Creating and using materialized query tables (MQT) in IBM DB2 for iSeries

Version 1.0

Michael W. Cain
DB2 for i5/OS Center of Competency
IBM Business Strategy and Excellence
September 2004

Table of contents

Abstract

This white paper explains, at length, the process for creating and implementing multi-threaded query plans (MTQP) within the IBM DB2 for OS/390 and z/OS (DB2) V7 query technology. DB2's V7 query technology marks a dramatic paradigm shift in the way queries are processed. It offers new methods for improving high performance query processing.

Introduction

The very definition, even database performance depends on good design. And good design includes a solid understanding of the underlying operating system and database technology, as well as the proper application of specific strategies.

This is also true for IBM DB2 for OS/390, which provides a robust set of technologies that assist with query optimization and performance.

This paper introduces DB2 for OS/390 multi-threaded query tuning (MTQ) and hints at their design, creation and use.

It is strongly recommended that database administrators, analysts and developers who are new to the DB2 System z/OS platform, or need a brief, attend the DB2 for OS/390 and Query Performance Monitoring and Tuning workshop. This course teaches the developer the proper way to architect and implement a high performing DB2 for OS/390 solution. The DB2 Online Services Web site listed in Appendix B provides more information about this workshop.

This white paper assumes some knowledge of DB2 for OS/390. It is helpful to refer to, and familiarize yourself with, the information contained in the Bibliography (Listed Web site listed in Appendix B) as well as the following publications:

Recommended publications:
- DB2 for OS/390 SQL Reference
- DB2 for OS/390 Database Performance and Query Optimization

Bookmarks:
- Preparing for and Tuning the DB2 for OS/390 Query Engine on DB2 for OS/390 for Beginners (z/OS V1R6 ed.)
- DB2 Performance Diagnosis on DB2 for OS/390 for Starters
- (z/OS V1R6 ed.)

Papers:
- Indexing and Statistics Strategies for DB2 for OS/390
- Star Schema and Transport within DB2 for OS/390
- DB2 for OS/390 Sysplex Multiprocessing

Links to these as well as other mainframe publications and papers are available on the DB2 for OS/390 portal listed in the Resources section of this course.

MQT overview

Beginning with Microsoft SQL Server 2000, the SQL Server supports the creation and explicit use of MQT. In Microsoft SQL Server, we refer to them as indexed views.

An MQT is a DB2 table that contains the results of a query, along with the query's definition. An MQT can also be thought of as a materialized view or automatic summary table that is stored on an underlying table or set of tables. These underlying tables are referred to as the base tables. By running the appropriate aggregate query over time (by using the base tables) and then storing the results as MQT, they are accessible on subsequent requests. This provides a substantial data processing and query performance significance.

MQTs are a powerful way to improve response time for complex SQL queries, especially queries that involve some of the following:

- Aggregation on commercial data that occurs over a large subject areas
- Joined and aggregated data covering a set of tables
- Commonly accessed subset of rows that's, a specific slice of data

In many environments, users often issue queries repetitively against large volumes of data with slight variations in query predicates. For example:

- Query1 requests the revenue figures for retail grocery items sold in the central region each month (this year/last year)
- Query2 requests the revenue figures for retail grocery items sold in all regions for the month of December
- Query3 requests the revenue figures for a specific item sold in all regions during the past six months

The results of these types of queries are almost always expressed as summaries or aggregates for groups. They often require not only reading millions or billions of base table rows but also joining many tables. For each query, the raw detailed data needs to be processed. Query response times are likely to be slow, along with high resource utilization.

MQTs were introduced to assist the query optimizer and database engine and to alleviate these performance issues.

The functionality of an MQT is similar to the role of an index. Both objects provide a path to the data that the user is normally unaware of. Unlike an index, a user might directly query the MQT just like a table or view. However, adopting queries to use an MQT directly might not be a trivial exercise for the user.

Though MQTs can be directly specified by a user's query, that user cannot control how the query optimizer's ability to recognize the existence of an appropriate MQT implicitly, and to rewrite the user's query to use that MQT. The query accesses the MQT instead of accessing one or more of the specified tables. The decision can dramatically minimize the amount of data read and processed (see Figure 1).

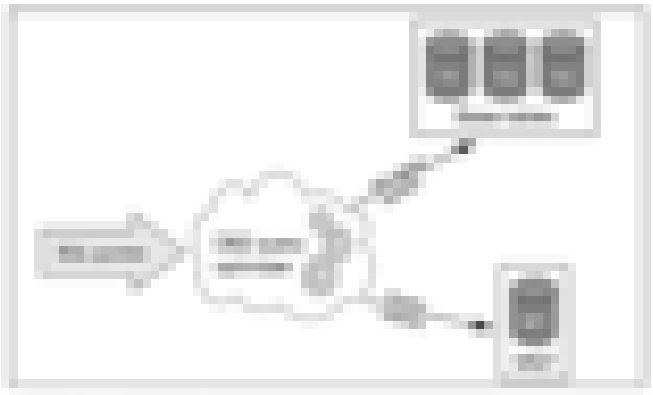


Figure 1. Use a query optimizer

During research testing in the DB2 for AIX® Center of Competency, a grouping query that was issued against a 1.2 GBytes row table took seven minutes to run without an MQT. With an MQT available, the same query took well under 0.1 seconds. Many type of query were several times a day. One creation of an MQT saves significant time and resources.

MQT implementation considerations

The implementation of MQTs does not completely eliminate query performance problems. Some challenges still include identifying the best MQT for the expected SQL, keeping MQTs when the base tables are updated, recognizing the existence of an MQT, and supporting the ability to actually use the MQT for the query.

DB2 for AIX does not automatically maintain the MQTs as the data changes in the base tables.

The decision to implement MQTs depends on answers to the following questions:

- Is it acceptable if the query calls (Record) results depending on whether the query uses the MQT or the base tables directly?
- What is the acceptable latency of data for the query?
- Are the performance benefits of implementing MQTs significant enough to offset the overhead of their creation and maintenance?

For online analytical processing (OLAP) and strategic reporting, there can be some cases, mostly for low volume instance of maintenance (latency) such as end of day, end of week or end of month batch periods. In such cases, the MQTs do not need to be kept synchronized with the base tables.

For online transaction processing (OLTP) and tactical reporting, any MQT latency might be unacceptable.

It is important to note that significant system resources and time can be required for creating and rebuilding the MQTs when the volume of change activity is high on the base tables and large. The overhead includes:

- Temporary space (often creating and populating the MQT) and associated indexes
- Permanent space to house the MQTs and associated indexes
- Processing resources when creating and maintaining the MQTs and associated indexes
- Time available to create and maintain the MQTs and associated indexes

The general steps to implementing MQT are:

1. Analyze the data model and queries
2. Design and launch the MQT definitions
3. Create and verify the MQTs
4. Populate the MQTs
5. Test and tune the MQTs
6. Design and launch the MQT refresh strategies
7. Test and tune the MQT refresh strategies

In DB2 UDB, for the optimizer to consider and use MQTs implicitly, the following DFTs must be included or implemented:

1. [illegible]
2. [illegible]
3. [illegible]
4. [illegible]
5. [illegible]
6. [illegible]
7. [illegible]

All the MQT support is provided in the DB2 UDB base code.

Installing the latest DB2 Group FP8 package is recommended before implementing MQTs.

1. [illegible]
2. [illegible]

Note: DB2 is the new name for DB2/MVS with the release of Version 8 Release 1.

Analyzing the data model and queries

Determining the proper MDX creation and usage strategy relies on analyzing and understanding the data model, the data itself, the queries and the implementation expectations. This analysis can be proactive or reactive in nature. In practice, both approaches are necessary. The data mining-based approach is generally performed before you have detailed knowledge about the data. The workload-based approach is performed after gaining experience with the queries.

MDX provides the most benefit when the queries are frequently aggregating or summarizing similar data from many members (see Figure 3).

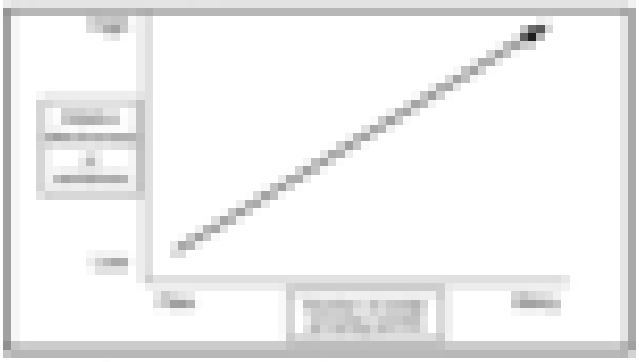


Figure 3. MDX usage with a number of similar grouping queries

MDX provides the most benefit when most queries are frequently aggregating or summarizing data that resides in only a few groups. In other words, as the ratio of base data cells to distinct groups approaches one-to-one (1:1), the effectiveness of the MDX declines (see Figure 4).

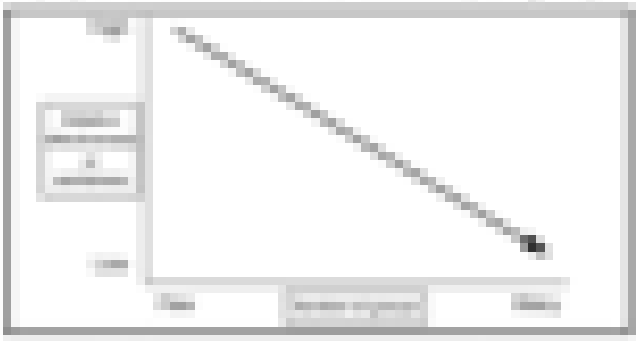


Figure 4. MDX effectiveness diminishes as the number of groups increases

Furthermore, if the MDX requests select very few members from the hierarchies, the data processing required to join and the query response time is fast, difficult the cost for precalculation offers little. For example, a query criteria of two members for Year = 2003 (Month = June) and Customer = John Doe. This MDX request causes the database engine to access and aggregate 100 rows out of billions of rows. There are many customers, that is, there are few rows per group represented in the data.

On the other hand, a different query selects all transactions for Store 4 (BBB) and Month 4 June. This MDX request causes the database engine to access and aggregate hundreds of thousands of rows (only a few Year and Month combinations (that is, many rows per group) are represented in the data).

The more often the MDX has to be refreshed, the more effective the MDX might be. The common retrieval latency between the base tables and the MDX. If the MDX requires refreshing less often and there is an adequate window of time to perform the refreshes, more MDX can be employed.

Figure 4. Frequency of change v more and MDX tables

Analyzing the application

Analyzing the data model and application reveals very requirements for grouping and aggregations, along with common and frequent join criteria. It is important to pay particular attention to detailed or explicit hierarchies represented in both the data model and the data itself. If there is a need for frequent aggregations along a hierarchy within a detailed table, MDX can provide help. For example, a detailed transactions table has a time hierarchy of Year | Quarter | Month | Week | Day, and these periods are specified as grouping criteria in a particular query. This is most MDX can be created to support this operation against the detailed data, especially if these reports run frequently.

Using the SQL Performance Monitor (database monitor), queries can be explored and analyzed. To help create an MDX that will benefit, the analysis has to be designed to determine frequently accessed tables where grouping or joining criteria are specified. The candidate SQL requests might be longer running queries that use a lot of processing or I/O resources. By sorting and grouping the queries based on the tables accessed, as well as the aggregated columns and the grouping criteria, an MDX can improve performance. It is also important to understand which query engine (Classic Query Engine (CQE) or DB2 Query Engine (DQE)) optimizes and executes query segment. Only the new DQE supports the optimization and uses MDX(s).

An XQUF use context allows any query definition, but the query definition supports only a limited set of functions when matching XQUF in a query. In general, the XQUF matching algorithm includes the following:

- Single-table queries
- [illegible] queries
- [illegible]
- [illegible] clause
- [illegible] and optional [illegible] clauses
- [illegible] clause
- [illegible]
- Views, restricted table expressions, and nested table expressions
- Indexes
- Partitioned tables

Note: XQUF XQUE supports only one XQUF per query. XQUF XQUE allows more than one XQUF per query.

The items with limited or no support for matching XQUF to queries include the following:

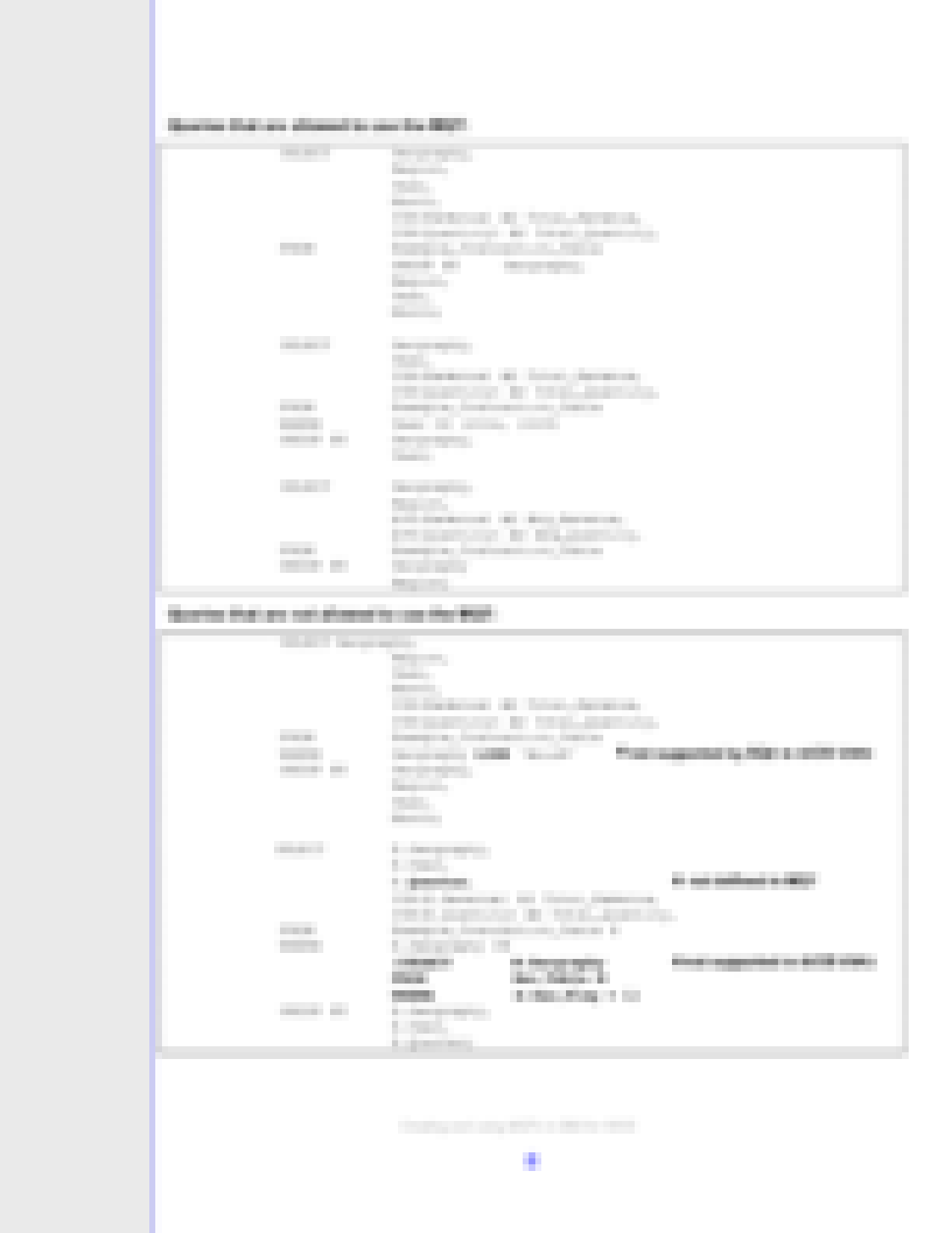
- Scalar subselects
- User-defined functions (UDFs) and user-defined table functions (UDTFs)
- Recursive common table expressions (RCTEs)
- Scalar functions (with [illegible] variations) in which one or more of the parameters is not a [illegible] (for example, [illegible] or [illegible]), or the [illegible] clause [illegible] [illegible]

Furthermore, the XQUE result options for query definition within the XQUF XQUE options for XQUF query. Even then XQUF is not explicitly used for queries. Additional information on XQUF query matching support is found in the publication *DB2 Universal Database for iSeries Database Performance and Query Optimization*. Appendix D provides a technical document in the CBU for DB2 Multisystem Information Center.

The lines following code snippets are examples of an XQUF, as well as the queries that can, and cannot, make use of it.

XQUF definitions

```
[illegible]
[illegible]
```



Background considerations for MDT's

Some applications and scenarios are very good candidates for MDT usage. Business intelligence (BI) and data warehousing (DW) environments lend themselves to the advantages of unconstrained data. BI and DW applications normally store and query vast quantities of data. BI and DW applications typically validate and process data using transactions such as time and business-related values. These transactions provide natural opportunities to create MDTs. Furthermore, BI and DW environments usually have clearly defined latency between the transaction data and the warehouse data. For example, adding daily transactions to the data warehouse defines a natural and consistent latency in the BI system on a periodic basis. Reviewing the transaction and the query requests within the BI environment yields a set of MDTs that can provide transactions benefit and can also be maintained as part of the DW indexes transaction load (ETL) processes.

Star schema or snowflake schema data models are specific cases where MDTs can be employed. Traditionally, the fact table contains transactions or measures that can often be as wide, contain and records. The dimension tables contain the descriptive information and the information frequently defines a hierarchy. For example, the time dimension contains a time hierarchy (year, month, day). The product dimension contains a product hierarchy (category, product) and the location dimension contains a location hierarchy (country, region, city/town). MDTs can be practically defined to provide precomputed roles along the most commonly used levels.

In many cases, a precomputation process already exists and is in use. In these situations, the existing process can be left as is or it can be modified to include the use of MDTs and the optimizer's query rewrite capability. By altering the existing summary tables to be MDTs and modifying the queries to access the base tables, the optimizer can be relied upon to make the decision whether or not to use the base tables or the MDTs. The decision is based on the estimated runtime cost for each valid query execution. Using the query optimizer to make this decision allows more flexibility. On the other hand, it is not limitless and not all queries be the best strategy for the existing precomputation process.

Designing MDT definitions

With any modified transaction or querying efforts, trying out the MDTs can be straightforward. Using the previous example of a time hierarchy, assume 100 million rows in a detailed transaction table that represents three years of evenly distributed data, including the following distinct levels of groups:

- 3 years
- 100 quarters (3 years x 4 quarters)
- 300 months (3 years x 100 months)
- 1095 weeks (3 years x 365 weeks)
- 1095 days (3 years x 365 days)

Executing a query that groups all 100 million rows by the most detailed level (day) results in only 1095 distinct groups. Yet, additionally processing all 100 million rows is time consuming and resource intensive. This is where designing an MDT is valuable.

By providing an MDT that represents all the data grouped by Year, Quarter, Month, Week, Day, the query optimizer can rewrite the query to use the MDT instead of the base transaction table. Rather than

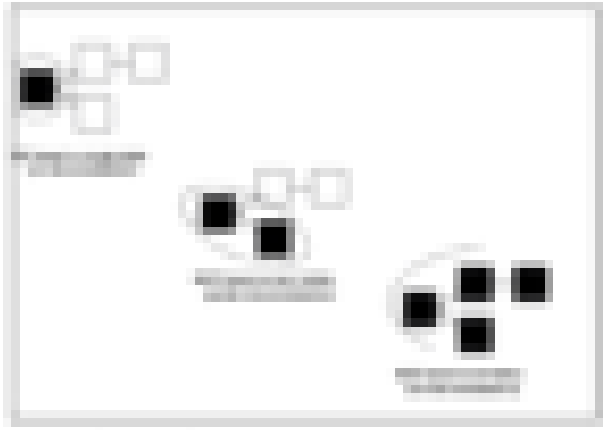


Figure 4-1. Creating MDX cubes based on single or multiple cubes

[illegible] can be created with local collection against one or more tables. In this case the [illegible] is considered for the operation only when the data represented by the specified local collection. Because the [illegible] only considers some of the data from the base tables, its overall usefulness might be decreased. Furthermore, any literals in the [illegible] and in the user's query are replaced with parameter markers during optimization. This makes matching the user's query to the [illegible] impossible in [illegible]. Because of this behavior, it is recommended that you do not use literal values anywhere in the [illegible] query.

For example, the following query will not use the index to optimize any literals used. It

```
[illegible]                                                    [illegible]
[illegible]                                                    [illegible]
                                                               [illegible]
                                                               [illegible]
[illegible]                                                    [illegible]
[illegible]                                                    [illegible]
[illegible]                                                    [illegible]
[illegible]                                                    [illegible]
```

An [illegible] additional support provides better matching of queries [illegible]. The [illegible] optimization process matches the associated [illegible] when creating the [illegible] to the associated [illegible]. The local collection predicate of the user query. The optimizer verifies that the values of the parameter markers are the same. Therefore, the [illegible] can be used. The [illegible] matching algorithm also attempts to match where the predicates in the [illegible] and the query are not exactly the same. For example, if the [illegible] has a predicate [illegible] and the query has the predicate [illegible], then the [illegible] contains the rows necessary to run the query. The [illegible] is used in the query. The predicate [illegible] is left as a local collection in the query. Therefore, values [illegible] must be defined in the [illegible].

Designing [illegible] that are based on queries

When designing an [illegible], it is useful to account for all the columns projected in the set of queries. Either the [illegible] must include all the columns used in the queries or must include the join columns to facilitate gathering the columns through a join to the associated tables.

For example, the following grouping query selects rows from a single table that match [illegible] and [illegible], summarizing two columns (Sales and Quantity) and grouping by [illegible]. The [illegible] designed to avoid this query must contain the projected columns [illegible], [illegible], and [illegible]. If one or more columns are omitted, the [illegible] is not usable.

```
[illegible]                 [illegible]
                            [illegible]
                            [illegible]
                            [illegible]
                            [illegible]
[illegible]                 [illegible]
[illegible]                 [illegible]
[illegible]                 [illegible]
[illegible] [illegible]     [illegible]
                            [illegible]
                            [illegible]
                            [illegible]
```

As another example that specifies a join and groups between two tables, either of the MDT designs can be used to execute the query:

- the MDT that is based only on **Drama_Table**
- the MDT that is based on both **Drama_Table** and **Role_Table**

```
SELECT                  D.*,
                        R.*,
                        M.*,
                        C.*,
                        COUNT(*) OVER() AS Total_Count,
                        COUNT(*) OVER(PARTITION BY Role_Actor) AS Role_Count,
FROM                    Drama_Table AS D
                        INNER JOIN
                        Role_Table AS R
                        ON D.Drama_Name = R.Role_Drama
                        INNER JOIN
                        Movie_Table AS M
                        ON D.Drama_Name = M.Movie_Drama
                        INNER JOIN
                        Cast_Table AS C
                        ON D.Drama_Name = C.Cast_Drama
WHERE                   D.Drama_Name = 'Hamlet'
ORDER BY                D.Drama_Name,
                        R.Role_Actor,
                        M.Movie_Year;
```

The MDT that is based only on **Drama_Table** must contain the DataKey, CHARCluster and CHARDataKey collections. The query optimizer can specify a join between the MDT and **Role_Table**, regrouping the rows based on **DataKey**.

The MDT that is based on both **Drama_Table** and **Role_Table** must project the Total_Count and Role_Count from **Role_Table**, and the CHARCluster and CHARDataKey collections from **Drama_Table**. The query optimizer can omit the join entirely because the MDT contains all the data required to complete the query.

As another example that specifies a join between three tables, either of the MDT designs can be employed to execute the query:

- the MDT that is based on all three tables
- the MDT that is based on two tables

```
SELECT                  D.*,
                        R.*,
                        M.*,
                        C.*,
                        COUNT(*) OVER() AS Total_Count,
                        COUNT(*) OVER(PARTITION BY Role_Actor) AS Role_Count,
FROM                    Drama_Table AS D
                        INNER JOIN
                        Role_Table AS R
                        ON D.Drama_Name = R.Role_Drama
                        INNER JOIN
                        Movie_Table AS M
                        ON D.Drama_Name = M.Movie_Drama
                        INNER JOIN
                        Cast_Table AS C
                        ON D.Drama_Name = C.Cast_Drama
WHERE                   D.Drama_Name = 'Hamlet'
ORDER BY                D.Drama_Name,
                        R.Role_Actor,
                        M.Movie_Year;
```

The MDT that is based on all three **Drama_Table**, **Role_Table** and **Cast_Table** tables must project the Total_Count and Role_Count from **Role_Table**, CHARCluster from **Cast_Table** and DataKey and CharacterKey from **Drama_Table**. Note that the DataKey and CharKey join columns are specified in the MDT's query definition such as the **WHERE** clauses, but the columns are not part of the MDT data. The query optimizer can omit the join entirely because the MDT contains all the data required to complete the query.

An MDT that is based on the Promo_Table and Date_Table tables must project the CustKey, Sales and Quantity columns from Promo_Table and the Year, Month and Day columns from Date_Table. The CustKey column facilitates joining the MDT to Customer_Table.

An MDT that is based on the Promo_Table and Customer_Table tables must project the CustKey, Sales and Quantity columns from Promo_Table and the Customer column from Customer_Table. The CustKey column facilitates joining the MDT to Date_Table.

An MDB stores the query optimizer can exploit the join between a cell of buffer because the MDT contains all the data required to that join at index. The inclusion of the other some is join columns allows the MDT to be joined to a index and implemented in the MDT.

The two-table design which includes a third table's join columns allows additional queries to use the respective MDT. Keep a view that the number of view groups in the MDT might not longer based on the number of distinct join columns values (such as the join column specified in the GROUP BY clause).

Designing MDTs that are based on data models

MDTs designed for a star schema or snowflake schema data model can be based either on a single fact table or on the fact table plus one or more dimension tables. If only the fact table is considered, then foreign keys for one or more dimension tables must be included in the MDT. This allows for selection on the dimension table, joining from the dimension table to MDT, and regrouping of the MDT data.

The granularity of a foreign key column represents the most detailed level within a given dimension. For example, each additional foreign key that is defined in the MDT as a grouping column results in more groups and more differentness. For example, if the TimeKey column represents 1095 distinct days or groups, adding the MonthKey grouping column can multiply the number of distinct groups by the number of distinct MonthKey values. If 100 stores have a transaction each day, this will result in the representation of 109,500 distinct groups in the MDT (1095 days x 100 stores) (see Figure 6).

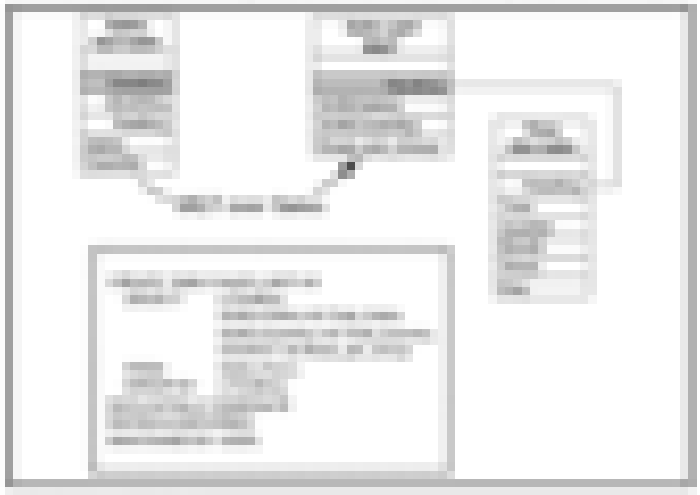


Figure 6 MDT over Tables

If the text table used one or more characters before and surrounded the results in an MDX that demonstrates the total model. By including all the combined columns in the MDX, the database engine can avoid running the entire algorithm (see Figure 7).

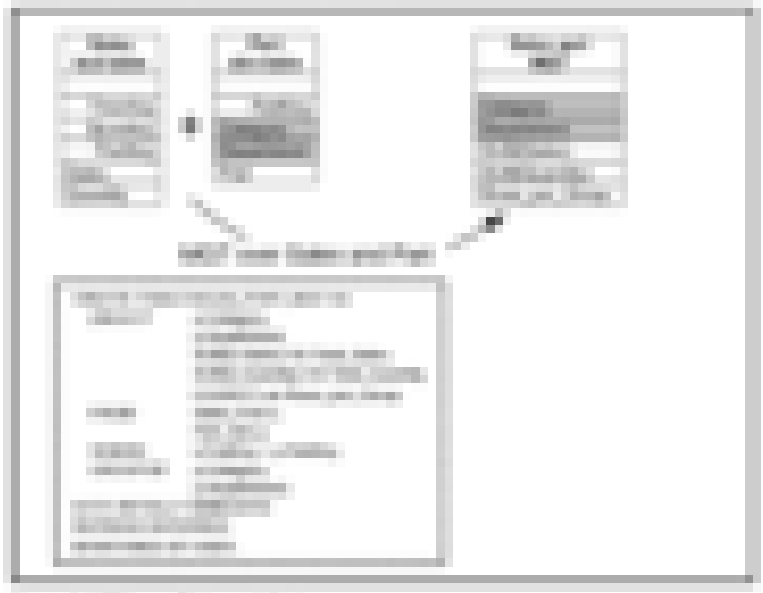


Figure 7 MDX over Data and Page

Designing MDX's Layout and Interactions

When creating an MDX collecting the appropriate grouping columns can make the difference between an effective or useless MDX. In a data model with roughly a eight-dimensions defined, the grouping columns specified can affect the MDX to return grouping counts at that level or higher in the hierarchy. This can be a very useful way to minimize the number of MDX's that must be created and maintained. Enough drill grouping significant benefits from your aggregation of data.

In the following example, an MDX that is created with the Year / Quarter / Month / Country / State_Province / County / City / Category / Department grouping columns can be used for that level or the any level above (such as Year / Quarter / Country / Category). A query that specifies grouping criteria below the level delivered by the MDX is not eligible to use the MDX (such as Year / Week / Store / Part) (see Figure 8).

Figure 4-4 MDX coverage

Another case to watch out MDX can help regards repeated requests for a distinct list of values where this set of values is static or changing slowly over time. The following query scans the entire table and returns a distinct set of location values:

```
[illegible]
```

If the table contains billions of rows, the scan and the distinct processing can take a lot of time and resources. Given that new distinct location values are not added frequently, this is a great opportunity to put MDX to creating an MDX that contains the distinct list of location values. The query optimizer can use the MDX to satisfy the query with little time and resources:

```
[illegible]
```

Instead of reading and processing billions of rows in the transaction table, now a hundreds of rows are read and returned from the MDX. Furthermore, the MDX only needs to be refreshed when a new distinct location is added or removed.

Although MDXs can be partitioned, the SUM(), MIN() query optimizer does not explicitly use the MDX. Changing partitioned MDX's is not recommended as an MDX MIN() and maximum or MDX SUM(). The optimizer can monitor the query when the base table or the MDX is partitioned.

It is important to keep in mind that implementing MDX's is not free. Besides the time and resources to create it maintenance. We additionally have to update query increases as the additional overhead. Costs and more MDX's. It is best to design a few MDX's that provide the widest coverage over the largest benefit.

As general overview creating DDTs for the following query classes:

- Queries with grouping and aggregation, where the cells of most groups is high
- Queries with limited values, where the cells of many limited values is high
- Queries with joins, where the number of results is high and has sub-totals

Creating DDT definitions

An DDT definition consists of a query and the attributes that are associated with the population, maintenance and use of the table. There are two methods for creating an DDT: **CREATE TABLE** or **ALTER TABLE**. Either method can be selected through our GUI, request a by using the graphical user interface in DDT Section Navigator.

The **CREATE TABLE** method allows for the creation and population of new tables. The **ALTER TABLE** method allows pre-existing tables to be modified into an DDT.

CREATE TABLE example

Given an existing cube **Example_Transactions_Table** with the appropriate column definitions:

```
CREATE TABLE Example_Transactions_Table AS
    SELECT
        Customer_ID,
        Store_ID,
        Product_ID,
        Date_ID,
        SUM(Sales_Amount) AS Total_Sales,
        SUM(Quantity) AS Total_Quantity,
        COUNT(*) AS Transaction_Count
    FROM
        Transactions
    GROUP BY
        Customer_ID,
        Store_ID,
        Product_ID,
        Date_ID
WITH DATA
PRIMARY INDEX (Customer_ID, Store_ID, Product_ID, Date_ID);
```

To create a new MDX by using Model Navigator - Customize (Figure 4):

1. Navigate to and open the **Schema**.
2. Right-click **Tables**.
3. Select **New > Materialized Query Table**.

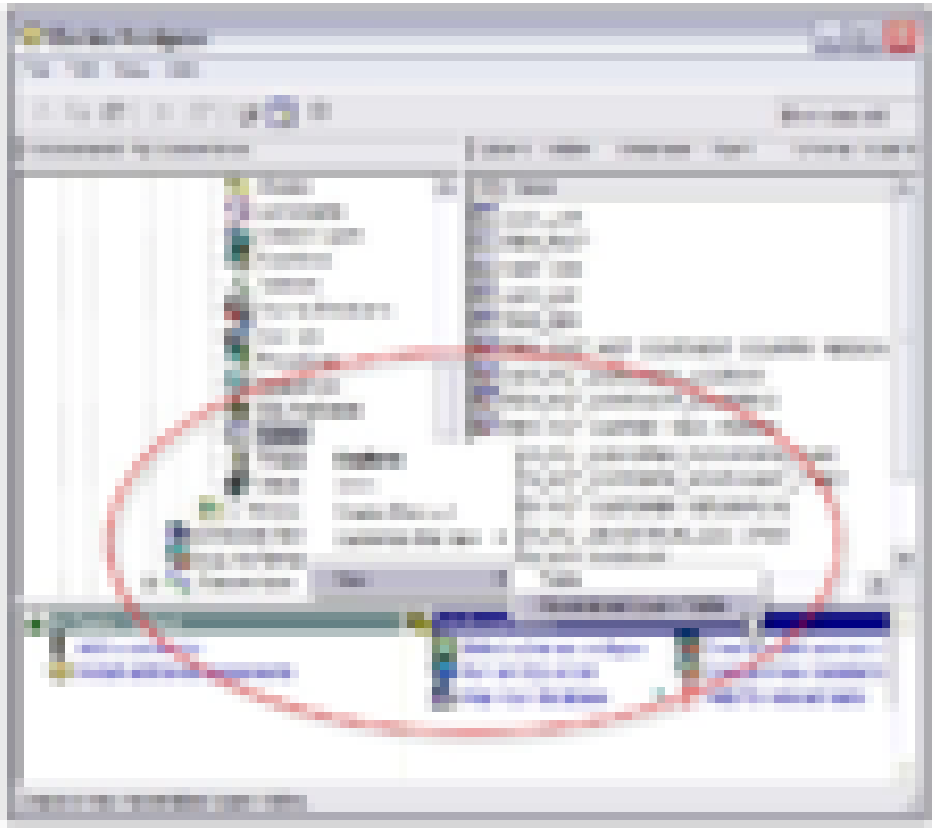


Figure 4. Create a new MDX using Model Navigator - Customize

The dialog to use the dialogs to specify the MDX definition and definitions.

ALTER TABLE example

Given an existing **Employee_Maintenance_Table** table with the appropriate columns definitions

```
[illegible]
```

To alter an existing table to become a new MDT using Oracle Navigator - Database:

1. Navigate to and open the **Schema**.
2. Open **Tables** and right-click a specific table.
3. Select **Definition** and select the **Materialized Query Table** tab (see Figure 10).

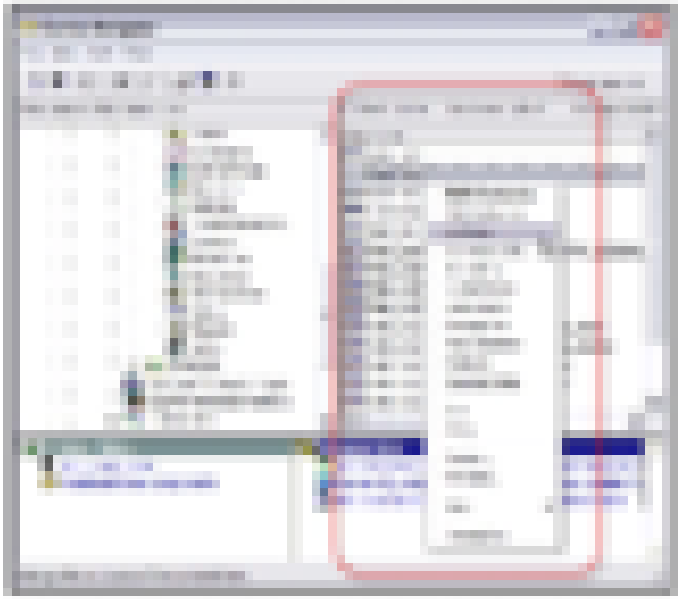


Figure 10 - Alter existing table to become a new MDT using Oracle Navigator - Database

This brings up the dialog to register the table as an MDT and specify the MDT attributes as in definition

[illegible]

[illegible] is not you are [illegible] with REST. It is important to understand the anatomy of the XML file's elements used to create and REST in ORE for ArcGIS. Different classes control the properties and [illegible] of REST's, and some functionality is not yet supported (see Figure 11).

Figure 11 Anatomy of an REST

Unlike the [illegible] and [illegible] classes, ORE does not automatically keep the REST's synchronized with the layers within. When the base tables change, there can be a difference between the contents of the REST's and the base tables. This difference represents the data latency.

The use of specific REST mapping conventions is helpful for quick identification during analysis and administration. Consider placing [illegible] conventions on the table names.

When laying out the materialized query definition, it is a good practice to specify all the columns that are considered additive due to expressions. As a reminder, these columns could be used with the [illegible] functions. As appropriate, other functions such as [illegible] and [illegible] can be specified. Providing these columns and functions might allow the REST to be more widely considered and used.

The number of rows per query provides the query optimizer with additional ways to take advantage of the REST. For example, determining averages for many measurements. It is always a good practice to consider the function [illegible] in the [illegible] clause of the REST definition. If expecting to calculate the average of sum over columns, the aggregate [illegible] data statistics in the query. [illegible] data statistics must be provided in the REST, not just [illegible].

When an INSERT is defined, the following **select** statement restrictions apply:

- The **select** statement cannot contain a reference to another INSERT or to a view that refers to an INSERT.
- The **select** statement cannot contain a reference to a declared global temporary table, a table in **SESSION**, a program described by an external file, a program that is not **PROGRAM** status.
- The **select** statement cannot contain a reference to a view that contains an invalid data for an INSERT.
- The **select** statement cannot contain an expression with a **DATALINK** or a distinct type that is based on a **DATALINK** where the **DATALINK** is **FILE LINK CONTROL**.
- The **select** statement cannot contain a result column that is not an XML data type, such as **XMLPARSE** with parameters **DOCUMENT** or **CONTENT**.

When an INSERT is defined with the **INHERIT SPECIAL REGISTERS** attribute, the following additional **select** statement restrictions apply:

- A **CURRENT DATE** value is allowed, but is only used by **REFRESH TABLE**; it might improve the locality of the data references in the INSERT.

Additional information on INSERT creation support can be found in the publication *DB2 Universal Database for iSeries SQL Reference*. Appendix B provides a link to the DB2 for i5/OS Publications Information Center where you can find this manual.

When creating INSERTs, the actual calculations and population of data will occur as part of the object creation request or anytime after creation. If the **DEFINE PERSISTENT** attribute is specified, the INSERT population is initiated as part of the creation phase. If the **DEFINE NOT PERSISTENT** attribute is specified, the INSERT population is not done. If the data is initially populated, the calculations and population can be initiated by using the **REFRESH TABLE** statement, or the process can be initiated and maintained by the programmer. By defaulting, the user can determine the best time and mechanism for calculating and populating the INSERT.

When defining tables to the INSERT, the original table can be empty or fully populated. When defining an existing table that contains data, it is the user's responsibility for the integrity and accuracy of the data.

Prior to populating the INSERT, it is advantageous to clarify whether the query optimizer will consider using the INSERT to redirect a user's query. A simple method for doing this verification is to examine the query plan for a few queries. The Display Description - Visual Explain tool can be used to explain the query. If the query optimizer replaces the base tables with the INSERT, the INSERT is shown in place of one or more base tables in the query plan. Refer to "Visual Explain" on page 19 for details of such a query plan. Consult also the "Tracking and Tuning Materialized Query Tables" section of this paper. If the query optimizer rejects the INSERT, further analysis and redesign can be done prior to INSERT population.

Populating MQTs

Calculation and population of the MQT data is a time- and resource-intensive exercise because of the fact that creation of MQTs normally requires accessing all the data in the base tables and aggregating values over very potentially many groups.

Creating an MQT might result in reading and processing millions or billions of rows.

Whether the aggregation is under the control of the database engine or the programmer, the query that is used to populate the MQT must be tuned.

Indexes to support MQT creation

Prior to the creation of MQTs, providing the proper indexes on the base tables is a critical success factor. The following guidelines need to be employed when creating the MQT's query:

- Create table and composite index indexes for any local selection columns.
- Create table indexes for all join columns.
- Create table indexes for all grouping columns.

Try this **CREATE MQT** example:

```
CREATE TABLE Summary_MQT AS
(SELECT
                                  Col_Grouping,
                                  Region,
                                  Country,
                                  Address,
                                  SUM(Col_Numeric_Values),
                                  COUNT(Col_Numeric_Values),
                                  COUNT(*) AS Total_Rows
                          FROM
                                  Base_Table_Name
                          GROUP BY
                                  Col_Grouping,
                                  Region,
                                  Country,
                                  Address)
DATA INITIALLY DEFERRED
REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY SYSTEM;
```

Build the following index:

```
CREATE INDEX Summary_MQT_Index ON Summary_MQT
        (Col_Grouping, Region, Country, Address);
```

This provides the query optimizer and database engine with statistics on the grouping columns and provides an index for implementation. Finally!

Use the CREATE INDEX command

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ]
    ON [ ONLY ] table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass [ ( opclass_parameter = value [, ... ] ) ] ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ INCLUDE ( column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

Build the following indexes

```
CREATE INDEX Sales_Year_Month_Idx
    ON Sales_Fact
        (Month_Key,
         Year_Key);

CREATE INDEX Sales_Prod_Idx
    ON Sales_Fact
        (Product_Key);

CREATE INDEX Sales_Cust_Idx
    ON Sales_Fact
        (Customer_Key);
```

This provides the query optimizer with database access with statistics on the primary and grouping columns and provides indexes for implementation. Provided

Environment to support MQT creation

Providing a proper working environment is also important when aggregating large data sets so that we provided. It is interesting the time to create and populate the MQT is important, the aggregation process needs to be run in an environment that provides as much processor power as possible and a large memory pool. The MQT population process must be allowed to run as isolated process. A possible configuration would be to **REPLACE** the Sales_Fact aggregation the data and populates the MQT allows the query optimizer to consider all the statistics in the MQT's and Dimensions when the database engine can direct the appropriate data to the MQT's. The **REPLACE** parallel degree also allows the query optimizer to consider running the MQT query with symmetric multiprocessing (SMP), provided that the DB2 INSTANCE is configured as the system and logical partition (LPAR). To allow the query to complete success. MQT tables resources the time the example, more resources are used in a given environment

Cascading MQT creation

When creating multiple MQT's the original hierarchy (such as Year, Month, Day), the hierarchies might be additionally queried. A better approach is to take advantage of the previously aggregated data, using it as the basis for the next MQT in the hierarchy. This minimizes the time and resources required to aggregate the next level of data in the hierarchy. The MQT that represents the lowest level in most detailed data is created first (Year, Month, Day), followed by the next level (Year, Month), followed by the next level (Year) (see Figure 10).

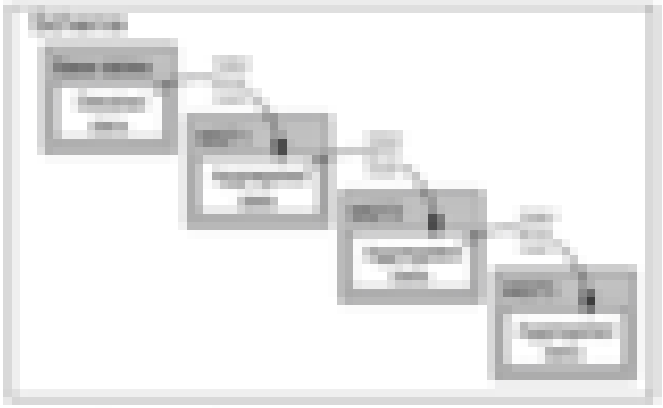


Figure 4. Cascading MDT creation

Since that MDT definitions cannot be based on other MDTs, the process of cascading the MDT creations involves some programmer intervention. That is to say, the MDT definitions must reference the base tables that the user wants to populate the MDT, or from a previously created MDT.

Figure 5 shows the general steps to create MDTs in a cascading fashion:

1. Create the initial MDT in the hierarchy and populate it from the base tables.
2. Create the next MDT in the hierarchy with the [redacted] attribute and populate this MDT from MDT created in step number 1.
3. Create the next MDT in the hierarchy with the [redacted] attribute and populate this MDT from the MDT created in number 2.

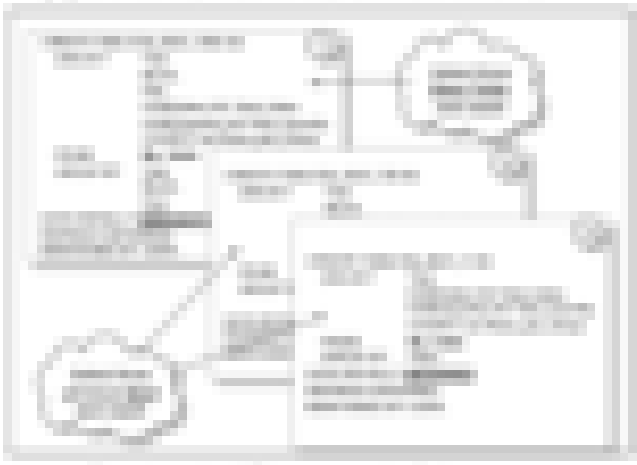


Figure 5. Steps in creating MDTs in a cascading fashion

For this process to be successful, all the MDT definitions in the hierarchy must reference another level in the hierarchy, except the last one, which references base tables.

Strategies and methods for aggregation

The query optimizer has two basic methods of grouping data for aggregation:

- Grouping with an index (pipelined or transient)
- Grouping with a hash table

Each method has its own requirements, characteristics and advantages. Understanding and anticipating the cost of either method beforehand, whether programmer intervention is required to speed up the SELECT processes.

The optimal use of either grouping strategy requires the optimizer to have a good understanding of the estimated selectivity of the query (normally WHERE and more importantly, a good understanding of the estimated number of groups and the average number of rows per group). This information comes from indexes and column statistics.

For hash grouping to be an optimal strategy, the optimizer and database engine need enough memory to the query plan to create the hash table. A large number of distinct groups results in a larger hash table, and a larger hash table requires a longer memory pool to perform efficiently. If the optimizer suspects the pool has not enough memory to be smaller than the estimated hash table size, the hash grouping strategy is avoided. When grouping with a hash table, the ability to read and process data in parallel with SMP is available. This feature allows grouping queries to perform faster by using more processors.

Index grouping is the preferred strategy when hash grouping is not viable. The memory footprint of using an index can be much smaller than creating an entire hash table. For index grouping to be optimal, a permanent index that covers the grouping columns is required. Without a permanent index available, the optimizer has to create a temporary data structure known as an internal list. This adds more time to the query execution.

When grouping with an index, one cannot wait and group the data in parallel through SMP. In other words, if an index is used for grouping, SMP does not help the aggregation run faster. The creation of a temporarily ordered list can employ SMP.

If index grouping is employed, and the SELECT calculation and population is not meeting response time expectations, programmer intervention is required. This might take the form of adding a specific populated condition that takes advantage of various forms of parallelism.

Parallel execution is not available when the database engine is writing the aggregated data to the SELECT through a single SQL request. For example, when populating the groups from the hash table, the data is inserted serially. If the SELECT is to be populated with many rows (such as groups), then designing a parallel SELECT calculation and population process is advantageous. It is a good practice to understand the optimizer's strategy for aggregation prior to running the SELECT creation query in the production environment. This can be accomplished by using the query optimizer's SetReject and Explain Navigation Viewer Engines.

The SELECT creation and population query can be explained only by using the Explain Navigation Viewer Engine utility. This allows the query optimizer's feedback to be analyzed without actually running the query.

Figure 11.3 shows an example of grouping with a permanent index as known through Visual Explain.

Figure 11.3 Grouping with a permanent index

Figure 11.4 shows an example of grouping and aggregation through a temporary hash table with SMP parallelism, as known with Visual Explain.

Figure 11.4 Example of grouping and aggregation with a temporary hash table with SMP parallelism

When the grouping columns are from more than one table, the selection and joining of rows from the base tables occurs before any grouping and aggregation. A single permanent index does not cover all the grouping columns. In this case, either a temporary hash table or a temporary indexed table is used to facilitate the grouping and aggregation of data.

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Figure 7.8 shows an example of grouping with a permanent index on Event with Visual Explain.

Figure 7.8. Example of grouping with a permanent index

Programmer intervention: Be it yourself

In cases where the MQT creation query uses index grouping, or the result can be in grouped form or be derived into the MQT, a large scale programmer intervention might be helpful in speeding up the calculation and population process. In cases where the MQT creation is based on joining tables that result in a large fact table of many scale programmer intervention might be helpful in speeding up the join process (see Figure 7.9).

Without programmer intervention some MQT creation queries might run for many hours, or in extreme cases, many days.

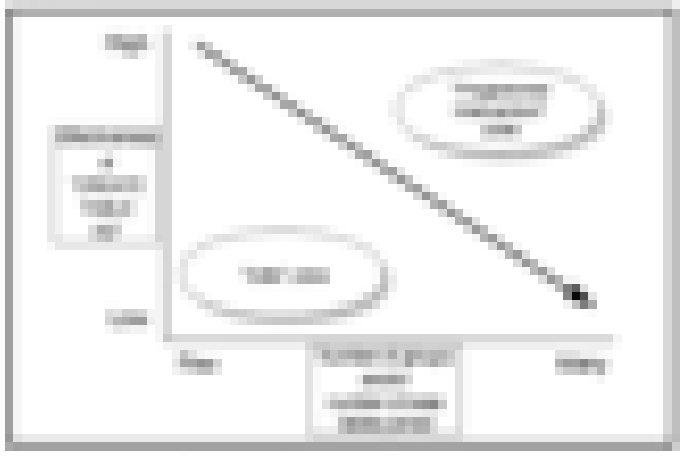


Figure 7.9. Number of groups and/or number of tables joined

The key to better joining, aggregation and insertion is parallelism. When the database engine is unable to employ WAP explicitly, the programmer can design and implement a parallel process. The process consists of breaking the data down into base tables and profiling it into logical ranges, selecting and processing each range in parallel, and inserting the aggregated data into the WAP in parallel (see Figure 24).

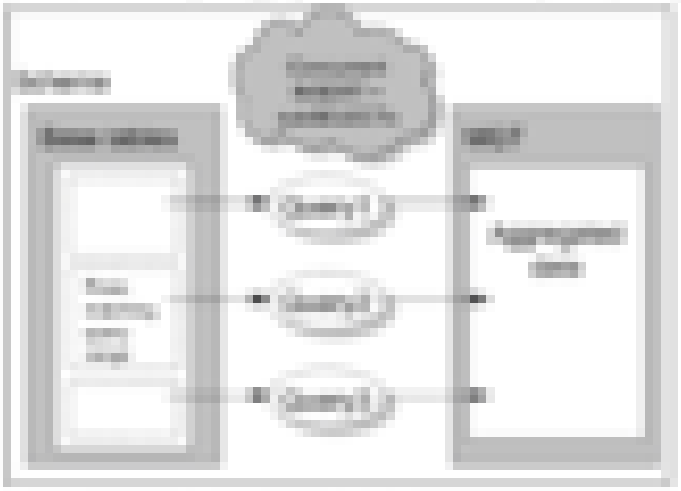


Figure 24: Concurrent detailed aggregation into a WAP

The design process starts with profiling the data represented by the first one or two grouping columns. Usually the distinct ranges of data within the first one or two grouping columns are compared with the number of processors available during query execution. When the number of processors is the number of ranges determines the level of parallelism is simple. The goal is to have all the processors as busy as possible, thus maximizing throughput and minimizing the time to populate the WAP. Using all the processing resources is possible as WAP assumes all the resources are available for this activity. If other jobs are running on the system, the degree of parallelism needs to be reduced.

For example, given a customer transaction table with grouping columns of Year, Customer, Home State, and Phone Area, and Thousands of customers represented in the data. What degrees of parallelism can be used. The parallel grouping queries can each select and process one of these years. On a system with three or more available processors, each grouping query runs on one processor. A parallel 3 could experience and available and a larger degree is desirable. More additional grouping queries can be used, each processing a given year and responsible ranges of customers.

When running on set of months all the same time, the ORACLE DOP degree could be set to **NONE**. Furthermore assume that all the base ranges represented in the base tables are concentrated on. It is easy to end in ranges resulting in incomplete data.

Employing a parallel process

Here is an example of creating and populating an RDD with a parallel process. First, you create the RDD with no data using the **[API Reference]** process.

```
[illegible]
```

Depending on the processing resources available, Hadoop will run a set of parallel cycles. Each cycle reads in values a distinct range of rows, aggregates the data and inserts the results into the RDD.

```
[illegible]
```

Given that the individual queries select subsets of data from the base table, the work to provide proper indexes over the local selection columns. This provides the query optimizer and database engine greater flexibility. Providing both wide and so-called narrow indexes can be advantageous. Using the previous example, the proper indexes to provide are as follows:

```
CREATE [ NONCLUSTERED ] INDEX ...
    ON <database_name>.<schema_name>.<table_name> (<column_name>)

CREATE [ NONCLUSTERED ] INDEX ...
    ON <database_name>.<schema_name>.<table_name> (<column_name>)
```

Be sure to test and verify the queries, the parallel process and the query results before relying on the MDT. The integrity and accuracy of the MDT data is the responsibility of the programmer.

Testing and tuning MDTs

If the query optimizer identifies the best query to access an MDT instead of the base tables, the query tuning methods and strategies are employed to access and process the MDT. Specifically, the MDTs are scanned and probed for local selection, joining, grouping and ordering. It is important to test and tune the MDTs prior to relying on them in a production environment. The key to good query performance is a proper indexing and statistics strategy.

The indexing and statistics strategy for MDTs is essentially the same as the base tables. That is to say, add indexes on the MDTs like query optimizer has the necessary statistics and has many choices when implementing the query request.

Column statistics used by SQL are collected and stored on a table by table basis, and like indexes, MDTs don't automatically collect a column statistic for an MDT. It is a good practice to identify those columns potentially used to ensure that the statistics are updated after populating or refreshing the MDT. This minimizes poor query performance immediately after the MDTs are repopulated.

Given that the MDT rows can be selected, joined, grouped and ordered, indexes need to be created to cover these activities. To determine the proper set of indexes, analyze the SQL code invoked and look the queries with the MDTs in place. Be sure to create indexes on any local selection columns and join columns. In addition, consider creating indexes on any grouping columns and ordering columns especially if the MDT has many rows.

Appendix B provides a link to the IBM Virtual Innovation Center where you can find more information on indexing and statistics strategies (see the Resources & Samples there also).

Enabling MDT support

The explain considerations and uses of MDTs by the query optimizer must be explicitly enabled. The query optimizer's default behavior is to ignore MDTs.

The special behaviors of the MDT also affects whether the MDT is considered. The attributes that affect the considerations and usage of MDTs are:

- ENABLE | DISABLE QUERY OPTIMIZATION (Enables the MDT(s) for SQL SELECT statements)
- MAINTAINED BY USER | MAINTAINED BY SYSTEM (Manual or Automatic)
- REFRESH DEFERRED | REFRESH IMMEDIATE (Batch or Real-time)

[illegible] a process other than [illegible] to populate the RESULT. Use [illegible] the option [illegible] must be set to TRUE.

Appendix A provides a detailed description of these [illegible] options and all the possible values. The query runtime environment affects the optimization and use of RESULT. If an RESULT is to be considered:

- The environment must specify [illegible] access.
- The base table to be replicated with an RESULT must be a table or a table expression with this query.

Feedback on the query's use of an RESULT

Help on optimization feedback to determine whether the query optimizer has rewritten the user query to use an RESULT. Also, rely on the optimizer's feedback to know the access and processing of RESULTs. For example, the user's query has local predicates and the query is rewritten to use an RESULT, the new query might also have local predicates. The RESULT does not contain all the appropriate columns are available. This situation can be identified and resolved by analyzing the feedback.

Feedback on RESULT optimization and usage is provided through Visual Explain. Visual Explain and the RUNSTATS performance monitors (database monitors).

Figure 31 shows an example of visually exploring a query rewritten to use an RESULT.

Figure 31. Example of visually exploring a query rewritten to use an RESULT

An RESULT, Visual Explain does not explicitly highlight the use of an RESULT. Look for base or more of the base tables in the replicated with an RESULT. Using a naming convention can help identify RESULTs present in the query plan.

An ArcGIS 10.0(SP1) Visual Example is enhancement to provides an option to highlight any MXD's in the problems. This enables identifying the cause of MXD's not to execute.

To highlight an MXD in the query plan using Visual Example (see Figure 10):

1. Click the Show menu.
2. Select Highlight Misconfigured Query Profiles.

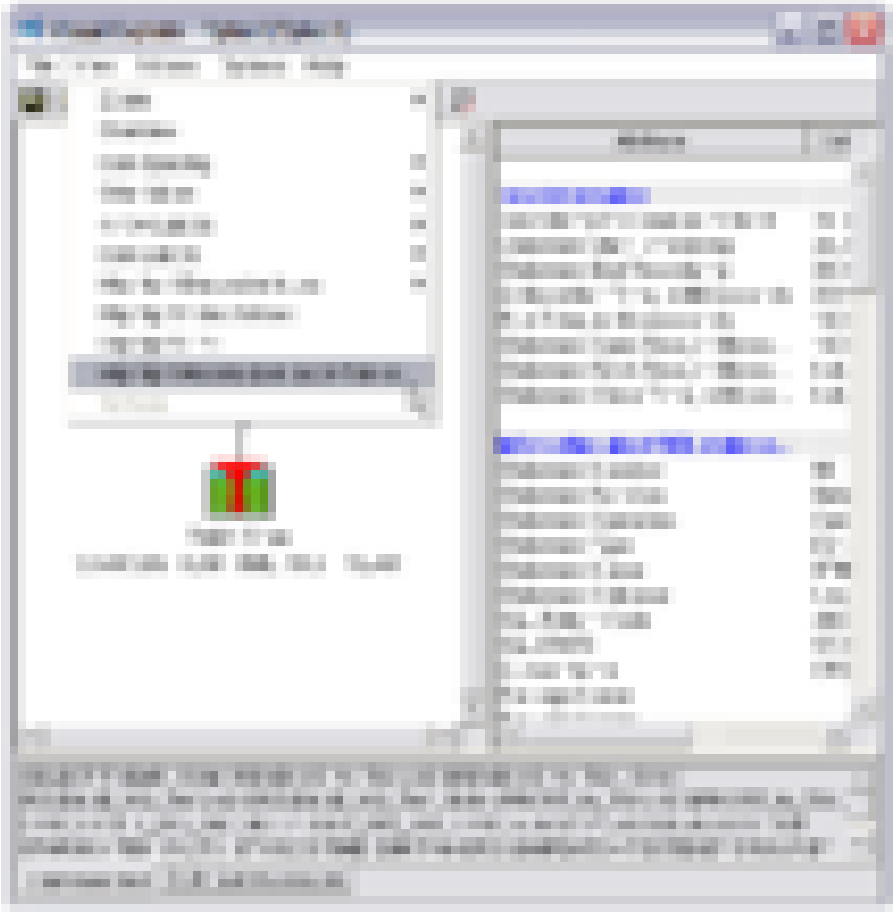


Figure 10 Example of highlighting an MXD in the query plan

When selecting the highlighting option, any BDTs in the process are augmented with an orange background. (see Figure 24)

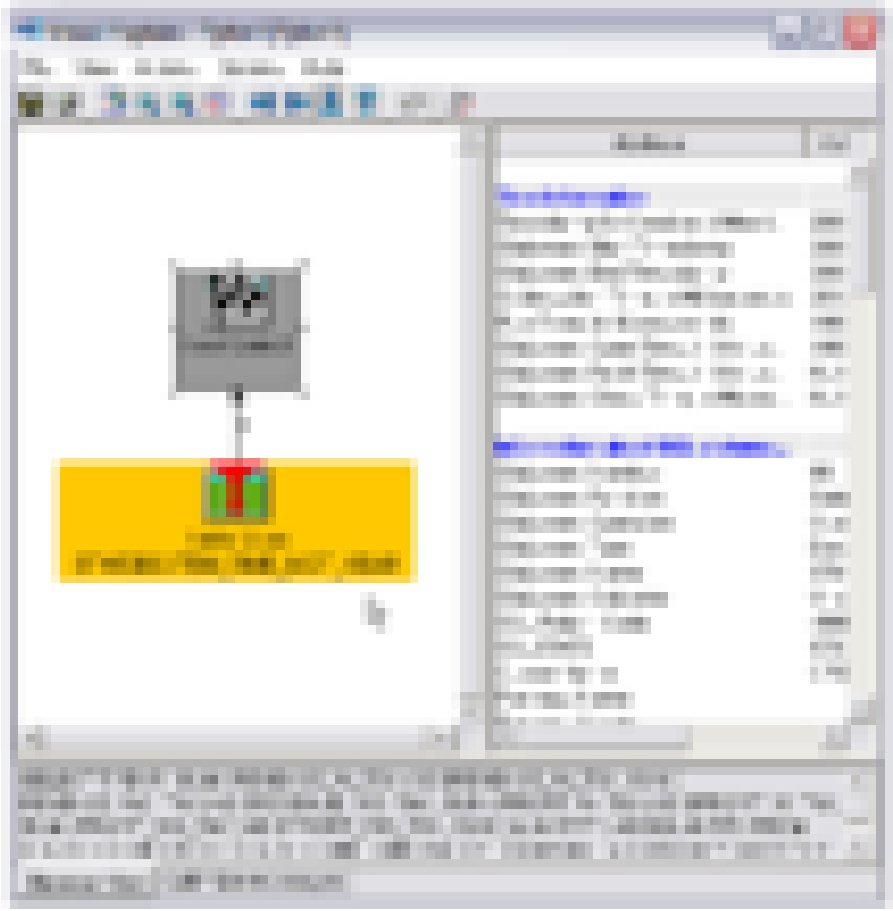


Figure 24 Example of BDTs that are augmented with orange background

As written to Oracle Trace, the detailed monitor data is enhanced to reflect the optimization and use of BIND. Specifically, the following information is provided:

- **[illegible]** multiple columns containing information about the BIND's that were examined. In most instances written only if BIND's are enabled and BIND's exist over the tables specified in the query.
- **[illegible]**, columns [illegible] contains '1' or '0', which indicates that an INDF replaced tables. This remaining information is based on the INDF instead of the base tables.
- **[illegible]**, columns [illegible] contains the [illegible] [illegible] [illegible] [illegible] [illegible] location and [illegible] contains the [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] and [illegible].
- **[illegible]**, columns [illegible] contains [illegible] access plan that results in the table becoming the INDF is not longer eligible as a candidate.

A simple query to determine which INDF's are used in what queries:

```
[illegible]          [illegible]
                     [illegible]
                     [illegible]
                     [illegible]
                     [illegible]
[illegible]          [illegible]
[illegible]          [illegible]
[illegible]          [illegible]
[illegible]          [illegible]
[illegible]          [illegible]
                     [illegible]
```

Note that the summary monitor is memory-based and does not collect the explicit uses of INDF's.

Additional information on Oracle Performance Monitors and the monitor data can be found in the publication: *[illegible] Database for Alpha or Database Performance and Query Optimization*. Appendix B provides a link to the INDF to SYSTEM Publications Information Center where you can find this manual.

Another very useful enhancement to BIND [illegible] is the ability to list all the INDF's on a given table and exclude the usage of the INDF's. For example, it is now possible to subscribe when a given INDF is used by the optimizer and how many times it is used. Conversely, it is possible to determine that a given INDF has never been used.

To view MBOs that are based on an existing index, use Oracle Navigator → Database (see Figure 26):

1. Navigate to and open a Schema.
2. Open Tables and right-click a specific index.
3. Select Show Materialized Query Tables.

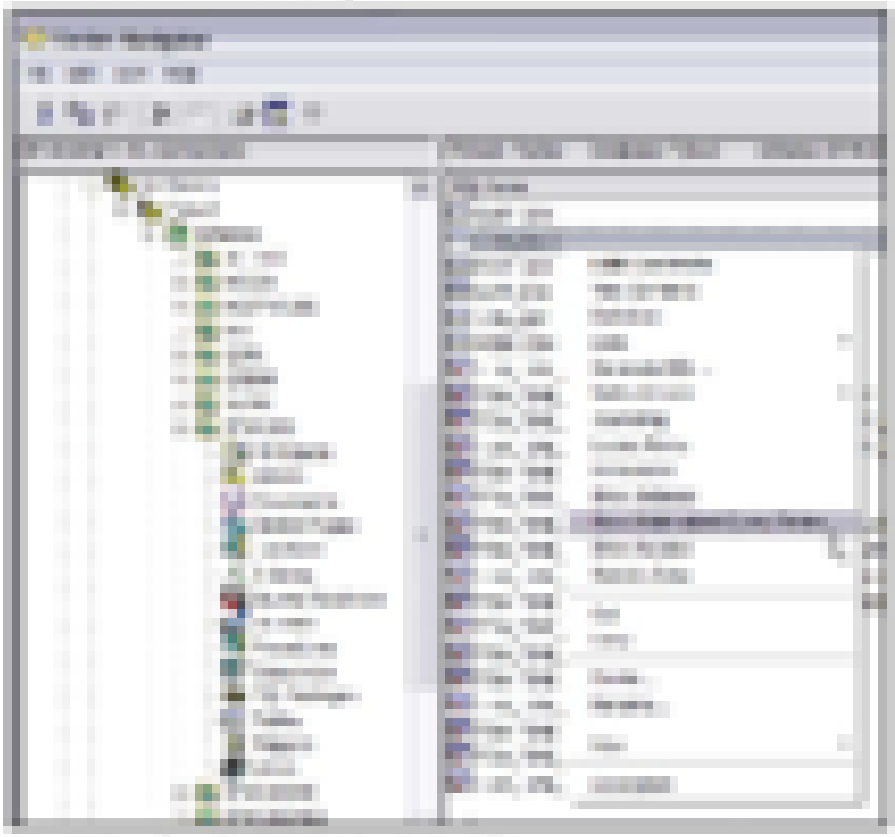


Figure 26: Using Oracle Navigator – Indexes to show MBOs

Note that the statistics can also be accessed through an application programming interface (API).

The report that is provided can be used to verify which Array MSCS are layout out the same. The report shows descriptive information such as the following (see Figure 35):

- Object Array name
- Object short (symbolic) name
- MSCF cluster Array unit name
- MSCF membership (Yes or No) for replication

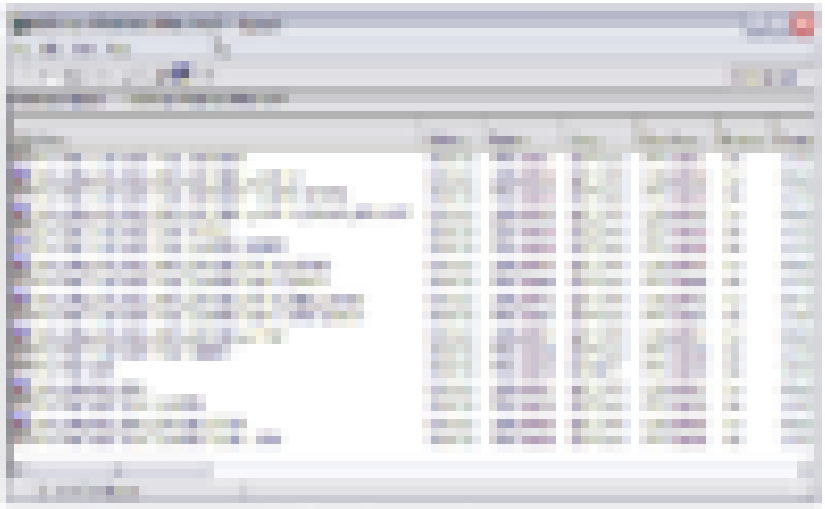


Figure 34 - The report generated by Array Analyzer - Inventory

Scrolling to the right of the report shows information on when the MSCF was used (see Figure 35):

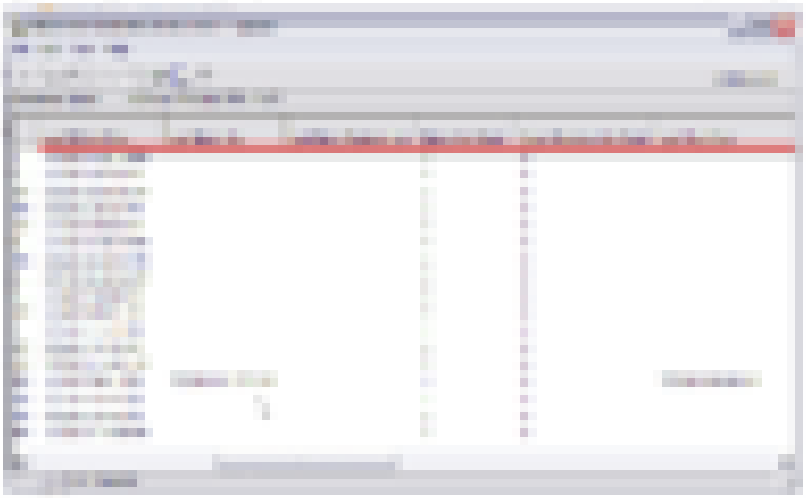


Figure 35 - Scrolling to the right of the report generated by Array Analyzer - Inventory

The **Last Query Use** column shows the timestamp when the MQT was last used by the optimizer to explain even specified tables in a query.

The **Query Use Count** column shows the number of times the MQT was used by the optimizer to explain even specified tables in a query.

The **Last Query Statistics Use** and **Query Statistics Use Count** columns are currently not relevant, not populated. The query optimizer does not use MQTs for statistics.

Designing MQT refresh strategies

DB2 for i5/OS does not automatically keep MQTs synchronized with the base tables. When the base tables change because of insert, update or delete activity, there can be a difference between the contents of the MQT and the base tables. This difference represents the data latency.

It is the responsibility of the user or programmer to refresh or maintain the MQT data. If queries access MQTs that are not maintained as the base tables change, the query results increasingly diverge from the results obtained when querying the base tables directly.

The most direct method to refresh an MQT is to issue the **REFRESH TABLE** statement:

```
REFRESH TABLE [library/]MQT1;
```

We assume that this is a full refresh of the MQT and causes the following:

1. Any indexes on the MQT are removed.
2. The contents of the MQT are removed.
3. The underlying SQL query is run.
4. The MQT is repopulated from the base tables.
5. Any indexes on the MQT are recreated.

The same considerations apply to refreshing or maintaining an MQT as the initial creation and population. More importantly, the time and resources available to refresh or maintain the MQT might be restricted because of other data processing. Programmer intervention might be necessary or advantageous to create an appropriate MQT refresh strategy that meets the business and technical requirements.

Some data and query environments lend themselves nicely to a periodic MQT refresh process. Many business queries involving quarter to quarter or year to year comparisons of data in MQTs can be used. The MQT will only need to be refreshed when that particular data changes. For example, if reports are based on a full year over year query, the MQT that contains the aggregated data (representing Year 1/Month 1) can be refreshed as part of the month-end processing.

IBM and ISV and other environments where data is loaded on a periodic basis. These periods provide a natural opportunity and in some cases a requirement to refresh or maintain the MQTs. If the MQTs are not to be kept in synch (for example, end of day processing), the MQT refresh or maintenance strategy can also occur on a daily basis. Refreshing the entire MQT does not to incorporate only one day's worth of data might be inefficient. In this case, the MQT can be maintained (not refreshed) by calculating the incremental changes to the MQT. You can fully recalculate the MQT's that are based on these tables can be refreshed or maintained using the same logic as they use the fully refreshed using the recreating method discussed earlier. DB2 triggers created on the base tables can provide a mechanism for initiating the MQT maintenance. As the base tables change, the appropriate MQTs can be changed as well.

When using a process other than [redacted] to perform a full refresh of the MQT, it is a good practice to use the following steps:

1. Disconnect any volume switches on the MQT.
2. Set the isolation level to uncommitted read (UR) for the [redacted].
3. Drop any indexes on the MQT.
4. Delete all the rows in the MQT.
5. Calculate the aggregates and populate the MQT.
6. Create any indexes on the MQT.
7. Refresh any volume switches on the MQT.

It is important to test and verify the MQT refresh or maintenance process before using it in a production environment.

Testing and tuning MQT refresh strategies

Test and verify that the MQT refresh and maintenance processes are tuned to meet or exceed the response-time requirements. If the refresh time exceeds the processing window, the MQT creation, usage and maintenance strategy must be amended, or additional processing resources must be supplied. For example, a full-refresh strategy might be too time-consuming, but an incremental maintenance strategy is acceptable. When implementing an incremental strategy, bear in mind the changes to the underlying base tables. Be sure to maximize and understand any additional time and resources required. In other words, the resource requirements of MQTs encompass the time necessary for the initial update and delete operations on the base tables.

Using the DB2 Performance Monitors and Collection Services tools can assist you with understanding the component workload and resource utilization of the refresh and maintenance process. You can get the most out of the available resources to identify overcommitted resources. For example, when initially refresh processing in parallel, if processor resources are underutilized, a higher degree of parallelism can be used to help increase throughput. On the other hand, if all of the processor resources are fully used, the parallel degree is adequate, or too high, for some processors. Advanced configurations where the I/O subsystem (memory and disk units) is capable of supporting the processors available.

Planning for success

Prior to creating MQTs, gather some baseline metrics on what SQL segments are slowest. You also need to determine the frequency of the SQL queries and how the queries are optimized and run, you can do this by using tools such as the DB2 Performance Monitor. Also, gather some baseline information on how the system resources are used. Collection Services is a good tool to use for this. After understanding MQTs, this baseline information can be used to quantify any differences or reduction in performance of the applications.

Seriously consider creating a benchmark to test any MQT creation, refresh and usage strategies. A good place to run a benchmark or proof of concept is the IBM Benchmarking and Proof of Concept Centers in Rochester, Minnesota or Montpellier, France. Understanding the costs and benefits of MQTs before implementing in a production environment is a critical success factor. [redacted] Appendix A provides a Web site listing for more information about the IBM Benchmarking and Proof of Concept Centers.

Summary

With the latest version of DB2 for i5/OS, IBM continues to deliver additional features and functionality to assist with the implementation of robust data-centric applications. The ability to create and use DB2/I provides you another option for high performance query processing. When paired along with the aforementioned publications, provided below, guidance and insight into using this new feature.

Appendix A provides a Web site listing for the latest information regarding DB2 for i5/OS support of DB2/I

Appendix A: SQL query engine details

With Derby 10.0.2.1, a newly reengineered SQL query engine was introduced. This new query engine is referred to as DRDA. The original query engine is referred to as the classic SQL engine. DRDA replaces classic, as a result almost all queries were optimized and run by DRDA. With the availability of Derby 10.0.2.1, your client may have queries were optimized and run by DRDA. Only new queries optimized by DRDA were implicitly new Derby 10.0.2.1 with a query definition optimized by DRDA can be implicitly used in the query plan.

DRDA query language

In Derby 10.0.2.1, DRDA is not capable of optimizing and running queries that contain or use:

- CASE predicates
- LIKE predicates
- Column translations such as UPPER, LOWER and CAST conversions
- Alternate collating sequences and sort sequences
- An INTERSECT, EXCEPT, or ORDER BY clause
- Check triggers
- Correlated subqueries
- Logical File references
- References to tables in physical files that have Select/Omit logical files
- References to tables in physical files that have logical files with mapped or derived keys
- Overlapping tables
- Non-ODBC interfaces such as the CLI/ODBC, OPNQRYF and Distributy APIs

In Derby 10.0.2.1, DRDA is not capable of optimizing and running queries that contain or use:

- Column translations such as UPPER, LOWER and CAST conversions
- Alternate collating sequences and sort sequences
- Check triggers
- Correlated subqueries
- Logical File references
- References to tables in physical files that have Select/Omit logical files
- References to tables in physical files that have logical files with mapped or derived keys
- Overlapping tables
- Non-ODBC interfaces such as the CLI/ODBC, OPNQRYF and Distributy APIs

If the query contains any of these items, then CDB is used to optimize and run the query. DRDA is not considered or used.

Optimizer trace options for MEMORY

optimizer_trace_offset, optimizer_trace_limit

This option controls the query optimizer's recognition and use of MEMORY:

- 'DEFAULT' - The default value is 'ON,ON'.
- 'ON,ON' - MEMORY are not used in query optimization or implementation.
- 'ON,' - The user-maintained, default statistical query tables can be used.
- 'OFF,OFF' - User-maintained MEMORY can be used.

optimizer_join_buffer_size

This option further determines which MEMORY are eligible to be used, based on the last time a MEMORY table's statistics were done.

- 'DEFAULT' - The default value is 0. No MEMORY can be used.
- 'ON' - Any tables indicated by the optimizer_join_buffer_size option. If the table's statistics are too old (older than 2 weeks or 1,209,600 seconds), the query optimizer will not use the MEMORY table's data statistics, but the table can be recalculated. Note that optimizer_join_buffer_size MEMORY statistics will calculate when the server is busy.
- 'Timestamp_Duration' - Only tables indicated by the optimizer_join_buffer_size option that have a MEMORY table's statistics older than specified timestamp duration are used. This is a STRING,INT number that indicates a timestamp duration since the last MEMORY table's recalculation.

The optimizer_switch options 'index_merge_intersect', 'prefer_ordering_index', and 'condition_pushdown_for_subquery' are used.

Appendix B: Resources

About the author

[redacted]

Acknowledgments

[redacted]

Trademarks and special notices

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

IntelliStation, IBM, the IBM logo, ThinkCentre and System x are trademarks of International Business Machines Corporation in the United States, other countries, or both. Intel and all Intel-based trademarks are trademarks of Intel Corporation in the United States, other countries, or both.

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

Information is provided "AS IS" without warranty of any kind.