large amount of data. There are cases, however, where you can eliminate a SORT operator from a query or reduce the number of rows to be sorted.

You should check the previously described patterns before trying to eliminate or reduce sorting. A SORT operator might disappear as a result of optimizing another pattern.

Pattern: There is an expensive SORT operator that is processing many rows or a SORT operator on the inner side of a nested loop join.

SORT operators that process many rows are often expensive, that is, the estimated cost is much higher than that on the operator's input branch in the graph. In some cases, the compiler detects that the input is ordered already and the cost of sorting the data is estimated to be small. These cases are less important for manual optimization. SORT operators in the inner branch of a nested loop join might have a low cost, as shown in the explain plan. The SORT operator is called for every row coming from the outer table, and the aggregated cost might still be very high.

Action: Consider creating MDC or a regular index on the sorted columns. Try to achieve index-only access by including all columns that the query needs. If index-only access is not practical, reorganize the table with the index.

Best practices

- **Recommendation**: Distribute large tables across data nodes. Replicate smaller tables that you use frequently.
- **Recommendation**: Try to collocate joined tables that process large amounts of data. Increasing the skew on the data distribution of the smaller joined table by up to 10% is acceptable.
- **Recommendation**: In a star schema or snowflake data model, choose one dimension key for distributing the fact table and the dimension table. Replicate the other dimension tables with MQTs.
- **Recommendation**: Keep the number of regular indexes small. Use table partitioning or MDC where applicable.
- **Recommendation**: Use compression for MQTs. Issue the **RUNSTATS** command on MQTs.
- **Recommendation**: Collect statistics on all tables and keep the statistics current.
- **Recommendation**: Use the **RUNSTATS** command to gather distribution and index statistics by issuing a command similar to the following example: `RUNSTATS ON TABLE SCM.TAB WITH DISTRIBUTION AND SAMPLED DETAILED INDEXES ALL.`
- **Recommendation**: Define foreign key constraints as NOT ENFORCED for all foreign key relationships if the application that writes the data can guarantee its consistency.
- **Recommendation**: Define functional dependency constraints for all functional relationships in a table if the application that writes the data can guarantee the relationships.
- **Recommendation**: When analyzing a query, generate the explain plan with section actuals. If there are major differences between the estimated and actual numbers of rows, for example, by a factor of 2 or more, the compiler might not choose the optimal plan. Try to make the statistics more precise so that the number of rows that the optimizer estimates is closer to the actual number.
- **Recommendation:** After the estimated cardinalities match the actual cardinalities, walk top-down through the graph, following the path with the highest cost. Operators with high cost compared to the costs of the inputs are

critical to be optimized.

Conclusion

Maximizing system utilization and minimizing processing times are key goals for all large data-processing applications, particularly for large data warehouse systems. Optimizing the workloads is a key requirement for achieving these goals. The IBM Smart Analytics System includes configurations to optimize many types of workloads. This paper provides valuable techniques for further optimizing workloads for specific large warehouse systems and environments. The techniques that are presented in this paper focus on generating efficient execution plans that are tailored to your data and workloads. The first set of techniques helps provide the DB2 compiler with rich statistical information so that it can select an efficient execution plan. You can use the second set of techniques to refine execution plans based on the data and queries in your data warehouse environments. By following the optimization techniques presented in this paper, you can quickly tune and optimize workloads.

Appendix A. Optim™ Query Tuner for DB2® for Linux®, UNIX®, and Windows®

Optim Query Tuner for DB2 for Linux, UNIX, and Windows is an IBM product that can help cut cost and improve performance by providing expert advice on writing high quality queries and improving database design. Its easy-to-use advisors can help developers to write more efficient SQL queries.

Although this paper does not describe using Optim Query Tuner to perform the tasks documented the recommendations, you can use Optim Query Tuner to perform some of the tasks described. Optim Query Tuner's graphical interface and advisors can help you achieve results quickly and efficiently.

Further reading

- DB2 Best Practices papers on a variety of topics can be found at the following web site: http://www.ibm.com/developerworks/db2/bestpractices/

- The Best Practices document *Tuning and Monitoring Database System Performance* (http://www.ibm.com/developerworks/data/bestpractices/systemperformance/) explains how to monitor and tune settings in the operating system and in DB2 instances. The advice applies to all types of database applications.

- The Best Practices document *Physical Database Design* (http://www.ibm.com/developerworks/data/bestpractices/databasedesign/) provides general recommendations regarding table partitioning and MDCs.

- The Best Practices document *DB2 Workload Management* (http://www.ibm.com/developerworks/data/bestpractices/workloadmanagement/) provides recommendations on the design and implementation of a DB2 workload management environment.

- Technical information about Optim Query Tuner for DB2 for Linux, UNIX, and Windows can be found at the following web site: http://www.ibm.com/software/data/optim/query-tuner-luw/

Contributors

Adriana Zubiri
DB2 Performance QA and Data Warehouse Benchmarks

Calisto Zuzarte
DB2 Query Optimization

Garrett Fitzsimons
IBM Data Warehouse Best Practices

Haider Rizvi
IBM Smart Analytics System Architect

Jacques Milman
Sales & Distribution, Data Warehouse Architecture

Juergen Goetz
DB2 Enablement and Porting Consultant

Katherine Kurtz
DB2 and IBM Smart Analytics System Information Development

Paul McInerney
DB2 Product Development, User-centered Design

Stephen Addison
Database and Data Warehousing Principal

Serge Boivin
DB2 Information Development

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 does 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 2011. 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.

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.