



Creating and using Materialized Query Tables in IBM Db2 for i

*Kent Milligan
Michael Cain
IBM Technology Expert Labs*

June 2023

Table of contents

Abstract.....	1
Introduction	1
Prerequisites.....	1
MQT overview	2
MQT implementation considerations	4
Analyzing the data model and queries.....	5
Requirements and limitations	6
Natural environment for MQTs	9
Designing MQT definitions	10
Designing MQTs that are based on queries	11
Designing MQTs that are based on data models	14
Designing MQTs that are based on hierarchies	15
Creating MQT objects	18
CREATE TABLE Example	18
ALTER TABLE Example	19
Anatomy of an MQT	21
Populating MQTs	23
Indexes to improve MQT population performance	23
Environment to support MQT population performance.....	24
Cascading MQT population	24
MQT aggregation methods and strategies	26
MQT population: Programmer intervention	28
Employing a parallel population process	30
Testing and tuning MQTs.....	31
Enabling MQT Support	31
Feedback on MQT usage	32
Designing MQT refresh strategies	37
Testing and tuning MQT refresh strategies	38
Planning for success	38
Summary.....	39
Appendix A: Resources	40
About the authors.....	40
Trademarks and special notices	41

Abstract

This paper discusses the Materialized Query Table (MQT) support within IBM Db2 for i. This support can be used to enable high-performance query processing.

Introduction

On any platform, good 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 i, which provides a robust set of technologies that assist with query optimization and performance.

This paper introduces Db2 for I materialized query tables (MQT) and looks at their design, creation and use.

Prerequisites

It is strongly recommended that database administrators, engineers and developers who are new to the IBM i platform, or new to SQL, learn the skills covered in the Database Engineer (DBE) Enablement - SQL Performance offering. This offering provides in-depth information on the way to architect and implement a high-performing and scalable Db2 for i solution. You can find more information about this offering: ibm.biz/Db2iExpertLabs. Among other things, this workshop teaches the DBE or developer the proper way to architect and implement a high-performing Db2 for i solution. For more information about delivery of this offering, contact the author.

The points discussed in this paper assume some knowledge of Db2 for i. It is helpful to refer to and familiarize yourself with the information contained in the following publications contained in the IBM i documentation (at ibm.biz/db2iBooks).

- Db2 for i SQL Reference
- Db2 for i Database Performance and Query Optimization

The *Indexing and Statistics Strategies* white paper (at ibm.biz/db2Papers) is also a valuable resource that should be consulted.

MQT overview

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 based on an underlying table or set of tables. These underlying tables are referred to as the base tables. By running the appropriate aggregate query one time (by using the base tables and then storing the results so that they are accessible on subsequent requests), it is possible to enhance data processing and query performance significantly.

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

- Aggregated or summarized data that covers one or more subject areas
- Joined and aggregated data covering a set of tables
- Commonly accessed subset of rows (that is, a specific piece of data)

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

- Query1 requests the revenue figures for retail-group items sold in the central region each month of the previous year.
- Query2 requests the revenue figures for retail-group 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 by group. The data required can easily involve millions or billions of transactions that are stored in one or more 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, adapting queries to use an MQT directly might not be a trivial exercise for the user.

Though MQTs can be directly specified in a user's query, their real power comes from 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). This shortcut can drastically minimize the amount of data read and processed (see Figure 1).

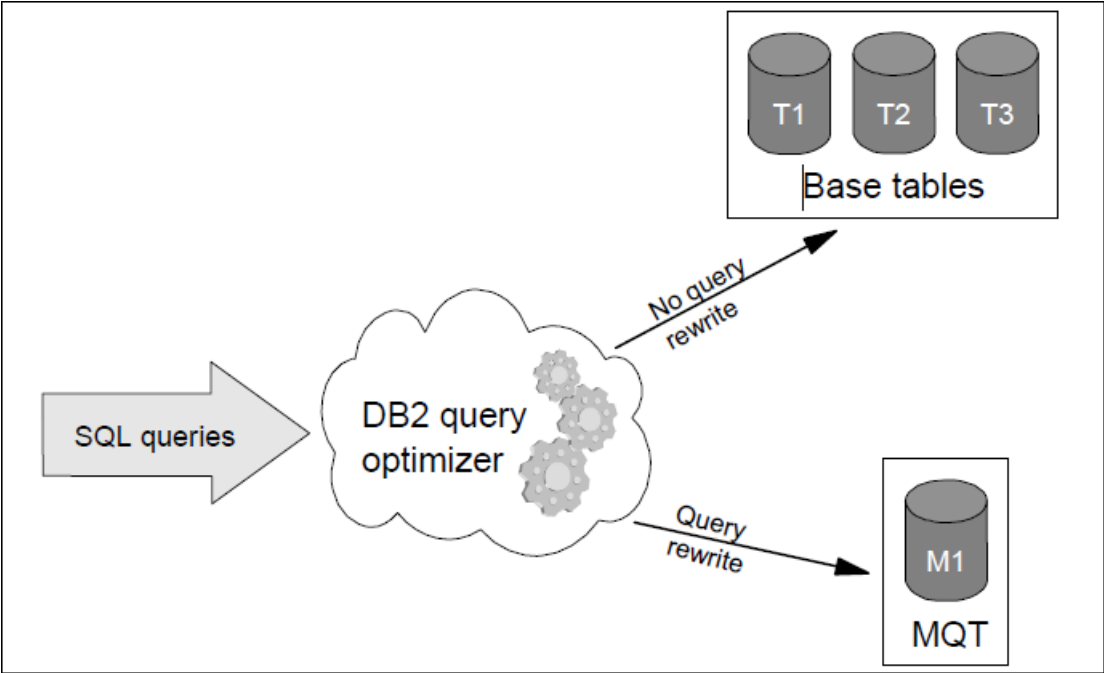


Figure 1: Query Optimization and MQTs

IBM performed some internal testing to quantify the performance impacts of an MQT. A grouping query that was issued against a 1.2 billion-row table took seven minutes to run without an MQT. With an MQT available, the same query took well under 0.7 seconds. If this type of query runs several times a day, the creation of an MQT can save 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 MQTs for the expected SQL requests, maintaining MQTs when the base tables are updated, recognizing the usefulness of an MQT, and supporting the ability to actually use the MQT for the query.

Db2 for i 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 gets different 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 analytics and strategic reporting, there can be (and in some cases, needs to be) some deferral 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 refreshing the MQTs when the volume of change activity is high or the base tables are large. This overhead includes:

- Temporary space when creating and populating the MQTs 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 for implementing MQT are:

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

Analyzing the data model and queries

Determining the proper MQT creation and usage strategy relies on analyzing and understating the data model, the data itself, the queries and the response-time expectations. This analysis can be proactive or reactive in nature. In practice, both approaches are necessary. The data model-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.

MQTs provide the most benefit when the queries are frequently aggregating or summarizing similar data from many rows (see Figure 2).

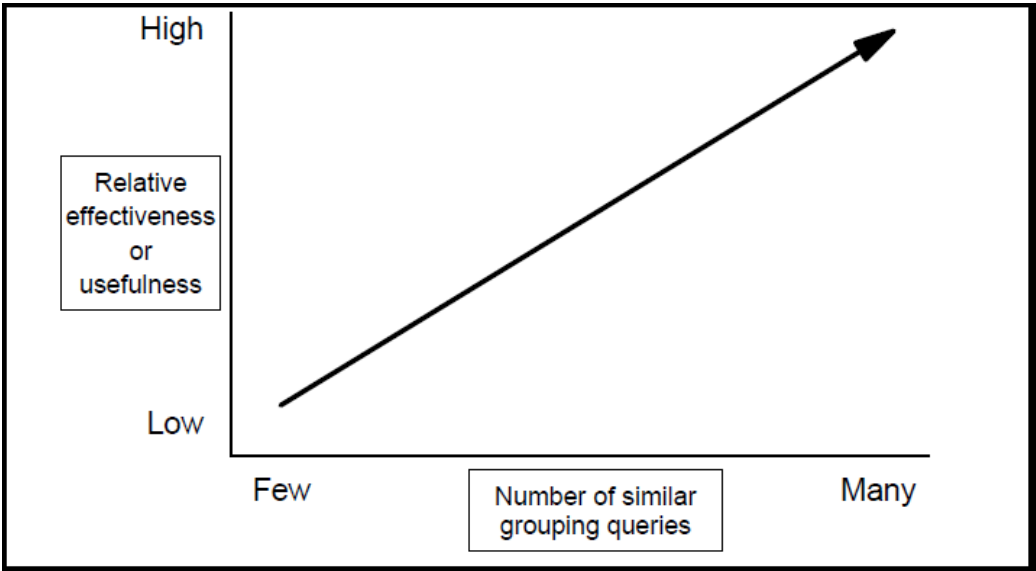


Figure 2: MQT usage with a number of similar queries

MQTs provide the most benefit when user queries are frequently aggregating or summarizing data that results in only a few groups. In other words, as the ratio of base-table rows to distinct groups approaches one-to-one (1:1), the effectiveness of the MQT diminishes (see Figure 3).

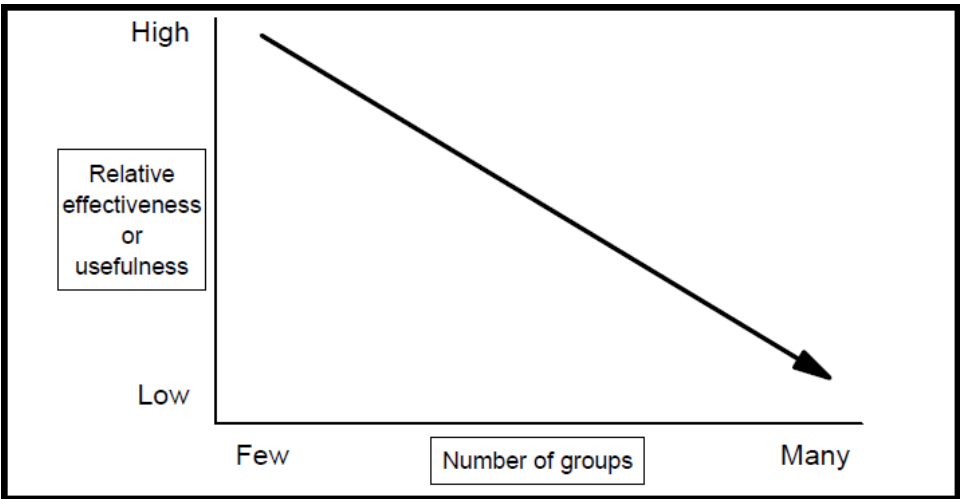


Figure 3: MQT diminished effectiveness as number of groups increases

Furthermore, if the SQL requests select very few rows from the base tables, the data processing required is low and the query-response time is fast, without the need for pre-summarization of the data. For example, a query selects all transactions for **Year = 2005**, **Month = 'June'** and **Customer = 'John Doe'**. The SQL request causes the database engine to access and aggregate 100 rows out of millions 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 **Year = 2005** and **Month = 'June'**. This SQL 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 MQT has to be refreshed, the less effective the MQT might be. This assumes minimal latency between the base tables and the MQTs. If the MQTs require refreshing less often and there is an adequate window of time to perform the refreshes, more MQTs can be employed.

The more often the MQT must be refreshed, the less effective the MQT might be. This assumes minimal latency between the base tables and the MQTs. If the MQTs require refreshing less often and there is an adequate window of time to perform the refreshes, more MQTs can be employed.

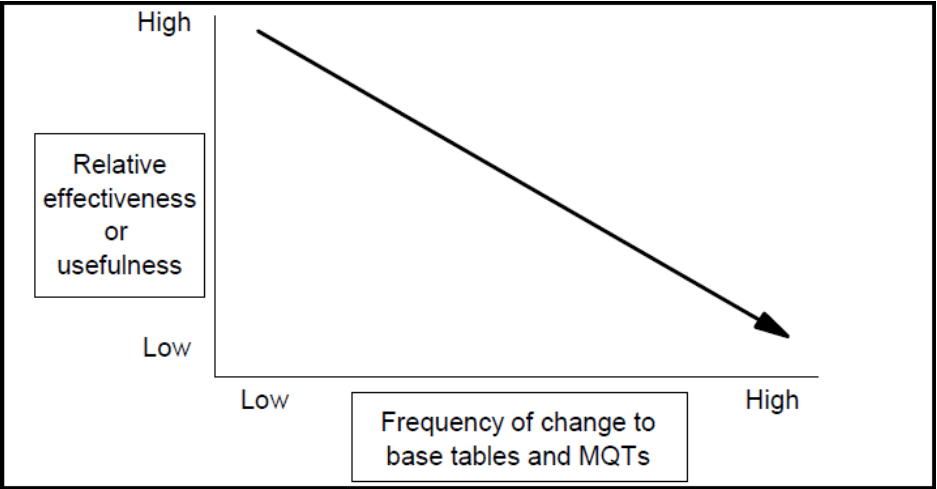


Figure 4: Frequency of change to base and MQT tables

Requirements and limitations

Analyzing the data model and application reveals any requirements for grouping and aggregations, along with common and frequent join criteria. It is important to pay particular attention to implicit 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, MQTs can provide help. For example, a detailed transaction table has a time hierarchy of Year / Quarter / Month / Week / Day, and these periods are specified as grouping criteria on a periodic basis. One or more MQTs can be created to support the queries against the detailed data, especially if these reports run frequently.

Using the SQL Performance Monitors (database monitors), queries can be captured and analyzed. To help construct the best set of MQTs, the analysis has to be designed to determine frequently queried 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 MQT can improve performance. It is also important to understand which query engine (Classic Query Engine [CQE] or SQL Query Engine [SQE]) optimizes the query request since only SQE supports the optimization of MQTs.

An MQT can contain almost any query definition, but the query optimizer supports only a limited set of query syntax when matching MQTs to a query. In general, the MQT-matching algorithm includes the following:

- Single table and join queries
- Views, common table expressions, and nested table expressions
- WHERE clause
- GROUP BY and optional HAVING clause
- ORDER BY clause
- FETCH FIRST n ROWS and LIMIT clause
- UNION clause
- Partitioned table references

An MQT can contain almost any query definition, but the query optimizer supports only a limited set of query syntax when matching MQTs to a query. In general, the MQT-matching algorithm includes the following:

- Scalar subselects
- User Defined Functions (UDFs) and user-defined table functions
- Built-in scalar functions implemented as UDFs (e.g., DAYNAME, REPEAT, etc.)
- Recursive Common Table Expressions (RCTE)

Additional details on MQT query-matching limit can be in found in the *Database Performance and Query Optimization* guide (ibm.biz/db2iBooks).

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

MQT Definition:

```
CREATE TABLE Example_MQT AS
(SELECT Geography, Region, Year, Month, SUM(Revenue) AS Total_Revenue,
      SUM(Quantity) AS Total_Quantity, COUNT(*) AS Rows_per_Group
 FROM Example_Transaction_Table
 GROUP BY Geography, Region, Year, Month)
DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER
```

Example queries that are allowed to use the MQT:

```
SELECT Geography,
       Region,
       Year,
       Month,
       SUM(Revenue) AS Total_Revenue,
       SUM(Quantity) AS Total_Quantity
FROM Example_Transaction_Table
GROUP BY Geography, Region, Year, Month;

SELECT Geography,
       Year,
       SUM(Revenue) AS Total_Revenue,
       SUM(Quantity) AS Total_Quantity
FROM Example_Transaction_Table
WHERE Year IN (2004, 2005)
GROUP BY Geography, Year;

SELECT Geography,
       Region,
       AVG(Revenue) AS Avg_Revenue,
       AVG(Quantity) AS Avg_Quantity
FROM Example_Transaction_Table
GROUP BY Geography, Region;
```

Example queries that are not allowed to use the MQT:

```
SELECT Geography,
       Region,
       Quarter,
       SUM(Revenue) AS Total_Revenue
FROM Example_Transaction_Table
GROUP BY Geography, Year, Quarter;
← Column not defined in MQT

SELECT Geography,
       Region,
       SUM(Revenue) AS Total_Revenue,
       SUM(Quantity) AS Total_Quantity
FROM Example_Transaction_Table
WHERE SOUNDEX(Geography)
      <> SOUNDEX('ASIA')
GROUP BY Geography, Region;
← SOUNDEX built-in function is a UDF
```

Natural environment for MQTs

Some application environments are very good candidates for MQT creation and usage. Business intelligence (BI) and data warehousing (DW) environments lend themselves to the advantages of pre-summarized data. BI and DW applications normally store and query vast quantities of data. BI and DW applications typically catalog and process data along hierarchies such as time and business subject areas. These hierarchies provide natural opportunities to create MQTs. Furthermore, BI and DW environments usually have clearly defined latency between the transaction data and the data warehouse data. For example, adding daily transactions to the data warehouse delivers a natural and consistent batch of data to the BI system on a periodic basis. Reviewing the hierarchies and the query requests within the BI environment yields a set of MQTs that can provide tremendous benefit and yet can also be maintained as part of the DW extract, transform and load (ETL) process.

Star-schema or snowflake-schema data models are specific cases where MQTs can be employed. Traditionally, the fact table contains detailed facts, or measures, that are rolled up as sums, averages and counts. The dimension tables contain the descriptive information and this 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 / territory). MQTs can be proactively defined to provide pre-aggregated data along the most commonly used levels.

In many cases, a pre-summarization 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 MQTs and the optimizer's query rewrite capability. By altering the existing summary tables to be MQTs, 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 MQTs. This decision is based on the estimated runtime cost for each set of data queried. Using the query optimizer to make this decision allows more flexibility. On the other hand, "if it is not broken, do not fix it" might be the best strategy for the existing pre-summarization process. It is important to pay particular attention to implicit or explicit hierarchies.

Designing MQT definitions

With any identified hierarchies or grouping criteria, laying out the MQTs 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 or groups:

- 3 years
- 12 quarters (3 years x 4 quarters)
- 36 months (3 years x 12 months)
- 156 weeks (3 years x 52 weeks)
- 1,095 days (3 years x 365 days)

Executing a query that groups all 100 million rows by the most detailed level (Day), results in only 1,095 distinct groups. Yet, continually processing all 100 million rows is time-consuming and resource intensive. This is where designing an MQT is valuable.

By providing an MQT that represents all the data grouped by Year / Quarter / Month / Week / Day, the query optimizer can rewrite the query to use the MQT instead of the base transaction table. Rather than reading and processing 100 million rows, the database engine reads and processes only 1095 rows from the MQT, resulting in a significant boost in performance.

It is compelling to create MQTs for the other levels of this time hierarchy (for example, Year / Quarter), but in this case, there is little benefit to be gained. The query optimizer is able to use the MQT with Year / Quarter / Month / Week / Day and regroup the MQT rows to build aggregates for Year / Quarter. The query does not need to match the precompiled results exactly. Reading and processing 1095 rows to build 12 distinct groups is significantly faster than reading and processing all 100 millions rows of the transaction table. Yet, accessing an MQT with 12 rows based on Year / Quarter is not that much faster than accessing an MQT with 1,095 rows. In other words, the largest benefit is derived from pre-aggregating 100 million transactions down to 1,095 groups.

If there is a requirement to maintain relatively static figures for each level of the hierarchy, MQTs can be created for each level. This is one way to take advantage of the data latency inherent in MQTs. In this case, building the lowest level first and then using that level to build and maintain the next MQT is the preferred approach. This avoids reading and processing the detailed transactions, thus minimizing the time and resources required to build the next level of groups. Using the previous example of a time hierarchy, it is advantageous to create the most detailed level first (Year / Quarter / Month / Week / Day), and then, at the appropriate time, to use this table to create a higher level (for example, Year / Quarter / Month) at the appropriate time. This cascading approach minimizes the time and effort to build or refresh the various levels of MQTs.

Another MQT benefit is the ability to minimize or avoid the joining of rows. Because joining many rows together can result in high physical I/O operations and potentially long response times, full or partial denormalization of the data model can significantly increase query performance.

MQTs can be created from one base table or many base tables. When creating MQTs over many base tables, the MQT is used to denormalize the base tables. This denormalization of data minimizes or eliminates the need to join rows during query execution (see Figure 5).

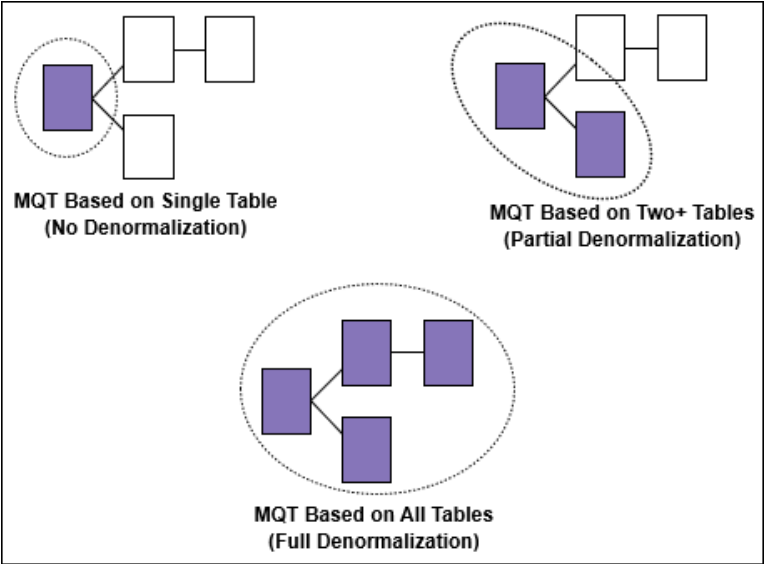


Figure 5: MQT tables based on single or multiple tables

MQTs can be created with local selection predicates against one or more tables. In this case, the MQT is considered to be sparse and only reflects the data represented by the specified local selection. Because the MQT only contains some of the data from the base tables, its overall usefulness might be decreased. Furthermore, any literals in the MQT definition and in the user's query are replaced with parameter markers during optimization. For example, the following query will be rewritten to replace any literal values with a parameter marker (i.e., ?):

<pre>SELECT Color, Amount FROM Some_Table WHERE Item = 11235 AND Amount > 700</pre>	\rightarrow	<pre>SELECT Color, Amount FROM Some_Table WHERE Item = ? AND Amount > ?</pre>
--	---------------	--

The MQT optimization process matches the value of the parameter marker used when creating the MQT to the value of the parameter marker in the local selection predicate of the user query. The optimizer verifies that the values of the parameter markers are the same; therefore, the MQT can be used. The MQT-matching algorithm also attempts to match where the predicates in the MQT and the query are not exactly the same. For example, if the MQT has a predicate `AMOUNT > 500` and the query has the predicate `AMOUNT > 700`, the MQT contains the rows necessary to run the query. The MQT is used in the query. The predicate `AMOUNT > 700` is left as local selection in the query. Therefore, the `Amount` column must be defined in the MQT.

Designing MQTs that are based on queries

When designing an MQT for a set of queries, it is important to account for all the columns projected in the set of queries. Either the MQT 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 appropriate tables.

For example, the following grouping query selects rows from a single table that match **Year = 2005** and **Month = July**, summarizing two columns (**Sales** and **Quantity**) and grouping by **Year / Month / Day**. The

MQT designed to assist this query must contain the projected columns Year, Month, Day, SUM(Sales) and SUM(Quantity). If one or more columns are omitted, the MQT is not useful.

```
SELECT Year,
       Month,
       Day,
       SUM(Sales) AS Total_Sales,
       SUM(Quantity) AS Total_Quantity
FROM Example_Transaction_Table
WHERE Year = 2005 AND
       Month = 'July'
GROUP BY Year, Month, Day
```

In another example that specifies a join and groups between two tables, either of two MQT designs can be used to assist this query:

- An MQT that is based only on **Trans_Table**
- An MQT that is based on both **Trans_Table** and **Date_Table**

```
SELECT D.Year, D.Month, D.Day,
       SUM(T.Sales) AS Total_Sales,
       SUM(T.Quantity) AS Total_Quantity
FROM Trans_Table T INNER JOIN Date_Table D
  ON D.DateKey = T.DateKey
WHERE D.Year = 2005 AND
       D.Month = 'July'
GROUP BY D.Year, D.Month, D.Day
```

An MQT that is based only on **Trans_Table** must contain the DateKey, SUM(Sales) and SUM(Quantity) columns. The query optimizer can specify a join between the MQT and **Date_Table**, regrouping the rows based on DateKey.

An MQT that is based on both **Trans_Table** and **Date_Table** must project the Year, Month and Day columns from Date_Table, and the SUM(Sales) and SUM(Quantity) columns from Trans_Table. The query optimizer can omit the join entirely because the MQT contains all the data required to complete the query.

In another example that specifies a join between three tables; either of two MQT designs can be employed to assist this query:

- An MQT that is based on all 3 tables
- An MQT that is based on two tables

```
SELECT C.Customer, D.Year, D.Month, D.Day, T.Sales, T.Quantity
FROM Trans_Table T, Date_Table D, Customer_Table C
WHERE D.Year = 2005 AND
       D.Month = 'July' AND
       C.Customer = 'IBM Corporation' AND
       D.DateKey = T.DateKey AND
       C.CustKey = T.CustKey
ORDER BY Customer, Year, Month, Day
```

An MQT that is based on all three tables (**Trans_Table**, **Date_Table** and **Customer_Table**) must project the Year, Month and Day columns from Date_Table, Customer column from Customer_Table and the Sales and Quantity columns from **Trans_Table**. Note that the DateKey and CustKey join columns are specified in the MQT's query definition, but the columns are not part of the MQT result data. In this case,

the query optimizer can omit the join entirely because the MQT contains all the data required to complete the query.

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

An MQT that is based on the **Trans_Table** and **Customer_Table** tables must project the DateKey, Sales and Quantity columns from **Trans_Table** and the Customer column from **Customer_Table**. The DateKey column facilitates joining the MQT to **Date_Table**.

In both cases, the query optimizer can omit the join between a pair of tables because the MQT contains all the data required for that pair of tables. The inclusion of the other table's join column allows the MQT to be joined to a table not represented in the MQT definition.

The two-table design, which includes a third table's join column, allows additional queries to use the respective MQT. Keep in mind that the number of rows (groups) in the MQT might be larger, based on the number of distinct join-column values (such as the join column specified in the GROUP BY clause).

Designing MQTs that are based on data models

MQTs 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 summarized, then foreign keys for one or more dimension tables must be included in the MQT. This allows for selection on the dimension table, joining from the dimension table to MQT, and regrouping of the MQT data.

The grouping on a foreign key column represents the most detailed level within a given dimension. Be aware that each additional foreign key that is added to the MQT as a grouping column results in more groups and less effectiveness. For example, if the TimeKey column represents 1095 distinct days or groups, adding the StoreKey grouping column can multiply the number of distinct groups by the number of distinct StoresKey values. If 100 stores have a transaction every day, this will result in the representation of 109,500 distinct groups in the MQT (1095 days x 100 stores) (see Figure 6).

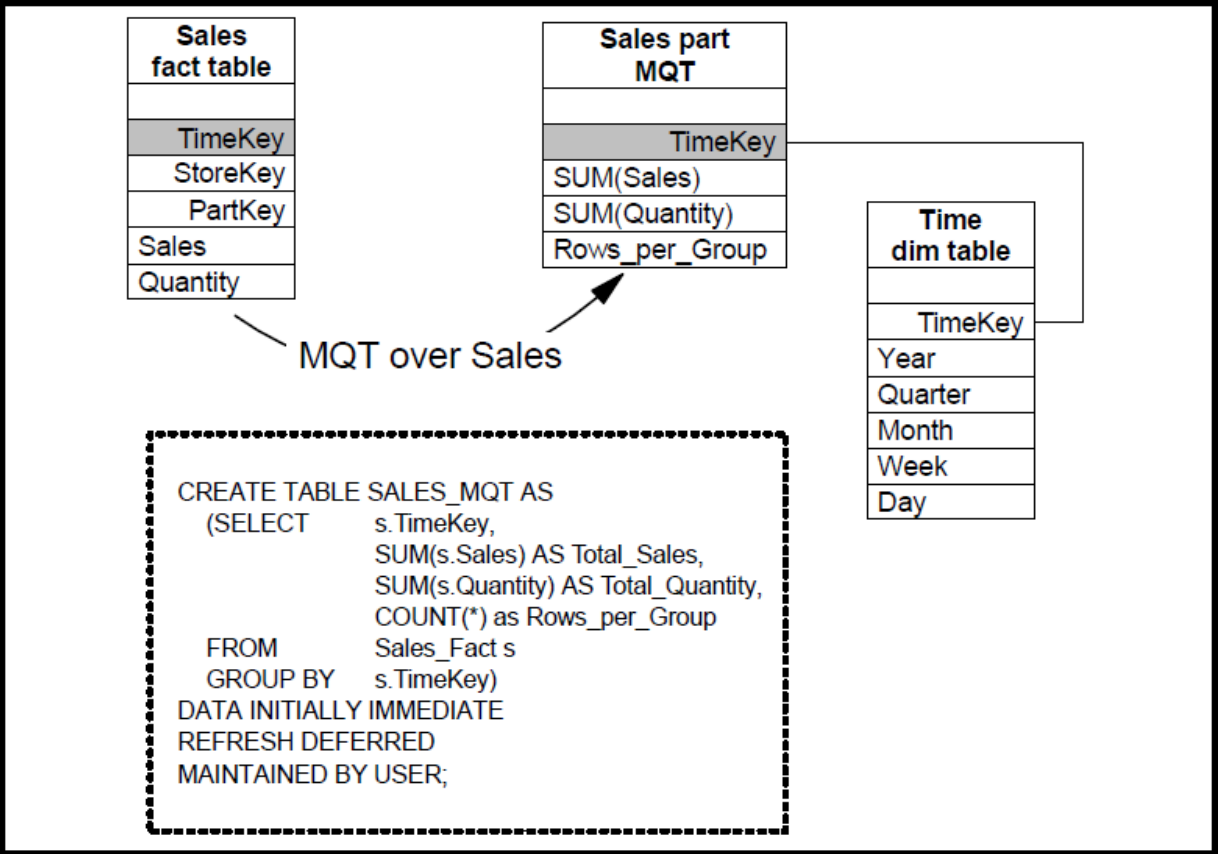


Figure 6: MQT over Sales table

If the fact table and one or more dimension tables are summarized, this results in an MQT that denormalizes the data model. By including all the pertinent columns in the MQT, the database engine can avoid joining the tables together (see Figure 7).

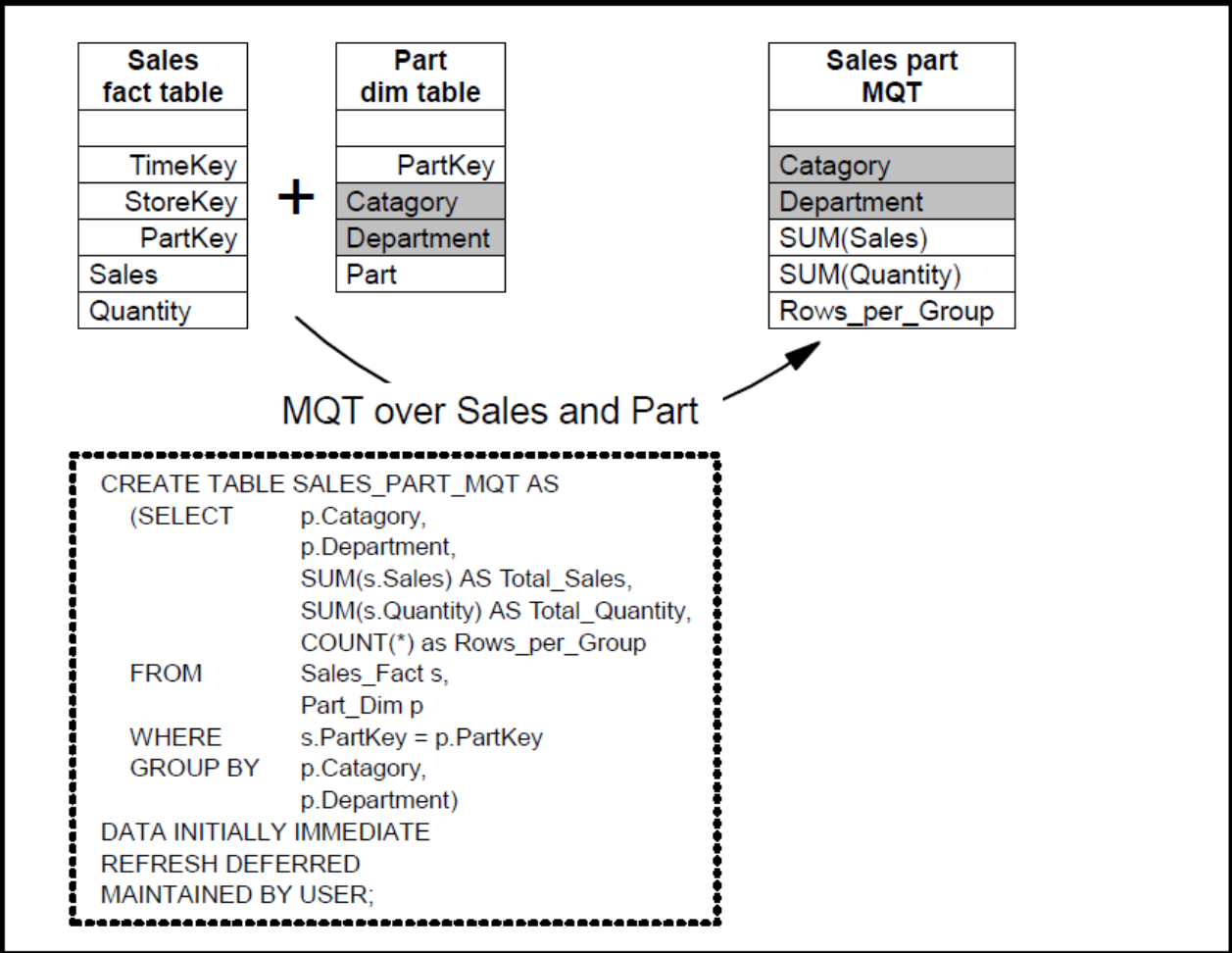


Figure 7: MQT over Sales & Part tables

Designing MQTs that are based on hierarchies

When creating an MQT, selecting the appropriate grouping columns can mean the difference between an effective or useless MQT. In a data model with implicit or explicit hierarchies defined, the grouping columns specified can allow the MQT to cover grouping queries at that level or higher in the hierarchy. This can be a very useful way to minimize the number of MQTs that must be created and maintained, though still gaining significant benefits from pre-aggregation of data.

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

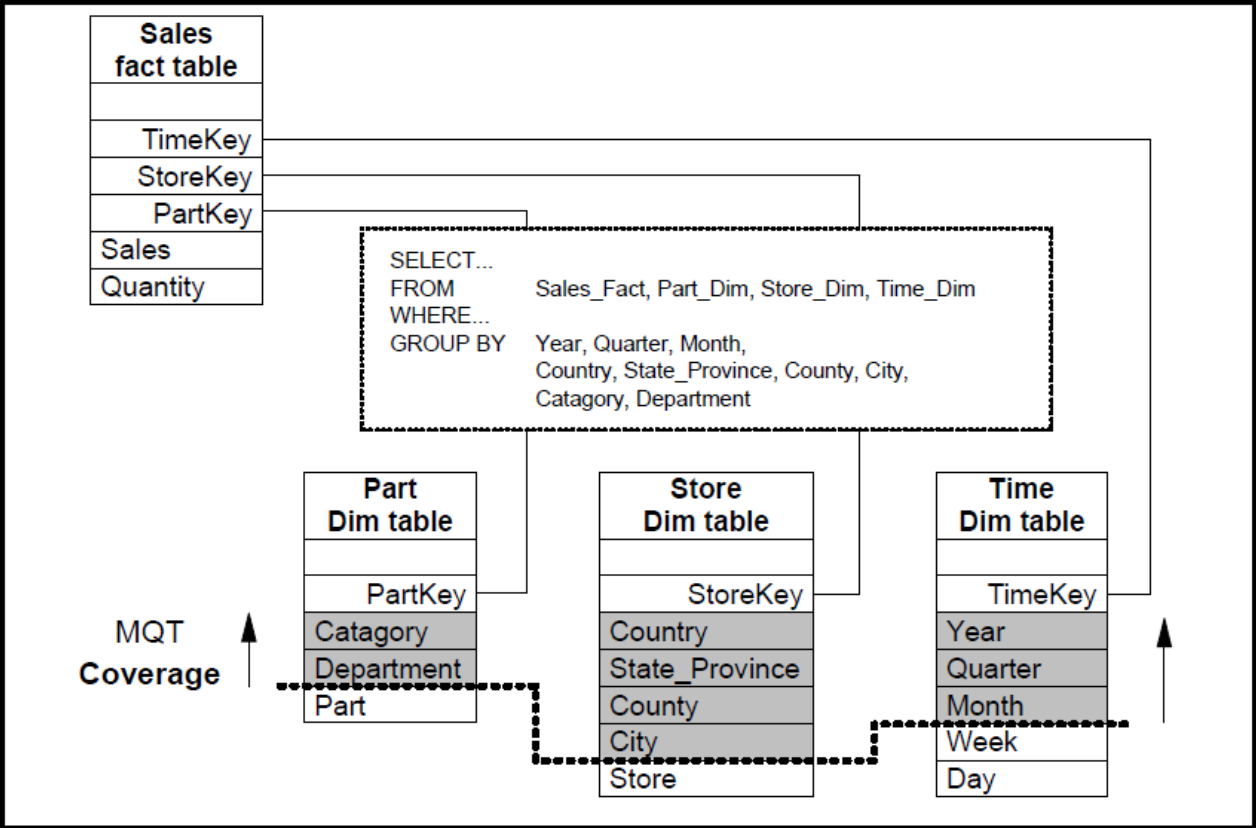


Figure 8: MQT coverage

Another case in which an MQT can help regards repeated requests for a distinct list of values where the 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:

```
SELECT DISTINCT Location
FROM Transaction_Table
```

If the table contains millions 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 for an MQT. By creating an MQT that contains the distinct list of location values, the query optimizer can use the MQT to satisfy the query with little time and resources. of the IBM i operating system

```
CREATE TABLE Locations_MQT AS
(SELECT DISTINCT Location FROM Transaction_Table)
DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER
```

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

It is important to keep in mind that implementing MQTs is not free. Besides the time and resources to perform maintenance, the optimization time for a given query increases as the optimizer considers more and more MQTs. It is best to design a few MQTs that provide the widest coverage and the largest benefit.

In general, consider creating MQTs for the following query classes:

- Queries with grouping and aggregation, where the ratio of rows to groups is high
- Queries with distinct values, where the ratio of rows to distinct values is high
- Queries with joins, where the number of results is high and fan-out occurs

Creating MQT objects

An MQT definition consists of a query and the attributes that are associated with the population, maintenance, and usage of the table. There are two methods for creating an MQT: CREATE TABLE or ALTER TABLE. Either method can be initiated through an SQL request or by using the graphical user interface of IBM i Access Client Solutions (ACS).

The CREATE TABLE method allows for the creation and population of new tables. The ALTER TABLE method allows an existing table to be modified into an MQT.

CREATE TABLE Example

Here's a CREATE TABLE statement used to create an MQT based on an existing table, Example_Transaction_Table, with the appropriate column definitions.

```
CREATE TABLE Example_MQT AS
  (SELECT Geography,
        Region,
        Year,
        Month,
        SUM(Revenue) AS Total_Revenue,
        SUM(Quantity) AS Total_Quantity,
        COUNT(*) AS Rows_per_Group
   FROM Example_Transaction_Table
   GROUP BY Geography, Region, Year, Month)
DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER
```

A new MQT can be created using the ACS graphical interface as shown in Figure 9 using the following steps:

1. Launch the ACS Schemas tool.
2. Navigate to the Schema where you want the MQT to reside.
3. Right-click on the Tables folder.
4. Select the New → Materialized Query Table action.

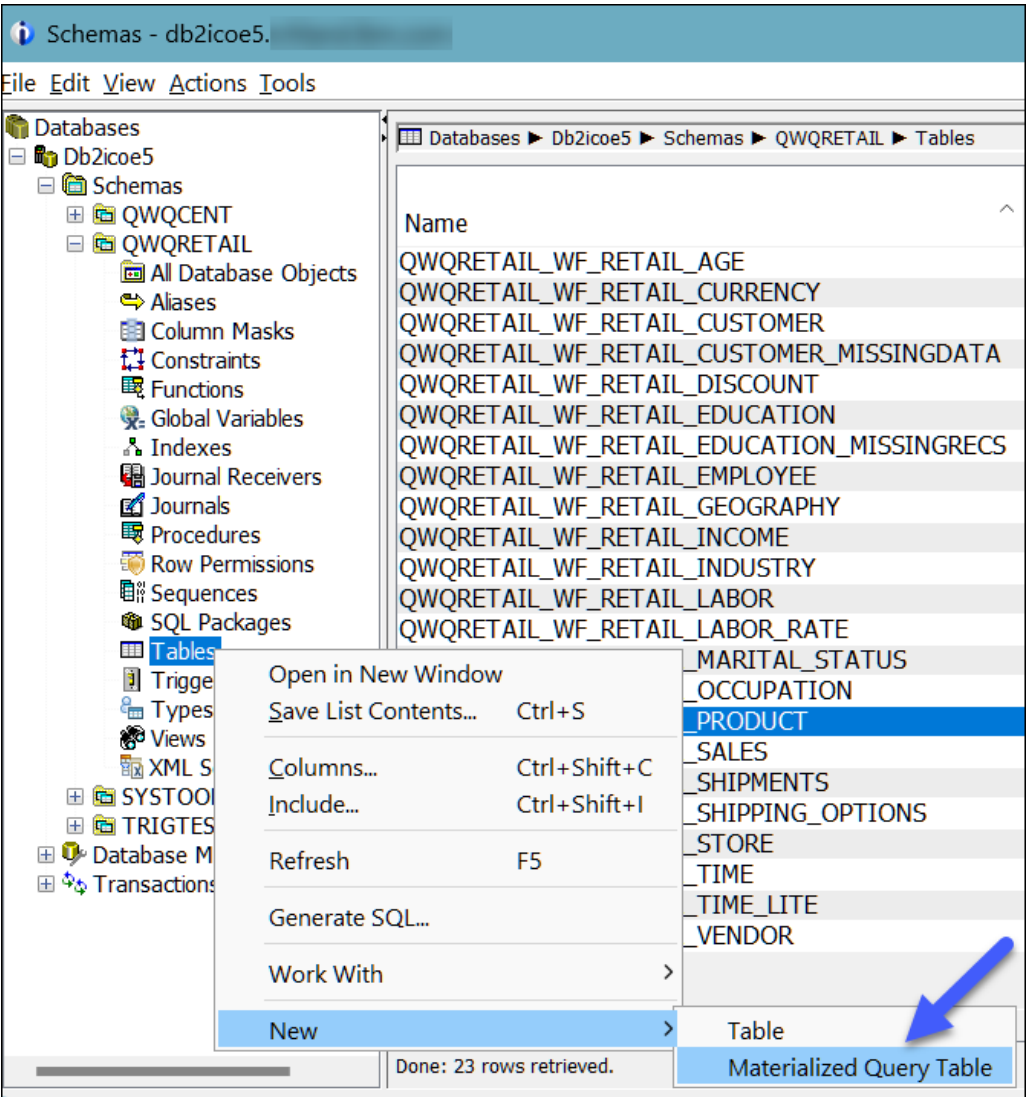


Figure 9: Create a new MQT using IBM i Access Client Solutions

These actions bring up a new window to specify the MQT attributes and definition.

ALTER TABLE Example

Here’s an ALTER TABLE statement used to change an existing table into an MQT based on an existing table, Example_Transaction_Table, with the appropriate column definitions.

```
ALTER TABLE Example_Summary_Table
ADD MATERIALIZED QUERY
(SELECT Geography, Region, Year, Month,
      SUM(Revenue) AS Total_Revenue,
      SUM(Quantity) AS Total_Quantity
FROM Example_Transaction_Table
GROUP BY Geography, Region, Year, Month)
DATA INITIALLY IMMEDIATE REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER
```

ACS can also be used to convert an existing table into an MQT as shown in Figure 10 using the following steps:

1. Launch the ACS Schemas tool.
2. Navigate to the Schema containing the table to convert
3. Right-click on the specific table
4. Select the Definition task.

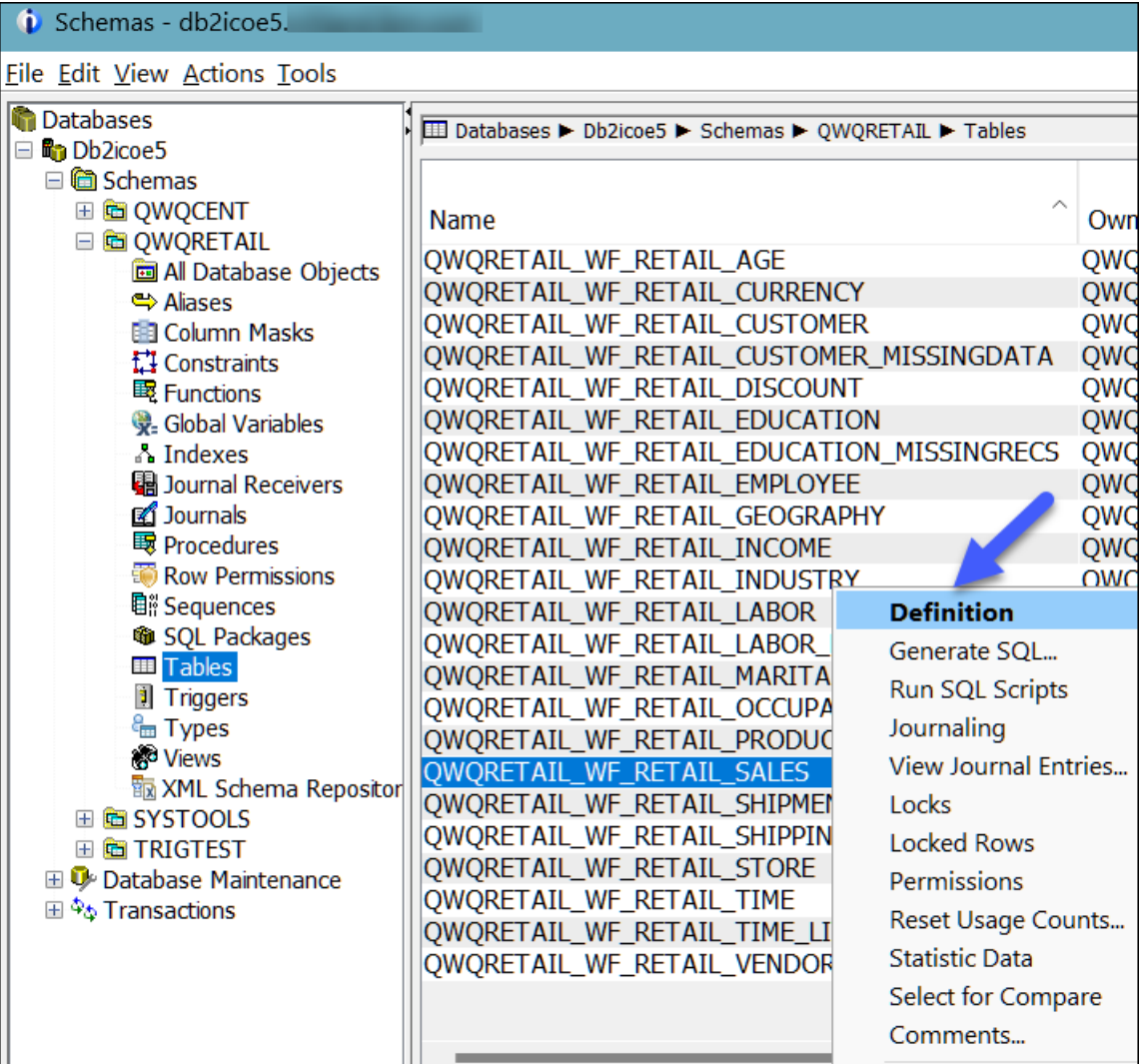


Figure 10: Alter existing table to become an MQT using IBM i Access Client Solutions

These actions bring up a new window to specify the MQT attributes and definition on the Materialized Query tab.

Anatomy of an MQT

Whether or not you are familiar with MQTs, it is important to understand the anatomy of the SQL statement used to create an MQT in Db2 for i. Different clauses control the population and maintenance of MQTs, and some functionality is not yet supported (see Figure 11).

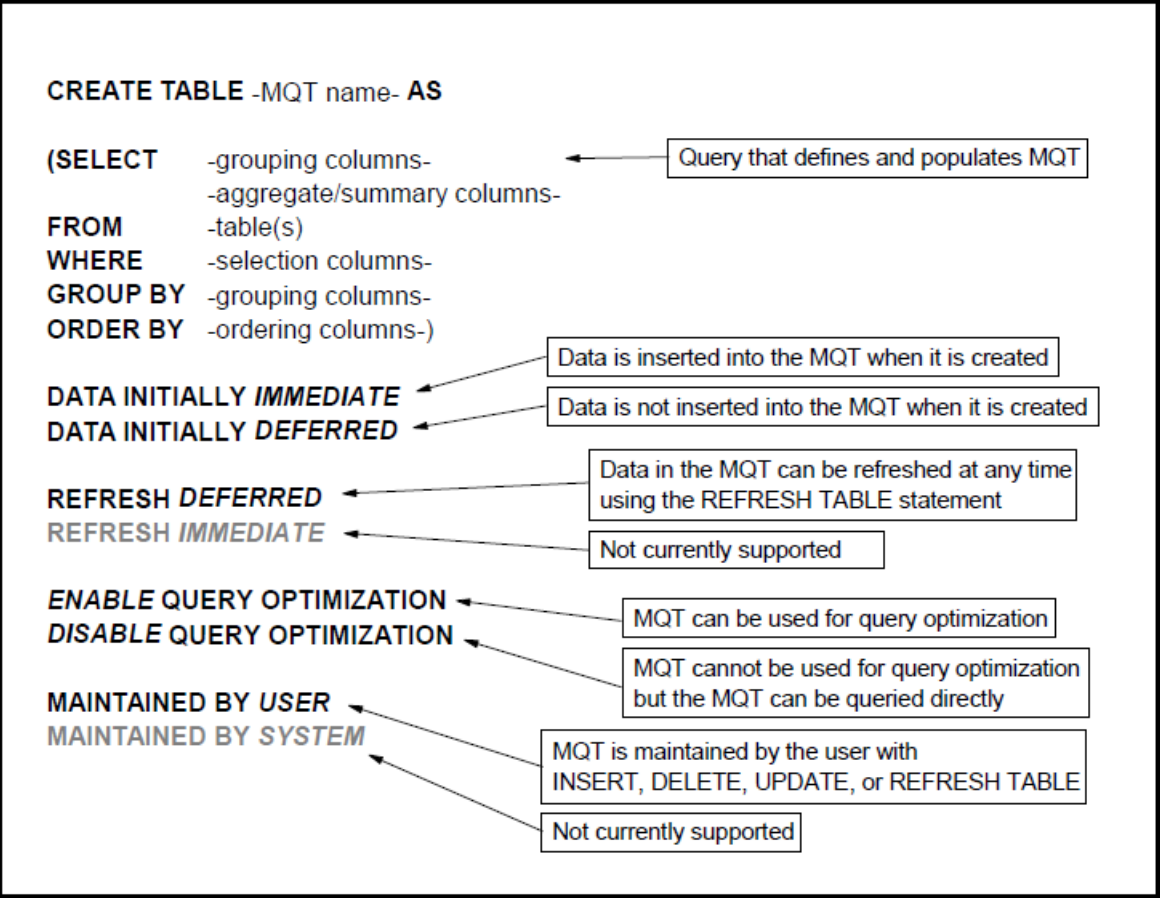


Figure 11: MQT Statement Anatomy

With the **REFRESH DEFERRED** and **MAINTAINED BY USER** clauses, Db2 does not automatically keep the MQTs synchronized with the base tables. When the base tables change, there can be a difference between the contents of the MQTs and the base tables. This difference represents the data latency.

The use of specific MQT-naming convention is helpful for quick identification during analysis and administration. Consider including the text *MQT* somewhere in the table name.

When laying out the materialized query definition, it is a good practice to specify all the columns that are considered additive facts or measures. At a minimum, these columns need to be used with the **SUM** function. As appropriate, other functions such as **AVG**, **MIN**, **MAX** and **COUNT** can be specified. Providing these columns and functions might allow the MQT to be more widely considered and used.

The number of rows per group provides the query optimizer with additional ways to take advantage of the MQT (for example, determining averages by using summations). It is always a good practice to provide the

function COUNT(*) in the SELECT clause of the MQT definition. If expecting to calculate the average of null capable columns, (for example, AVG(null capable column) in the query), COUNT(null capable column) must be provided in the MQT, not just a COUNT(*).

```
CREATE TABLE Example_MQT AS
  (SELECT Year,
         Month,
         Day,
         SUM(Sales) AS Total_Sales,
         SUM(Quantity) AS Total_Quantity,
         COUNT(*) AS Rows_per_Group
   FROM Example_Transaction_Table
  GROUP BY Year, Month, Day)
DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER
```

There are some restrictions for the query specified on the MQT definition. The *Db2 for i SQL Reference* should be consulted for a complete list of restrictions. Some of the more notable restrictions include the following:

- SELECT statement cannot contain a reference to another MQT or to a view that refers to an MQT.
- SELECT statement cannot contain a reference to a declared global temporary table, a table in QTEMP, a program-described file or a non-SQL logical file on the FROM clause..
- The Select statement cannot contain a reference to a global variable.

If the MQT definition includes the ENABLE QUERY OPTIMIZATION clause, the following additional restrictions apply:

- No references to special registers.
- No references non-deterministic functions.
- The ORDER BY clause is allowed, but is only used by the REFRESH table statement.

When creating MQTs, the actual population of the data can occur as part of the creation request, or anytime after creation. If the **DATA INITIALLY IMMEDIATE** attribute is specified, the MQT population is initiated as part of the creation phase. If the **DATA INITIALLY DEFERRED** attribute is specified, the MQT population is not done. If the data is initially deferred, the calculation and population can be initiated by using the **REFRESH TABLE** statement, or the process can be initiated and controlled by the programmer. By deferring, the user can determine the best time and mechanism for populating the MQT.

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

Prior to populating the MQTs, it is advantageous to verify whether the query optimizer will consider using the MQTs instead of the base tables. A simple method for doing this verification is to analyze the query plan for a few queries using the ACS Visual Explain tool. If the query optimizer replaces the base tables with the MQT, the MQT is shown in place of one or more base tables in the query plan drawn by Visual Explain. An example of such a query plan is shown later in this document. If the query optimizer rejects the MQT, further analysis and redesign can be done prior to MQT population.

Populating MQTs

Population of the MQT data is a time- and resource-intensive exercise due to the fact that creation of MQTs normally requires the processing of all the data in the base tables and aggregating column data over 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 improve MQT population performance

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 analyzing the MQT's query:

- Create radix and/or encoded vector indexes for any local selection columns.
- Create radix indexes for the join columns.
- Create radix indexes for the grouping columns.

These guidelines follow the index recommendations covered in the *Indexing and Statistics Strategies white paper*.

For the following MQT definition:

```
CREATE TABLE Example_MQT AS
(SELECT Geography, Region, Year, Month, SUM(Revenue) AS Total_Revenue,
      SUM(Quantity) AS Total_Quantity, COUNT(*) AS Rows_per_Group
 FROM Example_Transaction_Table
 GROUP BY Geography, Region, Year, Month)
DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER
```

Build the following index to provide the query optimizer with statistics on the grouping columns and an index to perform the aggregation, if needed.

```
CREATE INDEX Example_Transaction_Table_IX1
ON Example_Transaction_Table (Geography, Region, Year, Month)
```

For the following MQT definition:

```
CREATE TABLE Sales_Part_MQT AS
(SELECT p.Category, p.Department, SUM(s.Sales) AS Total_Sales,
      SUM(s.Quantity) AS Total_Quantity, COUNT(*) as Rows_per_Group
 FROM Sales_Fact s INNER JOIN Part_Dim p
      ON s.PartKey = p.PartKey
 GROUP BY p.Category, p.Department)
DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER
```

Build the following index to provide the query optimizer with statistics on the join and grouping columns and provides indexes for the runtime implementation, if needed.

```
CREATE INDEX Sales_Fact_IX1 ON Sales_Fact (Partkey);

CREATE INDEX Part_Dim_IX1 ON Part_Dim (Partkey);

CREATE INDEX Part_Dim_IX2 ON Part_Dim (Category, Department);
```

Environment to support MQT population performance

Providing a proper runtime environment is also important when aggregating large data sets as fast as possible. If minimizing 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 resource as possible and a large memory pool. The MQT-population process must be allowed to run an isolated process, if possible. Setting the parallel degree to *MAX for the job that aggregates the data and populates the MQT allows the query optimizer to consider all the memory in the job's pool. During run time, the database engine can then be as aggressive as possible with the I/O tasks. The *MAX parallel degree also allows the query optimizer to consider running the MQT query with symmetric multiprocessing (SMP), provided that the Db2 SMP feature is installed on the system or logical partition (LPAR). To allow the query to complete sooner, SMP trades resources for time (for example, more resources are used in a given unit of time).

Cascading MQT population

When creating multiple MQTs for a given hierarchy (such as Year / Month / Day), the base tables might be continually queried. A faster 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 or most detailed data is created first (Year / Month / Day), followed by the next level (Year / Month), followed by the last level (Year) (see Figure 12).

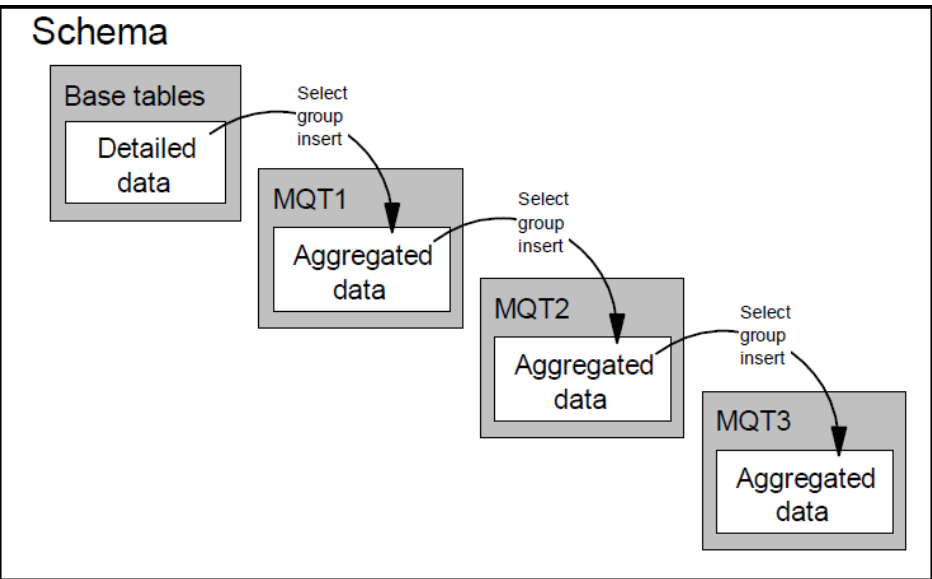


Figure 12: Cascading MQT Population

Given that MQT definitions cannot be based on other MQTs, the process of cascading the MQT creations involves some programmer intervention. That is to say, the MQT definitions must reference the base tables, but the data used to populate the MQT is from a previously created MQT.

Figure 13 shows the following general steps to create MQTs in a cascading fashion:

1. Create the initial MQT in the hierarchy and populate it from the base tables.
2. Create the next MQT in the hierarchy with the **DATA INITIALLY DEFERRED** attribute and populate this MQT from MQT created in step number 1.
3. Create the next MQT in the hierarchy with the **DATA INITIALLY DEFERRED** attribute and populate this MQT from the MQT created in number 2.

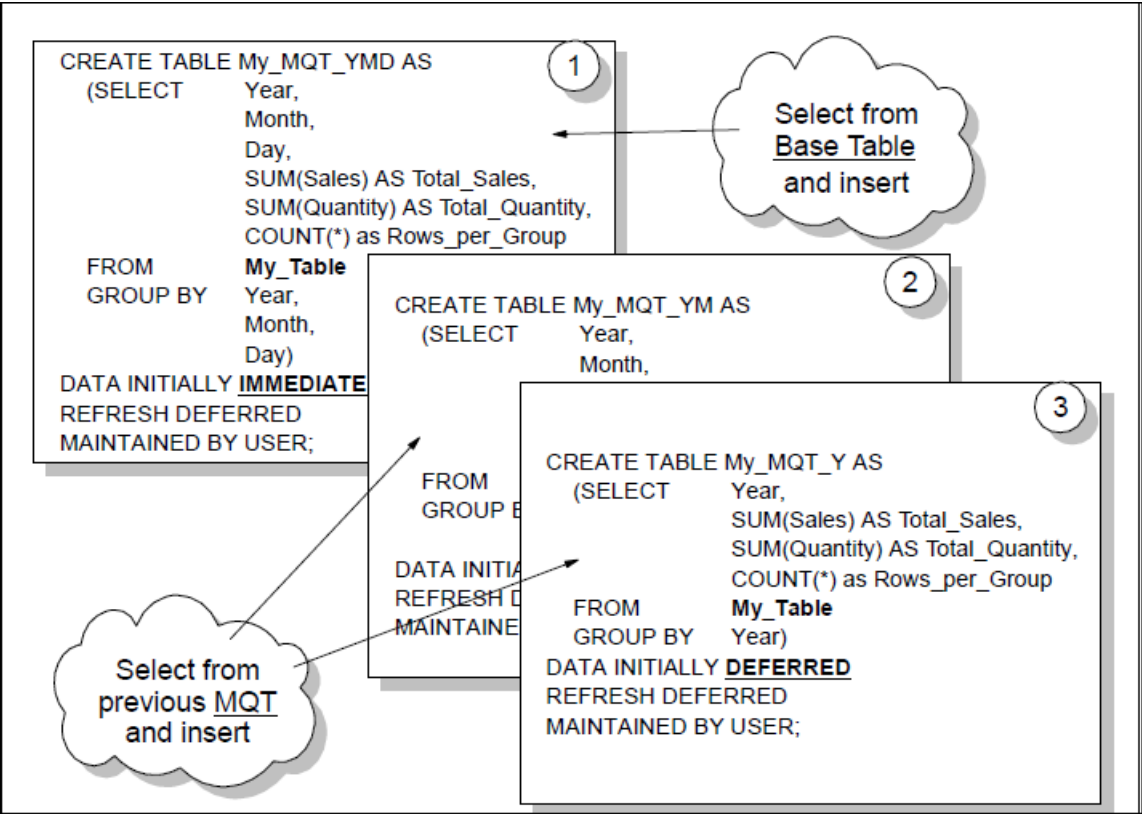


Figure 13: Steps to Populating an MQT in Cascading Fashion

For this process to be successful, all the MQT definitions in the hierarchy must reference and be based on the same underlying tables.

MQT aggregation methods and strategies

The query optimizer has two basic methods of aggregating data in the base tables:

- Grouping with an index
- Grouping with a hash table

Each method has its own requirements, characteristics and advantages. Understanding and anticipating the use of either method determines whether programmer intervention is required to speed up the MQT population.

The optimal use of either grouping strategy requires the optimizer to have a good understanding of the estimated selectivity of the query (normally 100%) 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 in the query job's pool to house the hash table. A large number of distinct groups results in a larger hash table, and a larger hash table requires a larger memory pool to perform efficiently. If the optimizer expects the job's fair share of 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 group the data in parallel with SMP is available. This feature allows grouping queries to perform faster by using more resources.

Index grouping is the preferred strategy when hash grouping is not viable. The memory footprint of using an index can be much smaller than housing 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 indexed list. This adds more time to the query execution.

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

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

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

The MQT creation and population query can be explained only by using the ACS Run SQL Scripts utility. This allows the query optimizer's feedback to be analyzed without actually running the query.

Figure 14 shows an example of grouping with a permanent index as presented by Visual Explain:

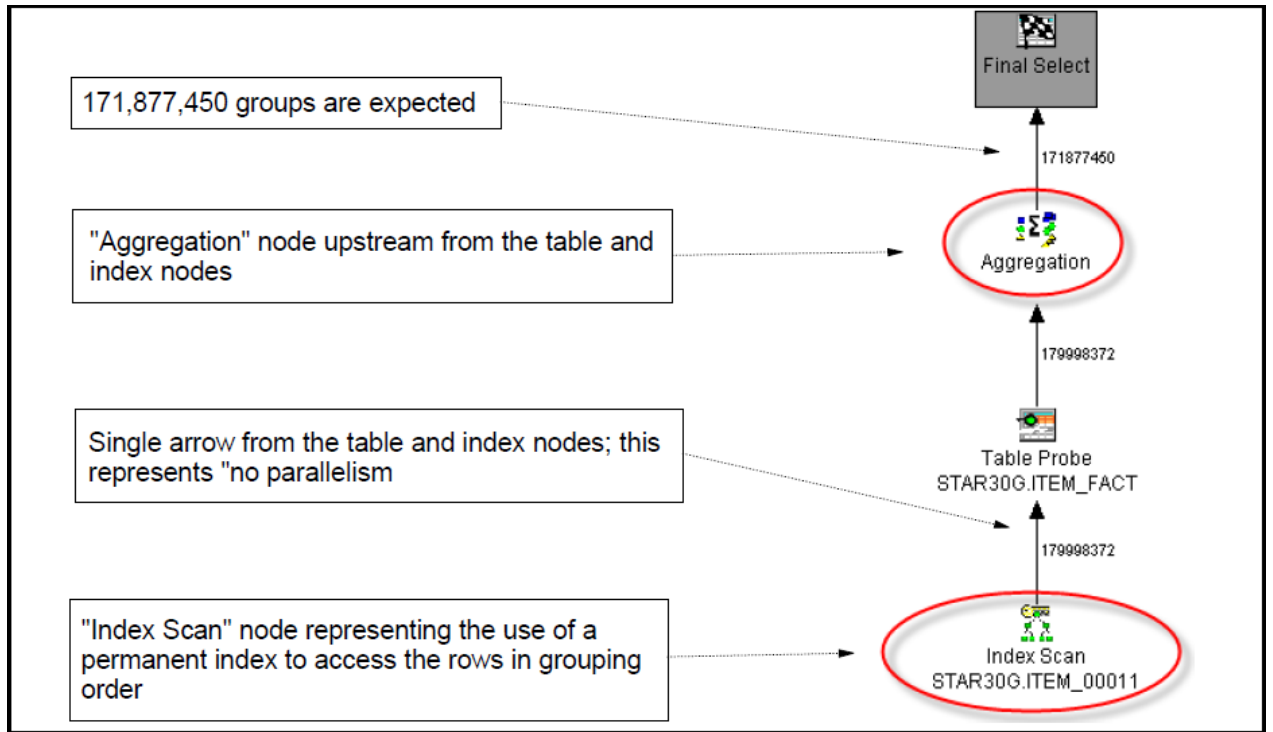


Figure 14: Grouping with a permanent index

Figure 15 shows an example of grouping through a temporary hash table with SMP parallel processing, as drawn with Visual Explain:

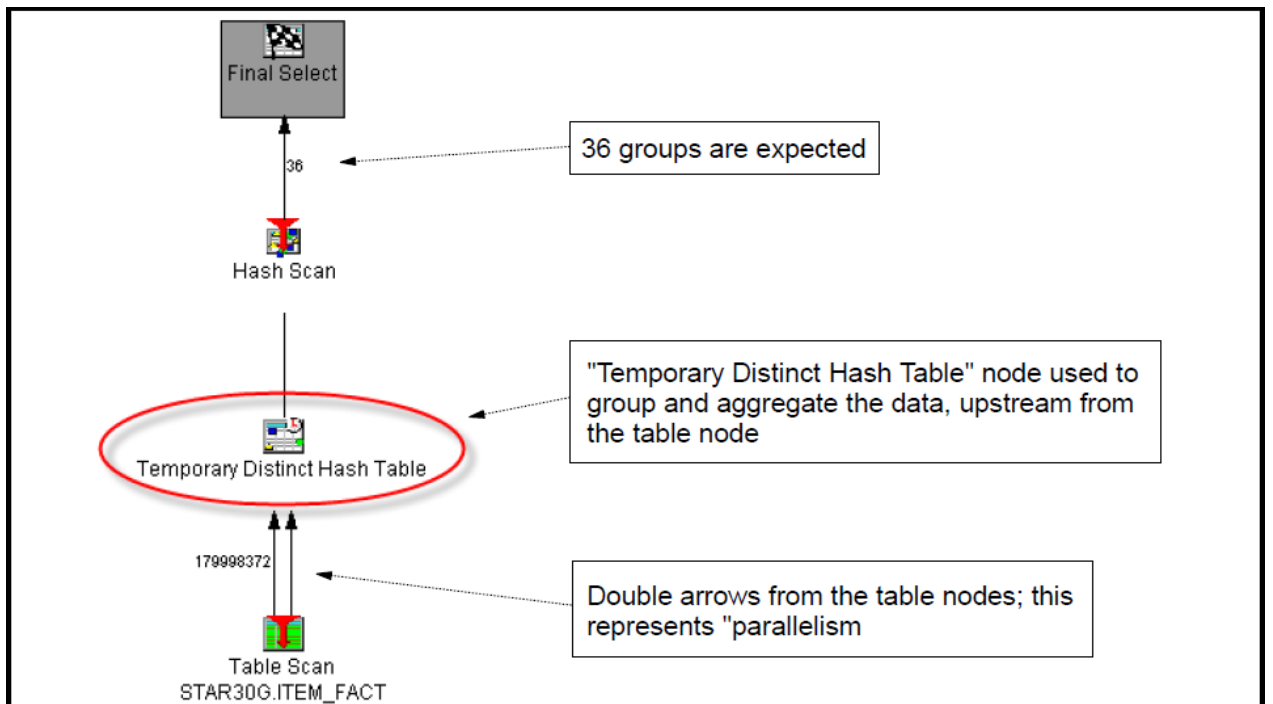


Figure 15: Grouping with a hash table

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 list is used to facilitate the grouping and aggregation of data.

MQT population: Programmer intervention

In cases where the MQT creation query uses index grouping, or the result set (such as groups) that is to be placed into the MQT is large, some 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 fan out of rows, some programmer intervention might be helpful in speeding up the join process (see Figure 16).

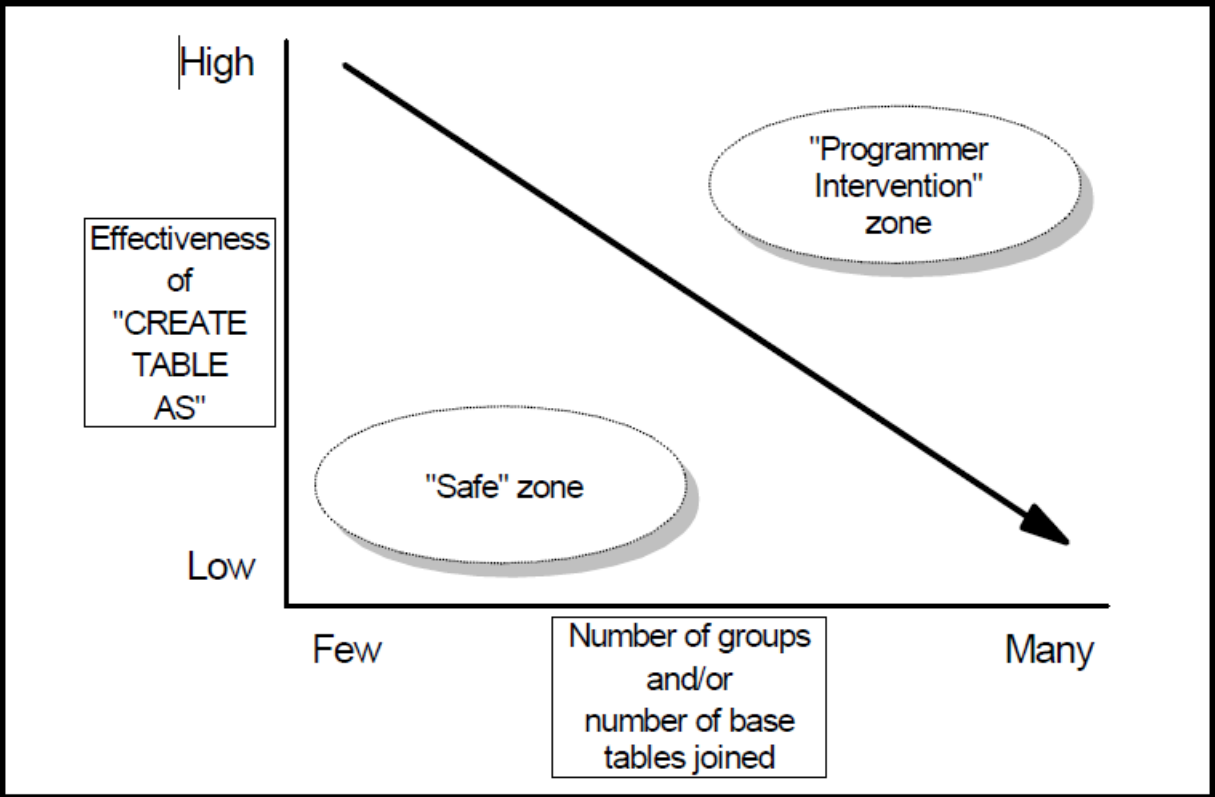


Figure 16: Number of groups and/or number of tables joined

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

The key to faster joining, aggregation and insertion is parallel processing. When the database engine is unable to employ SMP implicitly, the programmer can design and implement a parallel process. This process consists of breaking the data from the base tables and putting it into logical ranges, selecting and processing each range in parallel, and inserting the aggregated data into the MQT in parallel (see Figure 17).

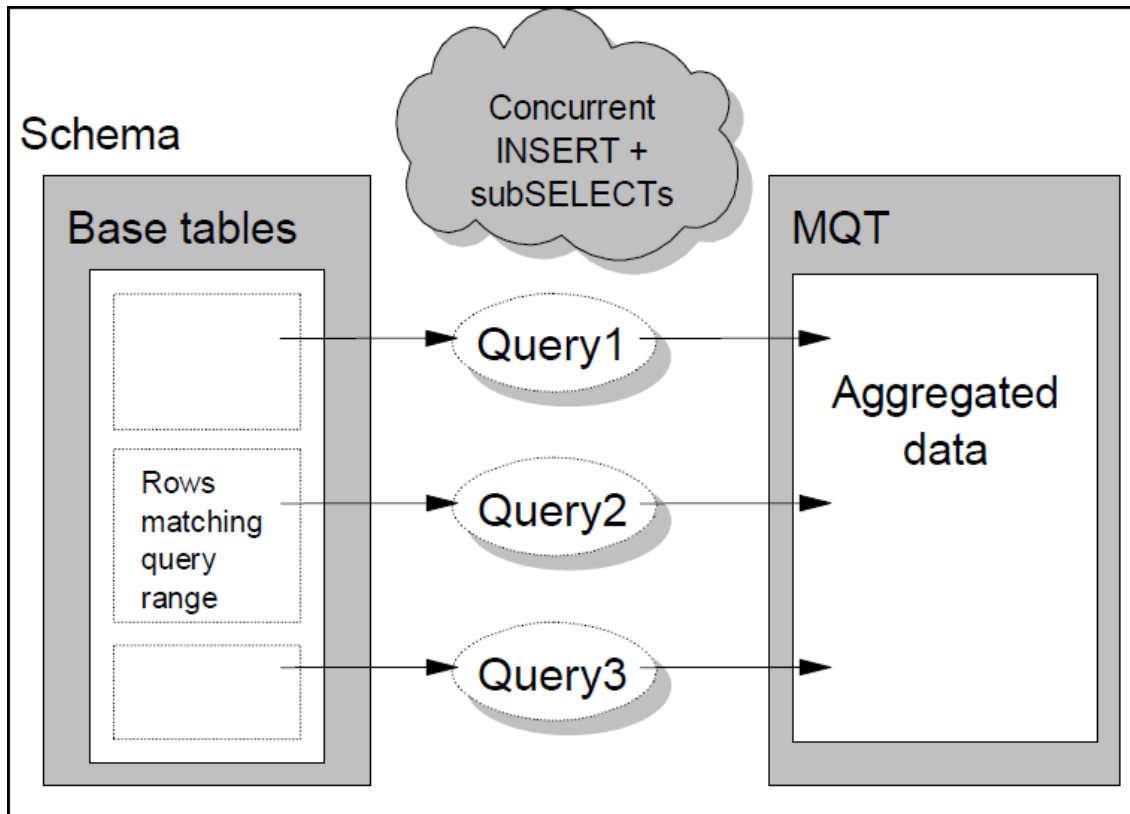


Figure 17: Number of groups and/or number of tables joined

The design process starts with profiling the data represented by the first one or two grouping columns. Identify the distinct ranges of data within the first one or two grouping columns and compare this number to the number of processors available during query execution. Either the number of processors or the number of ranges determines the level of parallelism to employ. The goal is to have all the processors as busy as possible, thus maximizing throughput and minimizing the time to populate the MQT. Using all the processing resources to populate an MQT 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 where there are three years, and thousands of customers represented in the data, three degrees of parallelism can be used. The parallel grouping queries can each select and process one of three years. On a system with three or more available processors, each grouping query runs on one processor, in parallel. If more resources are available and a larger degree is desirable, then additional grouping queries can be used, each processing a given year and separate range of customers.

When running a set of queries at the same time, the Db2 SMP parallel degree must be set to *NONE. Furthermore, ensure that all the data ranges represented in the base tables are accounted for. It is easy to omit a range, resulting in incomplete data.

Employing a parallel population process

Here is an example of creating and populating an MQT with a parallel process. First, you create the MQT with no data using the DATA INITIALLY DEFERRED attribute.

```
CREATE TABLE Year_Customer_MQT AS
(SELECT Year, Customer, SUM(Revenue) AS Total_Revenue,
      SUM(Quantity) AS Total_Quantity, COUNT(*) AS Rows_per_Group
 FROM Transaction_Table
 GROUP BY Year, Customer)
DATA INITIALLY DEFERRED
REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY USER
```

Depending on the processing resources available, define and run a set of parallel queries. Each query needs to select a distinct range of rows, aggregate the data and insert the results into the MQT.

```
INSERT INTO Year_Customer_MQT
(SELECT Year, Customer, SUM(Revenue) AS Total_Revenue,
      SUM(Quantity) AS Total_Quantity, COUNT(*) AS Rows_per_Group
 FROM Transaction_Table
 WHERE Year = 2020
 GROUP BY Year, Customer);

INSERT INTO Year_Customer_MQT
(SELECT Year, Customer, SUM(Revenue) AS Total_Revenue,
      SUM(Quantity) AS Total_Quantity, COUNT(*) AS Rows_per_Group
 FROM Transaction_Table
 WHERE Year = 2021
 GROUP BY Year, Customer);

INSERT INTO Year_Customer_MQT
(SELECT Year, Customer, SUM(Revenue) AS Total_Revenue,
      SUM(Quantity) AS Total_Quantity, COUNT(*) AS Rows_per_Group
 FROM Transaction_Table
 WHERE Year = 2022
 GROUP BY Year, Customer);
```

Given that the individual queries select subsets of data from the base data, be sure to provide proper indexes over the local selection columns. This provides the query optimizer and database engine greater flexibility. Providing both radix and encoded vector indexes can be advantageous. Using the previous example, the proper indexes to provide are as follows:

```
CREATE INDEX Transaction_Table_IX1
ON Transaction_Table (Year, Customer);

CREATE ENCODED VECTOR INDEX Tansaction_Table_IDX2
ON Transaction_Table (Year);
```

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

Testing and tuning MQTs

If the query optimizer rewrites the user query to access an MQT instead of the base tables, the same basic methods and strategies are employed to access and process the MQT. Specifically, the MQTs are scanned and probed for local selection, joining, grouping and ordering. It is important to test and tune the MQTs 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 MQTs is essentially the same as the base tables. That is to say, with indexes on the MQTs, the query optimizer has the necessary statistics and has many choices when implementing the query request.

Column statistics used by SQE are collected and stored on a table by table basis, and this includes MQTs. If SQE automatically collects a column statistic for an MQT, it is a good practice to identify those columns periodically and to ensure that the statistics are updated after recreating or refreshing the MQT. This minimizes poor query performance immediately after the MQTs are repopulated.

Given that the MQT 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 data model and test the queries with the MQTs 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 MQT has many rows.

The *Indexing and Statistics Strategies white paper* can be referenced for more details on this topic.

Enabling MQT Support

The automatic consideration and use of MQTs by the query optimizer must be explicitly enabled. The first explicit enablement action requires the MQT to be created with the ENABLE QUERY OPTIMIZATION attribute. In addition, there are two query options, MATERIALIZED_QUERY_TABLE_USAGE and MATERIALIZED_QUERY_TABLE_REFRESH_AGE, that must be enabled with a QAQQINI file.

The MATERIALIZED_QUERY_TABLE_USAGE QAQQINI option controls the optimizer's recognition and use of MQTs with the following values.

- *DEFAULT - The default value is *NONE.
- *NONE - MQTs are not used in query optimization.
- *ALL - The user-maintained, refresh-deferred query tables can be used.
- *USER - The user-maintained MQTs can be used.

The MATERIALIZED_QUERY_TABLE_REFRESH_AGE QAQQINI option determines which MQTs are eligible to be used based on their age (latency) with the following values. This option value is compared against the last time a REFRESH TABLE statement was done.

- *DEFAULT - The default value is 0, no MQTs can be used.
- *ANY - Any MQTs that are eligible are according to the MATERIALIZED_QUERY_TABLE_USAGE option can be used. If an MQT has never been refreshed by the REFRESH TABLE statement, but the MQT should be considered, then this option must be set to *ANY.

- Timestamp duration - Only tables indicated by the MATERIALIZED_QUERY_TABLE_USAGE option that have a refresh performed within the specified timestamp duration can be used. This value is a DECIMAL(20,6) number that indicates a timestamp duration since the last REFRESH TABLE was performed on the MQT..

Furthermore, there are query runtime environment settings that affect the query optimizer's ability to use MQTs in the implementation. The complete list of query runtime environment settings is documented in the *Database Performance and Query Optimization* guide (ibm.biz/db2iBooks). Here are some of the more notable attributes that must be true in the environment running the query:

- The query must specify ALWCPYDTA(*OPTIMIZE) or INSENSITIVE cursor.
- The query must not be a SENSITIVE cursor.
- The base table to be replaced with an MQT must not be update- or delete-capable within the query.

Feedback on MQT usage

Feedback from the query optimizer can be used to determine whether the optimizer has rewritten the user query to use an MQT. This same optimizer feedback can be used to tune the access and processing of MQT in a query. For example, using feedback from the query optimizer to identify new indexes that would speed up the retrieval of data from an MQT.

Feedback on MQT optimization and usage is provided through ACS through the use of Visual Explain and an SQL Performance Monitor (database monitor).

Visual Explain provides an MQT highlighter option that makes it easy to identify if the optimizer used an MQT in the query plan implementation. Figure 18 shows how simple it is to activate the MQT highlighter from the View pull-down menu.

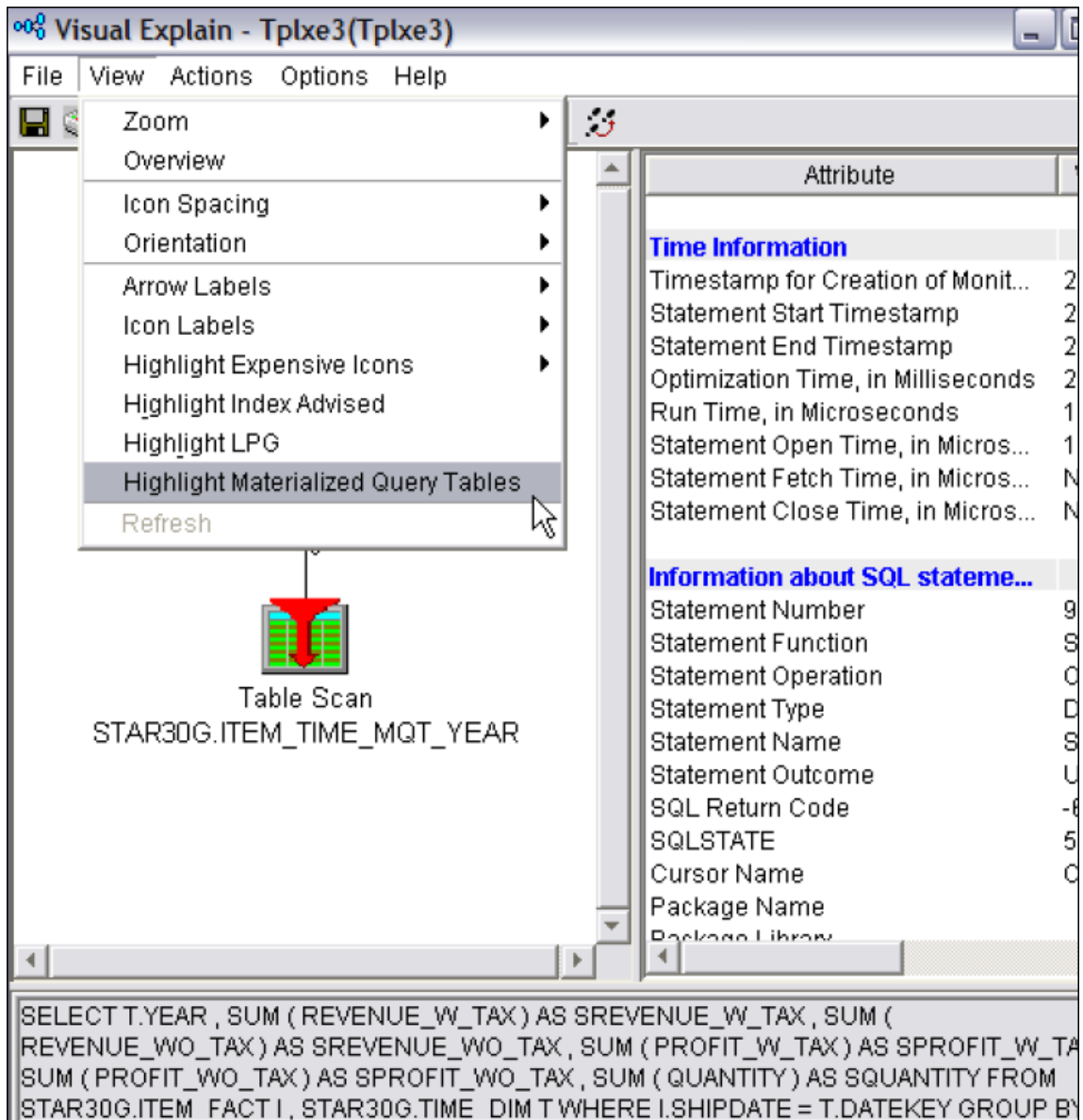


Figure 18: Enabling Visual Explain MQT highlighter

With the MQT highlighting option selected, Visual Explain will augment any MQTs in the query plan with a colored background as shown in Figure 19.

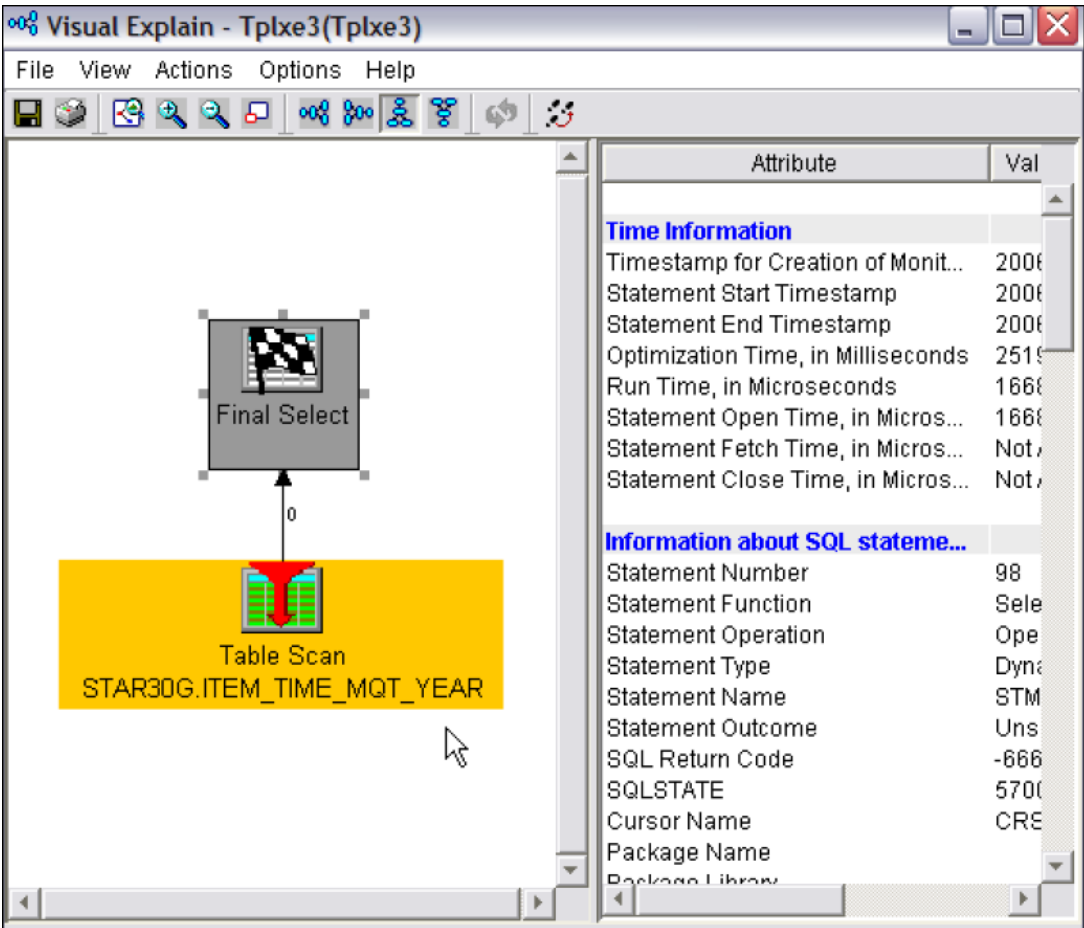


Figure 19: Visual Explain example of highlighted MQT

An SQL Performance Monitor collection can be reviewed for MQT usage using the Analyze function in the ACS SQL Performance Center as shown in Figure 20. After selecting this option, a new reporting window will be displayed which features a report that will display a list of the SQL statements which used an MQT in their implementation.

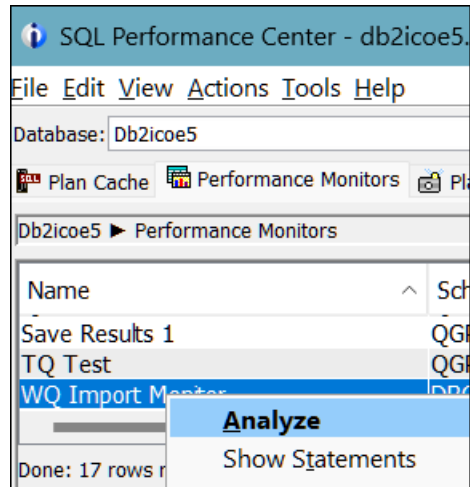


Figure 20: SQL Performance Monitor Analysis Reports

The ACS Schemas tool also provides the ability to list all the MQTs on a given table and evaluate the usage of the MQTs. For example, it is now possible to determine when a given MQT is used by the optimizer and how many times it is used. Conversely, it is possible to determine that a given MQT has never been used.

To show the MQTs that are based on a particular table, right-click on that specific table and select the Work With → Materialized Query Tables option as shown in Figure 21.

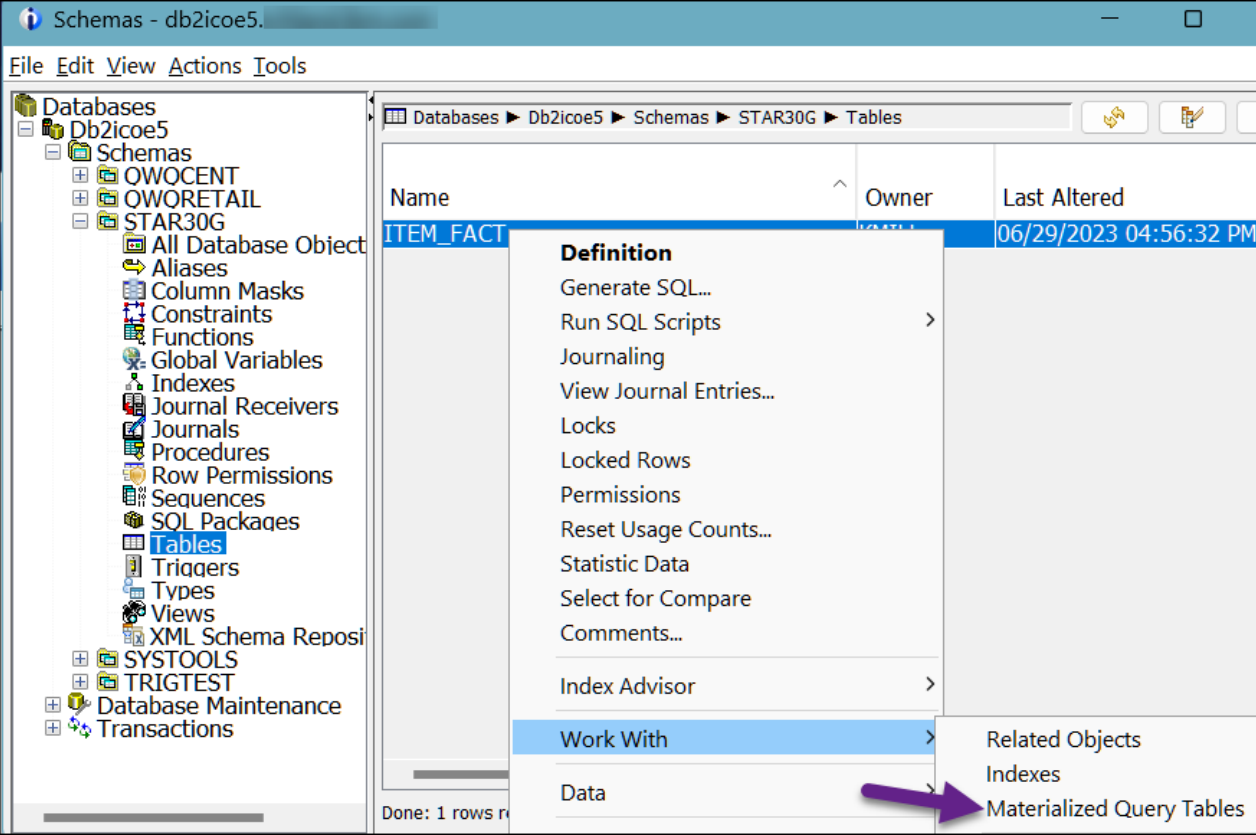


Figure 21: ACS Schemas Work With MQT function

The new window that is displayed can be used to verify which, if any, MQTs are based on this table. The report shows descriptive information, such as the following (see Figure 22):

- Name of the MQT
- Short system name of the MQT
- MQT creation date and time
- MQT enablement (Yes or No) for optimization

SQL Name	Schema	Partition	Owner	Short Name	Enabled	Creation
ITEM_TIME_CUST_MQT_YEAR_CONTINENT	STAR30G	ITEM_00031	QDFTOWN	ITEM_00031	Yes	1/12/0
ITEM_TIME_CUST_MQT_YEAR_CONTINENT_COUNTRY	STAR30G	ITEM_00032	QDFTOWN	ITEM_00032	Yes	1/12/0
ITEM_TIME_CUST_MQT_YEAR_CONTINENT_COUNTRY_REGION	STAR30G	ITEM_00035	QDFTOWN	ITEM_00035	Yes	1/13/0
ITEM_TIME_CUST_MQT_YEAR_CONTINENT_COUNTRY_REGION_TERRITORY	STAR30G	ITEM_00038	QDFTOWN	ITEM_00038	Yes	1/13/0
ITEM_TIME_CUST_MQT_YEAR_CUSTKEY	STAR30G	ITEM_00033	QDFTOWN	ITEM_00033	Yes	1/13/0
ITEM_TIME_CUST_MQT_YEAR_QUARTER_CUSTKEY	STAR30G	ITEM_00036	QDFTOWN	ITEM_00036	Yes	1/13/0
ITEM_TIME_CUST_MQT_YEAR_QUARTER_MONTH_CUSTKEY	STAR30G	ITEM_00039	QDFTOWN	ITEM_00039	Yes	1/13/0
ITEM_TIME_CUST_MQT_YEAR_QUARTER_MONTH_SUPPKEY	STAR30G	ITEM_00040	QDFTOWN	ITEM_00040	Yes	1/13/0
ITEM_TIME_CUST_MQT_YEAR_QUARTER_MONTH_WEEK_CUSTKEY	STAR30G	ITEM_00041	QDFTOWN	ITEM_00041	Yes	1/14/0
ITEM_TIME_CUST_MQT_YEAR_QUARTER_MONTH_WEEK_SUPPKEY	STAR30G	ITEM_00042	QDFTOWN	ITEM_00042	Yes	1/14/0
ITEM_TIME_CUST_MQT_YEAR_QUARTER_SUPPKEY	STAR30G	ITEM_00037	QDFTOWN	ITEM_00037	Yes	1/13/0
ITEM_TIME_CUST_MQT_YEAR_SUPPKEY	STAR30G	ITEM_00034	QDFTOWN	ITEM_00034	Yes	1/13/0
ITEM_TIME_MQT	STAR30G	ITEM_00020	MCKINLEY	ITEM_00020	Yes	3/25/0
ITEM_TIME_MQT_YEAR	STAR30G	ITEM_00027	QDFTOWN	ITEM_00027	Yes	1/12/0
ITEM_TIME_MQT_YEAR_QUARTER	STAR30G	ITEM_00028	QDFTOWN	ITEM_00028	Yes	1/12/0
ITEM_TIME_MQT_YEAR_QUARTER_MONTH	STAR30G	ITEM_00029	QDFTOWN	ITEM_00029	Yes	1/12/0
ITEM_TIME_MQT_YEAR_QUARTER_MONTH_WEEK	STAR30G	ITEM_00030	QDFTOWN	ITEM_00030	Yes	1/12/0

Figure 22: ACS Related MQT Report

Scrolling (to the right of the report) shows information on when the MQT was used (see Figure 23).

	Last Refresh Date	Last Query Use	Last Query Statistics Use	Query Use Count	Query Statistics Use Count	Last Used Date
1 ...	1/12/05 11:41:14 PM			0	0	
5 ...	1/13/05 1:07:53 AM			0	0	
AM	1/13/05 11:03:09 AM			0	0	
PM	1/13/05 9:26:24 PM			0	0	
AM	1/13/05 2:39:03 AM			0	0	
4 ...	1/13/05 12:43:35 PM			0	0	
PM	1/13/05 11:19:07 PM			0	0	
8 ...	1/14/05 12:52:52 AM			0	0	
3 ...	1/14/05 3:09:08 AM			0	0	
AM	1/14/05 4:54:51 AM			0	0	
7 ...	1/13/05 2:15:33 PM			0	0	
AM	1/13/05 4:07:01 AM			0	0	
AM	3/25/05 11:14:01 AM			0	0	
PM	1/12/05 7:48:13 PM	7/28/06 10:17:50 AM		1	0	7/28/06 9:49:40 AM
PM	1/12/05 8:36:59 PM			0	0	
PM	1/12/05 9:25:52 PM			0	0	
PM	1/12/05 10:14:40 PM			0	0	

Figure 23: ACS Schemas MQT Report Scrolled to the Right

The **Last Query Use** column shows the timestamp when the MQT was last used by the optimizer to replace user-specified tables in a query. The **Query Use Count** column shows the number of instances the MQT was used by the optimizer to replace user-specified tables in a query.

The **Last Query Statistics Use** and **Query Statistics Use Count** columns are currently not relevant, nor populated. The query optimizer does not use MQTs for statistics.

Designing MQT refresh strategies

Db2 for i 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 MQTs 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 like the following:

```
REFRESH TABLE Example_MQT
```

Be aware that this REFRESH TABLE statement initiates a full refresh of the MQT and causes the following:

- All indexes on the MQT are removed.
- All the rows in the MQT are deleted.
- The underlying MQT query is run to repopulate the data from the base tables.
- All 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. If the business requires reporting against a segment of data, an MQT 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 and monthly view, the MQT that contains the aggregated data (representing Year / Month) and is refreshed as part of the month-end processing.

BI and DW are 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 ETL process runs on a daily basis (for example, end-of-day processing), the MQT refresh or maintenance strategy can also occur on a daily basis. Refreshing the entire MQT data set to incorporate only one day's worth of data might be inefficient. In this case, the MQT can be maintained (not refreshed) by calculating, then inserting or updating rows with the new daily aggregates. Any MQTs that are based on hierarchies can be refreshed or maintained using the same data, or they can be fully refreshed using the cascading method discussed earlier. SQL 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 the REFRESH TABLE statement to perform a full refresh of the MQT, it is a good practice to use the following steps:

1. Document any column statistics on the MQT.
2. Set the isolation level (commitment control level) to *NONE.
3. Drop any indexes on the MQT.
4. Delete all the rows currently in the MQT.
5. Calculate the aggregates and populate the MQT.
6. Recreate the indexes that previously existed on the MQT.
7. Refresh any column statistics on the MQT.

It is important to test and verify the MQT refresh or maintenance process before using that process 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 applied. For example, a full-refresh strategy might be too time-consuming, but an incremental maintenance strategy is acceptable. When implementing a maintenance strategy based on immediate changes to the underlying base tables, be sure to measure and understand any additional time and resources required. In other words, the realtime maintenance of MQTs increases the time necessary for the insert, update and delete operations on the base tables.

Using the SQL Performance Monitors and Collection Services tools can assist you with understanding the increased workload and resource utilization of the refresh and maintenance process. You can get the most out of the available resources or identify overcommitted resources. For example, when running refresh processing in parallel, if processor resources are underused, a higher degree of parallelism can be tried 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. All of this assumes a balanced configuration 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 requests are issued. 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 SQL Plan Cache Snapshot and SQL Performance Monitor. Also, gather some baseline information on how the system resources are used; Collection Services is a good instrument for this. After implementing MQTs, the baseline information can be used to quantify any differences in behavior or performance of the application.

Seriously consider running a benchmark to test any MQT creation, refresh and usage strategies. A great place to run a benchmark is the IBM i Performance and Scalability Center in Rochester, Minnesota. Understanding the costs and benefits of MQTs before deploying to a production environment is a critical success factor. You can find more information about the IBM i Performance and Scalability Center by emailing systems-expert-labs@ibm.com.

Summary

With the latest version of Db2 for i, IBM continues to deliver additional features and functionality to assist with the implementation of robust, data-centric applications. The ability to create and use MQT provides yet another option for high-performance query processing. This paper, along with the aforementioned publications, provides some guidance and insight on using this new feature.

Appendix A provides a Web site listing for the latest information regarding Db2 for i support of MQT. You can find additional information and insight on IBM i information and data management at:

Db2IBMi.blogspot.com

Appendix A: Resources

The following websites provide useful references to supplement the information contained in this paper:

- Db2 for i Manuals & IBM i documentation
ibm.biz/db2iBooks
- Indexing and statistics strategy white paper
ibm.biz/db2iPapers
- Db2 for i Expert Labs Database Engineer Enablement and Skills Transfer
ibm.biz/Db2iExpertLabs
- Db2 for i Blog
Db2IBMi.blogspot.com

About the authors

Kent Milligan is a Senior Db2 for i Consultant in IBM Technology Expert Labs. Kent has over 25 years of experience as a Db2 for IBM i consultant and developer working out of the IBM Rochester lab. Prior to re-joining the Expert Labs practice in 2020, Kent spent 5 years working on healthcare solutions powered by IBM Watson technologies. Kent is a sought-after speaker and author on Db2 for i & SQL topics. You can reach Kent with your questions regarding partitioned table support or any Db2 for i topic at: kmill@us.ibm.com.

Mike Cain was a Senior Technical Staff Member and the team leader of the Db2 for i Center of Excellence in Rochester, Minnesota. Before his last position, he worked as an IBM systems engineer and technical consultant for the IBM i platform.

Trademarks and special notices

© Copyright IBM Corporation 2023.

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

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.

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

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

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Photographs shown are of engineering prototypes. Changes may be incorporated in production models.



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