Using Materialized Query Tables to Speed up Queries in DB2 UDB

Alexander Kuznetsov
August 22, 2002

Introduction

Sometimes a simple change in physical database structure dramatically improves query performance. In addition to indexes, DB2® Universal Database™ provides you with materialized query tables (known as "summary tables" in Version 7.2 and earlier releases), which in many cases are more efficient than indexes. In essence, a materialized query table (MQT) is a table that is defined based on the result of a query. This article describes some examples in which MQTs provide more effective performance improvements over the use of indexes alone.

Advantage: Avoid repetitious calculations

MQTs can help you avoid having to repeat calculations, such as SUMs, for each query. Let's assume there is a table called CUSTOMER_ORDER that stores customer orders for several years. The table has over one million records, with an average row width of 400 bytes. Now, assume that we must run multiple queries on orders for year 2001 and that we just need three columns from the table, such as:

```
select SUM(AMOUNT), trans_dt
from db2inst2.CUSTOMER_ORDER
where trans_dt between '1/1/2001' and '12/31/2001'
group by trans_dt
```

Or

```
select SUM(AMOUNT), status
from db2inst2.CUSTOMER_ORDER
where trans_dt between '1/1/2001' and '12/31/2001'
group by status
```
If there are appropriate indexes, these queries are executed as index scans. Listing 1 is an excerpt from an execution plan, indicating that the estimated cost of running the query with the index scan is 152455.

Listing 1. Cost of running query on CUSTOMER_ORDER table

```
-------------------- SECTION ---------------------------------------
Section = 1

SQL Statement:

```sql
select SUM(AMOUNT), trans_dt
from db2inst2.CUSTOMER_ORDER
where trans_dt between '1/1/2001' and '12/31/2001'
group by trans_dt
```

Estimated Cost = 152455
Estimated Cardinality = 378

(some lines are omitted here)

Subsection #2:
```
Access Table Name = DB2INST2.CUSTOMER_ORDER ID = 2,591
| #Columns = 1
| Index Scan: Name = DB2INST2.CUST_ORD_TRANS_DT ID = 4
| | Index Columns:
| | | 1: TRANS_DT (Ascending)
```

(some lines are omitted here)

End of section

Let’s now create a MQT that contains the columns and rows that we need, including the calculation for the grand total:

```
CREATE TABLE DB2INST2.SUMMARY_CUSTOMER_ORDER_2001 AS
(SELECT SUM(AMOUNT) AS TOTAL_SUM,
TRANS_DT,
STATUS
FROM DB2INST2.CUSTOMER_ORDER
WHERE TRANS_DT BETWEEN '1/1/2001' AND '12/31/2001'
group by TRANS_DT,
STATUS)
DATA INITIALLY DEFERRED REFRESH DEFERRED;
```

The clause DATA INITIALLY DEFERRED means that data is not inserted into the table as part of the CREATE TABLE statement. Instead, you have to do a REFRESH TABLE statement to populate the table. The clause REFRESH DEFERRED means that the data in the table only reflects the result of the query as a snapshot at the time you issue the REFRESH TABLE statement.

When we are ready to populate the MQT we have just created, we issue the following statement:

```
REFRESH TABLE DB2INST2.SUMMARY_CUSTOMER_ORDER_2001;
```
Now the queries against the MQT run much faster, because the MQT is quite small in size and its rows are short (just 45 bytes, as opposed to 400 bytes in the base table). An excerpt from the execution plan generated by dynexpln shown in Listing 2 indicates a dramatic performance improvement with an estimated cost of 101 versus 152455 in the previous plan.

Listing 2. Running a query on MQT reduces cost

```
-------------------- SECTION ---------------------------------------
Section = 1

SQL Statement:

    select sum(total_sum), trans_dt
    from db2inst2.summary_customer_order_2001
    where trans_dt between '1/1/2001' and '12/31/2001'
    group by trans_dt

Estimated Cost        = 101
Estimated Cardinality = 25

Subsection #1:
    Access Summary Table Name = DB2INST2.SUMMARY_CUSTOMER_ORDER_2001  ID = 2,44
    #Columns = 2
    Relation Scan
    Sortheap Allocation Parameters:
    #Rows     = 21
    Row Width = 45
    Piped

(All lines are omitted here)
```

Note that if the data in CUSTOMER_ORDER for the year 2001 gets updated after the refresh, you need to refresh the MQT again.

**Advantage: Avoid resource-intensive scans**

Let's assume we frequently need up-to-date totals for the year 2002. The reports, which used to run very quickly on January 3, 2002, run more slowly in May as the amount of data for the year 2002 increases. As we described previously, the queries are executed as index scans on the CUSTOMER_ORDER table.

Now we want to consider how MQTs might help us improve performance. However, because the data is being updated all the time and we need current data, we cannot use REFRESH DEFERRED, because the MQT will get out of sync with the base table the next time the base table is updated. Instead, let's create a MQT with REFRESH IMMEDIATE and ENABLE QUERY OPTIMIZATION options.
CREATE TABLE DB2INST2.SUMMARY_CUSTOMER_ORDER_2002 AS(
    SELECT
        TRANS_DT,
        STATUS,
        COUNT(*) AS COUNT_ALL,
        SUM(AMOUNT) AS SUM_AMOUNT,
        COUNT(AMOUNT) AS COUNT_AMOUNT
    FROM DB2INST2.CUSTOMER_ORDER
    GROUP BY TRANS_DT,
        STATUS
) DATA INITIALLY DEFERRED
REFRESH IMMEDIATE
ENABLE QUERY OPTIMIZATION;

REFRESH IMMEDIATE means that the contents of the MQT are always up-to-date after you have populated the MQT by the REFRESH TABLE statement.

**Important:** To enable the optimizer to choose the MQT automatically, the ENABLE QUERY OPTIMIZATION must be in effect (it is the default).

**Other syntax notes:** Although though our business is interested only in SUM(AMOUNT), we still have to include COUNT(*) and COUNT(AMOUNT) in the fullselect. The reason is simple to remember. Let's suppose all the records for a given date are being deleted from the base table:

DELETE FROM DB2INST2.CUSTOMER_ORDER WHERE TRANS_DT = '1/1/2002';

Now DB2 must detect that all the records for a particular date are gone and to delete all the corresponding records in the MQT. Having the COUNT field allows DB2 to do it quickly, without having to scan either the table or its index. COUNT(AMOUNT) is required only when the AMOUNT column is nullable.

Now it's time to populate our MQT and refresh its statistics:

REFRESH TABLE DB2INST2.SUMMARY_CUSTOMER_ORDER_2002;
RUNSTATS ON TABLE DB2INST2.SUMMARY_CUSTOMER_ORDER_2002 WITH DISTRIBUTION;

Now let's see how the query performance is improved (Estimated Cost = 392). **Listing 3** is an excerpt from the query execution plan:

Listing 3. Optimizer chooses to use MQT
Using Materialized QueryTables to Speed up Queries in DB2 UDB

--- SECTION ---

SQL Statement:

```sql
select SUM(AMOUNT), trans_dt
from db2inst2.customer_order
where trans_dt >= '1/1/2002'
group by trans_dt
```

Estimated Cost = 392
Estimated Cardinality = 268

Subsection #1:
Access Summary Table Name = DB2INST2.SUMMARY_CUSTOMER_ORDER_2002  ID = 2,46
| #Columns = 2
| Relation Scan

Note that the table CUSTOMER_ORDER, not the summary one, was specified in the query. The optimizer has automatically chosen to use the MQT.

Whenever the CUSTOMER_ORDER table is modified, an exclusive table lock may be held on SUMMARY_CUSTOMER_ORDER_2002 until the end of the transaction. That is true only for MQTs with both aggregate functions and the REFRESH IMMEDIATE option. So the transactions modifying relevant fields in CUSTOMER_ORDER (including all inserts and deletes) must be kept very short to reduce lock contention. This issue does not apply to MQT with the REFRESH DEFERRED option, nor to replicated MQTs (described in the next section).

**Advantage: Avoid broadcasts by using replicated MQTs**

Let's assume there is a big table named CUSTOMER_DATA in a partitioned environment. The table CUSTOMER_DATA is collocated with its child tables. The partitioning key is a system-generated integer CUSTOMER_ID. The table CUSTOMER_DATA has a reference to another one, named ZIP_CODE. The tables CUSTOMER_DATA and ZIP_CODE are not collocated. However, these two tables are very frequently joined. Let's have a look at the access plan shown in Listing 4.

Listing 4. Joins with ZIP_CODE cause cross-node broadcasts

--- SECTION ---

SQL Statement:

```sql
select c.*, z.zip, z.state_name, z.country_name
from db2inst2.customer_address c join db2inst2.zip_code z on
  c.zip_cd = z.zip_cd
```

Estimated Cost = 100975
Estimated Cardinality = 255819

Coordinator Subsection:
Distribute Subsection #2
| Broadcast to Node List
| | Nodes = 0, 1
The ZIP_CODE table is not updated often (there are not often new zip codes) and is quite frequently the target of a join. Every time the query is issued that causes the join, it is necessary to broadcast the ZIP_CODE table to every node.
This may be a good situation in which to use replicated MQTs, which is a based on a table that may have been created in a single-partition nodegroup, but that you want replicated across all of the database partitions in the nodegroup to enable collocation of frequently accessed data. To create the replicated MQT, invoke the CREATE TABLE statement with the REPLICATED keyword.

```sql
CREATE TABLE DB2INST2.SUMMARY_ZIP_CODE AS (SELECT * FROM DB2INST2.ZIP_CODE)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE ENABLE QUERY OPTIMIZATION REPLICATED;
```

Aggregates are not allowed in the definition. ZIP_CODE table has a unique index on ZIP_CD. Let's populate the table, create an index on it and update statistics:

```sql
REFRESH TABLE DB2INST2.SUMMARY_ZIP_CODE;
CREATE INDEX AAA_TTT ON DB2INST2.SUMMARY_ZIP_CODE(ZIP_CD);
RUNSTATS ON TABLE DB2INST2.SUMMARY_ZIP_CODE WITH DISTRIBUTION AND DETAILED INDEXES ALL;
```

Now the optimizer automatically chooses to use the replicated table, eliminating the need to broadcast ZIP_CODE table to every node every time the query is run:

Listing 5. By using replicated ZIP_CODE table, some cross-node broadcasts are avoided.

```
------------- SECTION -------------------------------
Section = 1

SQL Statement:
```
```sql
select c.*, z.zip, z.state_name, z.country_name
from db2inst2.customer_address c join db2inst2.zip_code z on
  c.zip_cd = z.zip_cd
```n```sql
```n
Estimated Cost = 101171
Estimated Cardinality = 255819

Coordinator Subsection:
  Distribute Subsection #1
    Broadcast to Node List
      Nodes = 0, 1
  Access Table Queue  ID = q1  #Columns = 38
  Return Data to Application
    #Columns = 38

Subsection #1:
  Access Summary Table Name = DB2INST2.SUMMARY_ZIP_CODE  ID = 2,47
    #Columns = 4
    Relation Scan
      Prefetch: Eligible
    Lock Intents
      Table: Intent Share
        Row : Next Key Share
        Insert Into Sorted Temp Table  ID = t1
          #Columns = 4
          #Sort Key Columns = 1
          Key 1: ZIP_CD (Ascending)
        Sortheap Allocation Parameters:
          #Rows = 8958
          Row Width = 36
        Piped
    Sorted Temp Table Completion  ID = t1
```
Although in our example, the estimated cost is slightly higher for using the replicated MQT (101171 vs. 100975) because we are running on an otherwise idle system of two partitions on one computer. However, the performance advantages of using a replicated MQT in this case become clear when the nodes reside on different computers and the network between them is busy.

So, using replicated MQTs may yield performance advantages when the table you replicate data from:

- Is very frequently joined with.
- Is rarely, if ever, updated.
- Is not too large (although you might consider replicating infrequently updated larger tables if the one-time cost of replication can be offset by the performance benefits of collocation.

In addition, the locking issues described for REFRESH IMMEDIATE tables don’t occur for replicated MQTs.

**REFRESH IMMEDIATE vs. REFRESH DEFERRED**

REFRESH IMMEDIATE MQTs affect performance of queries just like indexes do. They:

- Speed up the performance of relevant select statements.
- Are automatically chosen by the optimizer whenever it makes sense.
- Can degrade the performance of insert, update and delete statements.
- Cannot be updated directly.
- May occupy considerable disk space.
- May have exclusive locks held during updates of their base tables.

To see the performance impact on updates, see the EXPLAIN output for an INSERT statement shown in Listing 6 (there is no MQT yet).
Listing 6. INSERT on base ZIP_CODE table

<table>
<thead>
<tr>
<th>Section = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Statement:</td>
</tr>
<tr>
<td><code>insert into db2inst2.zip_code(zip_cd, zip, state_cd, state_name, country_name) values (60606, '60606', 'IL', 'Illinois', 'United States')</code></td>
</tr>
<tr>
<td>Estimated Cost = 25</td>
</tr>
<tr>
<td>Estimated Cardinality = 1</td>
</tr>
</tbody>
</table>

Now let's add a MQT with REFRESH IMMEDIATE option and review the EXPLAIN output shown in Listing 7.

Listing 7. INSERTs on MQTs with REFRESH IMMEDIATE can add performance cost

<table>
<thead>
<tr>
<th>Section = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Statement:</td>
</tr>
<tr>
<td><code>insert into db2inst2.zip_code(zip_cd, zip, state_cd, state_name, country_name) values (60606, '60606', 'IL', 'Illinois', 'United States')</code></td>
</tr>
<tr>
<td>Estimated Cost = 50</td>
</tr>
<tr>
<td>Estimated Cardinality = 1</td>
</tr>
</tbody>
</table>

In this particular case, the estimated cost of inserting a record is doubled when a REFRESH IMMEDIATE MQT exists. REFRESH DEFERRED MQTs, on the other hand, do not slow down the performance of insert, update and delete statements.

Use REFRESH IMMEDIATE MQTs in moderation to optimize only frequently run queries in which current data is important. Some MQTs are not eligible for immediate refreshes.

**Let the optimizer decide**

The optimizer may choose to use MQTs with REFRESH IMMEDIATE option instead of its base table, depending on:

- Current statistics on the base table, the MQT and their indexes.
- Value of CURRENT QUERY OPTIMIZATION setting.

The optimizer may choose to use a MQT with REFRESH DEFERRED option if CURRENT REFRESH AGE setting is set to ANY.
Provide the optimizer with a MQT, create appropriate indexes and keep the statistics current. And let the optimizer choose whether to use the base table or the summary one. In certain cases the optimizer will choose not to use the MQT.

You may directly specify MQTs, both REFRESH DEFERRED and REFRESH IMMEDIATE, in the WHERE clause of your SELECT statement regardless of the values of CURRENT REFRESH AGE and CURRENT QUERY OPTIMIZATION settings.

**Conclusion**

As we have seen, MQTs, if applied properly, are very useful in various situations. The above examples illustrate applying MQTs for improving query performance. While MQTs are very convenient, they come at a price of additional disk space. MQTs with REFRESH DEFERRED option do not impact the performance of inserts, updates and deletes on the base tables, while the ones with REFRESH IMMEDIATE option do.