|  |
| --- |
|  |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |
|  |  |  |  |  |  |

Contents

[1. Optimizing DML Operations During Refresh 3](#_Toc384183025)

[1.1. Implementing an Efficient MERGE Operation 3](#_Toc384183026)

[1.2. Maintaining Referential Integrity 5](#_Toc384183027)

[1.3. Purging Data 5](#_Toc384183028)

[2. Using Parallelism to Improve Data Warehouse Refresh 6](#_Toc384183029)

[2.1. Serial Join 6](#_Toc384183030)

[2.2. Parallel Joins 7](#_Toc384183031)

[2.3. Paralle Partition-Wise Joins 9](#_Toc384183032)

[3. Source Books and Articles 9](#_Toc384183033)

# Optimizing DML Operations during Refresh

You can optimize DML performance through the following techniques:

* Implementing an Efficient MERGE Operation
* Maintaining Referential Integrity
* Purging Data

## Implementing an Efficient MERGE Operation

Commonly, the data that is extracted from a source system is not simply a list of new records that needs to be inserted into the data warehouse. Instead, this new data set is a combination of new records as well as modified records. For example, suppose that most of data extracted from the OLTP systems will be new sales transactions. These records are inserted into the warehouse's sales table, but some records may reflect modifications of previous transactions, such as returned merchandise or transactions that were incomplete or incorrect when initially loaded into the data warehouse. These records require updates to the sales table.

As a typical scenario, suppose that there is a table called new\_sales that contains both inserts and updates that are applied to the sales table. When designing the entire data warehouse load process, it was determined that the new\_sales table would contain records with the following semantics:

* If a given sales\_transaction\_id of a record in new\_sales already exists in sales, then update the sales table by adding the sales\_dollar\_amount and sales\_quantity\_sold values from the new\_sales table to the existing row in the sales table.
* Otherwise, insert the entire new record from the new\_sales table into the sales table.

This UPDATE-ELSE-INSERT operation is often called a merge. A merge can be executed using one SQL statement.

Example 1 MERGE Operation

MERGE INTO sales s USING new\_sales n

ON (s.sales\_transaction\_id = n.sales\_transaction\_id)

WHEN MATCHED THEN

UPDATE SET s.sales\_quantity\_sold = s.sales\_quantity\_sold + n.sales\_quantity\_sold,

s.sales\_dollar\_amount = s.sales\_dollar\_amount + n.sales\_dollar\_amount

WHEN NOT MATCHED THEN INSERT (sales\_transaction\_id, sales\_quantity\_sold,

sales\_dollar\_amount)

VALUES (n.sales\_transcation\_id, n.sales\_quantity\_sold, n.sales\_dollar\_amount);

In addition to using the MERGE statement for unconditional UPDATE ELSE INSERT functionality into a target table, you can also use it to:

* Perform an UPDATE only or INSERT only statement.
* Apply additional WHERE conditions for the UPDATE or INSERT portion of the MERGE statement.
* The UPDATE operation can even delete rows if a specific condition yields true.

Example 2 Omitting the INSERT Clause

In some data warehouse applications, it is not allowed to add new rows to historical information, but only to update them. It may also happen that you do not want to update but only insert new information. The following example demonstrates INSERT-only with UPDATE-only functionality:

MERGE USING Product\_Changes S -- Source/Delta table

INTO Products D1 -- Destination table 1

ON (D1.PROD\_ID = S.PROD\_ID) -- Search/Join condition

WHEN MATCHED THEN UPDATE -- update if join

SET D1.PROD\_STATUS = S.PROD\_NEW\_STATUS

Example 3 Omitting the UPDATE Clause

The following statement illustrates an example of omitting an UPDATE:

MERGE USING New\_Product S -- Source/Delta table

INTO Products D2 -- Destination table 2

ON (D2.PROD\_ID = S.PROD\_ID) -- Search/Join condition

WHEN NOT MATCHED THEN -- insert if no join

INSERT (PROD\_ID, PROD\_STATUS) VALUES (S.PROD\_ID, S.PROD\_NEW\_STATUS)

When the INSERT clause is omitted, Oracle performs a regular join of the source and the target tables. When the UPDATE clause is omitted, Oracle performs an antijoin of the source and the target tables. This makes the join between the source and target table more efficient.

Example 4 Skipping the UPDATE Clause

In some situations, you may want to skip the UPDATE operation when merging a given row into the table. In this case, you can use an optional WHERE clause in the UPDATE clause of the MERGE. As a result, the UPDATE operation only executes when a given condition is true. The following statement illustrates an example of skipping the UPDATE operation:

MERGE

USING Product\_Changes S -- Source/Delta table

INTO Products P -- Destination table 1

ON (P.PROD\_ID = S.PROD\_ID) -- Search/Join condition

WHEN MATCHED THEN

UPDATE -- update if join

SET P.PROD\_LIST\_PRICE = S.PROD\_NEW\_PRICE

WHERE P.PROD\_STATUS <> "OBSOLETE" -- Conditional UPDATE

This shows how the UPDATE operation would be skipped if the condition P.PROD\_STATUS <> "OBSOLETE" is not true. The condition predicate can refer to both the target and the source table.

Example 5 Conditional Inserts with MERGE Statements

You may want to skip the INSERT operation when merging a given row into the table. So an optional WHERE clause is added to the INSERT clause of the MERGE. As a result, the INSERT operation only executes when a given condition is true. The following statement offers an example:

MERGE USING Product\_Changes S -- Source/Delta table

INTO Products P -- Destination table 1

ON (P.PROD\_ID = S.PROD\_ID) -- Search/Join condition

WHEN MATCHED THEN UPDATE -- update if join

SET P.PROD\_LIST\_PRICE = S.PROD\_NEW\_PRICE

WHERE P.PROD\_STATUS <> "OBSOLETE" -- Conditional

WHEN NOT MATCHED THEN

INSERT (PROD\_ID, PROD\_STATUS, PROD\_LIST\_PRICE) -- insert if not join

VALUES (S.PROD\_ID, S.PROD\_NEW\_STATUS, S.PROD\_NEW\_PRICE)

WHERE S.PROD\_STATUS <> "OBSOLETE"; -- Conditional INSERT

This example shows that the INSERT operation would be skipped if the condition S.PROD\_STATUS <> "OBSOLETE" is not true, and INSERT will only occur if the condition is true. The condition predicate can refer to the source table only. The condition predicate can only refer to the source table.

Example 6 Using the DELETE Clause with MERGE Statements

You may want to cleanse tables while populating or updating them. To do this, you may want to consider using the DELETE clause in a MERGE statement, as in the following example:

MERGE USING Product\_Changes S

INTO Products D ON (D.PROD\_ID = S.PROD\_ID)

WHEN MATCHED THEN

UPDATE SET D.PROD\_LIST\_PRICE =S.PROD\_NEW\_PRICE, D.PROD\_STATUS = S.PROD\_NEWSTATUS

DELETE WHERE (D.PROD\_STATUS = "OBSOLETE")

WHEN NOT MATCHED THEN

INSERT (PROD\_ID, PROD\_LIST\_PRICE, PROD\_STATUS)

VALUES (S.PROD\_ID, S.PROD\_NEW\_PRICE, S.PROD\_NEW\_STATUS);

Thus, when a row is updated in products, Oracle checks the delete condition D.PROD\_STATUS = "OBSOLETE", and deletes the row if the condition yields true.

The DELETE operation is not as same as that of a complete DELETE statement. Only the rows from the destination of the MERGE can be deleted. The only rows that are affected by the DELETE are the ones that are updated by this MERGE statement. Thus, although a given row of the destination table meets the delete condition, if it does not join under the ON clause condition, it is not deleted.

Example 7 Unconditional Inserts with MERGE Statements

You may want to insert all of the source rows into a table. In this case, the join between the source and target table can be avoided. By identifying special constant join conditions that always result to FALSE, for example, 1=0, such MERGE statements will be optimized and the join condition will be suppressed.

MERGE USING New\_Product S -- Source/Delta table

INTO Products P -- Destination table 1

ON (1 = 0) -- Search/Join condition

WHEN NOT MATCHED THEN -- insert if no join

INSERT (PROD\_ID, PROD\_STATUS) VALUES (S.PROD\_ID, S.PROD\_NEW\_STATUS)

## Maintaining Referential Integrity

In some data warehousing environments, you might want to insert new data into tables in order to guarantee referential integrity. For example, a data warehouse may derive sales from an operational system that retrieves data directly from cash registers. Sales is refreshed nightly. However, the data for the product dimension table may be derived from a separate operational system. The product dimension table may only be refreshed once for each week, because the product table changes relatively slowly. If a new product was introduced on Monday, then it is possible for that product's product\_id to appear in the sales data of the data warehouse before that product\_id has been inserted into the data warehouses product table.

Although the sales transactions of the new product may be valid, this sales data will not satisfy the referential integrity constraint between the product dimension table and the sales fact table. Rather than disallow the new sales transactions, you might choose to insert the sales transactions into the sales table. However, you might also wish to maintain the referential integrity relationship between the sales and product tables. This can be accomplished by inserting new rows into the product table as placeholders for the unknown products.

As in previous examples, we assume that the new data for the sales table is staged in a separate table, new\_sales. Using a single INSERT statement (which can be parallelized), the product table can be altered to reflect the new products:

INSERT INTO product

(SELECT sales\_product\_id, 'Unknown Product Name', NULL, NULL ...

FROM new\_sales WHERE sales\_product\_id NOT IN

(SELECT product\_id FROM product));

## Purging Data

Occasionally, it is necessary to remove large amounts of data from a data warehouse. A very common scenario is the rolling window discussed previously, in which older data is rolled out of the data warehouse to make room for new data.

However, sometimes other data might need to be removed from a data warehouse. Suppose that a retail company has previously sold products from XYZ Software, and that XYZ Software has subsequently gone out of business. The business users of the warehouse may decide that they are no longer interested in seeing any data related to XYZ Software, so this data should be deleted.

One approach to removing a large volume of data is to use parallel delete as shown in the following statement:

DELETE FROM sales WHERE sales\_product\_id IN (SELECT product\_id

FROM product WHERE product\_category = 'XYZ Software');

This SQL statement spawns one parallel process for each partition. This approach is much more efficient than a serial DELETE statement, and none of the data in the sales table will need to be moved. However, this approach also has some disadvantages. When removing a large percentage of rows, the DELETE statement will leave many empty row-slots in the existing partitions. If new data is being loaded using a rolling window technique (or is being loaded using direct-path INSERT or load), then this storage space will not be reclaimed. Moreover, even though the DELETE statement is parallelized, there might be methods that are more efficient. An alternative method is to re-create the entire sales table, keeping the data for all product categories except XYZ Software.

CREATE TABLE sales2 AS SELECT \* FROM sales, product

WHERE sales.sales\_product\_id = product.product\_id

AND product\_category <> 'XYZ Software'

NOLOGGING PARALLEL (DEGREE 8)

#PARTITION ... ; #create indexes, constraints, and so on

DROP TABLE SALES;

RENAME SALES2 TO SALES;

This approach may be more efficient than a parallel delete. However, it is also costly in terms of the amount of disk space, because the sales table must effectively be instantiated twice.

An alternative method to utilize less space is to re-create the sales table one partition at a time:

CREATE TABLE sales\_temp AS SELECT \* FROM sales WHERE 1=0;

INSERT INTO sales\_temp

SELECT \* FROM sales PARTITION (sales\_99jan), product

WHERE sales.sales\_product\_id = product.product\_id

AND product\_category <> 'XYZ Software';

<create appropriate indexes and constraints on sales\_temp>

ALTER TABLE sales EXCHANGE PARTITION sales\_99jan WITH TABLE sales\_temp;

Continue this process for each partition in the sales table.

# Using Parallelism to Improve Data Warehouse Refresh

Parallel operations – except for the most basic ones – typically require data redistribution. Data redistribution is required in order to perform operations such as parallel sorts, aggregations and joins. At the block-granule level there is no knowledge about the actual data contained in an individual granule. Data has to be redistributed as soon as a subsequent operation relies on the actual content. Remember the last car example? The car color mattered, but you don't know – or even control – what color cars are parked where on the street. You redistributed the information about the amount of cars per color to the additional two friends based on their color responsibility; enabling them to do the total counting for the colors, they are in charge of.

Data redistribution takes place between individual PX servers either within a single machine, or, across multiple machines in a Real Application Clusters (RAC). Of course in the latter case interconnect communication is used for the data redistribution.

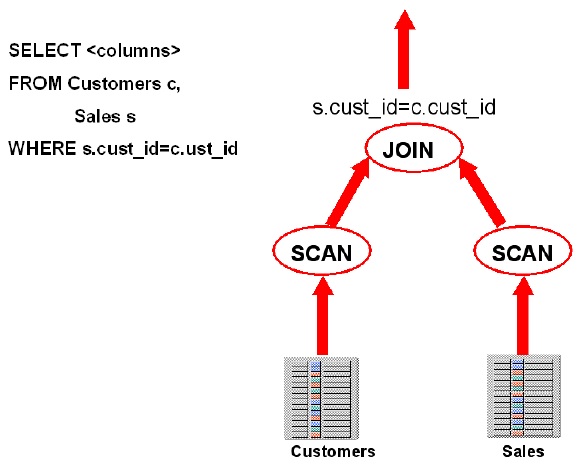
Data redistribution is not unique to the Oracle Database. In fact, this is one of the most fundamental principles of parallel processing, being used by every product that provides parallel capabilities. The fundamental difference and advantage of Oracle's capabilities, however, is that parallel data access (discussed in the granules section earlier) and therefore the necessary data redistribution are not constrained by any given hardware architecture or database setup.

Shared-nothing database systems also require data redistribution unless operations can take advantage of partition-wise joins (as explained further down in this section). In shared-nothing systems, parallel operations that cannot benefit from a partition-wise join – such as a simple three-way table join on two different join keys - always make heavy use of interconnect communication. Because the Oracle Database also enables parallel execution within the context of a node, parallel operations do not always have to use interconnect communication, thus avoiding a potential bottleneck at the interconnect.

The following section will explain Oracle's data redistribution capabilities using the simple example of table joins without any secondary data structures, such as indexes or materialized views.

## Serial Join

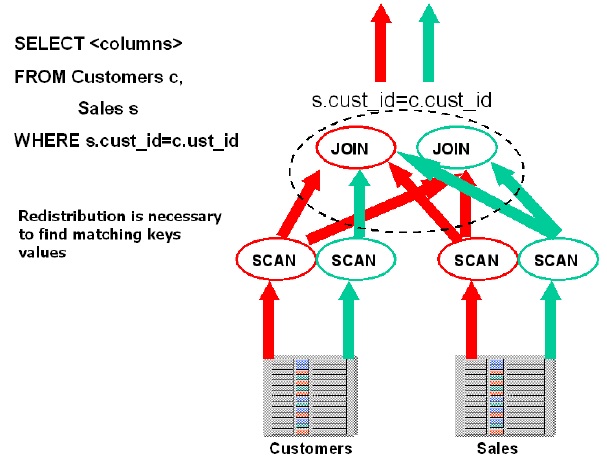
In a serial join a single session reads both tables and performs the join. In this example, we assume two large tables CUSTOMERS and SALES are involved in the join. The database uses full table scans to access both tables. For a serial join, the single serial session (red arrows) can perform the full join because all matching values from the CUSTOMERS table are read by one process. Figure 11 below depicts the serial join.



**Figure 1 Oracle Serial Join**

## Parallel Joins

Processing the same simple join in parallel, a redistribution of rows will become necessary. PX servers scan parts of either table based on block ranges and in order to complete the join, rows have to be distributed between PX servers. Figure 12 depicts the data redistribution for a parallel join at a DOP 2, represented by the green and red arrow respectively. Both tables are read in parallel by both the red and green process (using block-range granules) and then each PX server has to redistribute its result set based on the join key to the subsequent parallel join operator.



**Figure 2 Data redistribution for a simple parallel join**

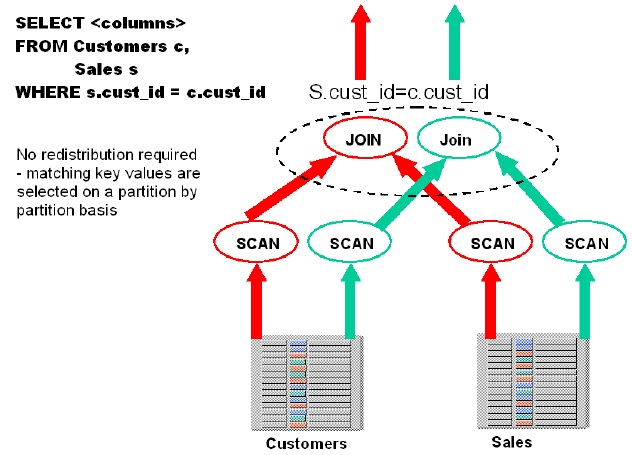
There are many data redistribution methods. The following five are the most common ones:

* HASH: Hash redistribution is very common in parallel execution in order to achieve an equal distribution of work for individual PX servers based on a hash distribution. Hash (re)distribution is the basic parallel execution enabling mechanism for most data warehouse database system.
* BROADCAST: Broadcast redistribution happens when one of the two result sets in a join operation is much smaller than the other result set. Instead of redistributing rows from both result sets, the database sends the smaller result set to all PX servers in order to guarantee the individual servers are able to complete their join operation. The small result set may be produced in serial or in parallel.
* RANGE: Range redistribution is generally used for parallel sort operations. Individual PX servers work on data ranges so that the QC does not have to do any sorting but only to present the individual parallel server results in the correct order.
* KEY: Key redistribution ensures result sets for individual key values to be clumped together. This is an optimization that is primarily used for partial partition-wise joins (see further down) to ensure only one side in the join has to be redistributed.
* ROUND ROBIN: Round-robin data redistribution can be the final redistribution operation before sending data to the requesting process. It can also be used in an early stage of a query when no redistribution constraints are required.

As a variation on the data redistribution methods you may see a LOCAL suffix in a parallel execution plan on a Real Application Clusters (RAC) database. LOCAL redistribution is an optimization in RAC to minimize interconnect traffic for inter-node parallel queries. For example, you may see a BROADCAST LOCAL redistribution in an execution plan indicating that the row set is produced on the local node and only sent to the PX servers on that node. Data redistribution is shown in the SQL execution plan in the PQ Distrib column.

## Parallel Partition-Wise Joins

If at least one of the tables accessed in the join has been partitioned on the join key the database may decide to use a partition-wise join. If both tables are equi-partitioned on the join key the database may use a full partition-wise join. Otherwise, a partial partition-wise join may be used in which one of the tables is dynamically partitioned in memory followed by a full partition-wise join.



**Figure 3 Full partition-wise joins do not require redistribution**

A partition-wise join does not require any data redistribution because individual PX servers will work on the equivalent partitions of both joined tables.

As shown in Figure 14, the red PX server reads data partition one of the CUSTOMERS table AND data partition one of the SALES table; the equi-partitioning of both tables on the join key guarantees that there will no matching rows for the join outside of these two partitions. The PX server will always be able to complete the full join by reading just these matching partitions. The same is true the green PX server, too, and for any pair of partitions of these two tables. Note that partition-wise joins use partition-based granules rather than block-based granules.

The partition-wise join is the fundamental enabler for shared nothing systems. Shared nothing systems typically scale well as long as they can take advantage of partition-wise joins. As a result, the choice of partitioning (distribution) in a shared nothing system is key as well as the access path to the tables. Operations that do not use partition-wise operations in an MPP system often do not scale well.

# Source Books and Articles

1. Kyte T. Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, Second Edition. Apress, 2010.
2. Lane, P. Oracle Database Data Warehousing Guide, 11g Release 2 (11.2). Redwood City: Oracle, 2013.