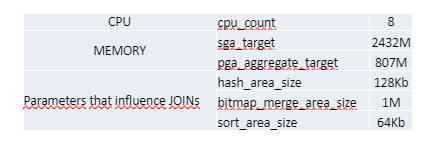
**Techniques of performance boost in a relationship database**

1. **Introduction**

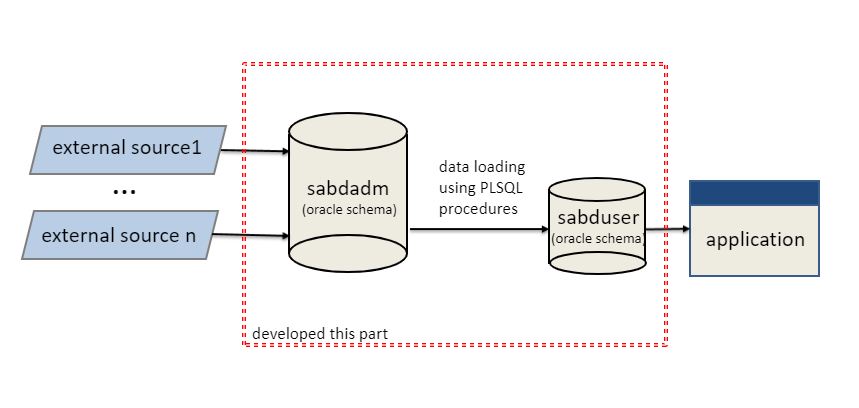
In order to demonstrate different boost performance techniques in relationship databases, we tried to emulate a real-life example where the relationship databases must stand up to their best: deal with a significant amount of data and process it in an acceptable amount of time and acceptable memory consummation. For these tests we used an Oracle environment. Chosen topic in this case was a relationship database used in a bank.

1. **Implementation**

Oracle database configuration used:



Overview of the architecture of the project:



There are two schemas involved in the project:

* SABDADM
* SABDUSER

**SABDADM** schema is a backend schema used to gather and hold also the historical data. Since this schema has no feed, we will generate dummy data in it. The data is found in the following tables:

1. DIM\_DATE – holds the information about the dates. Generated data will be in

interval 01.01.1990 – current date.

2. DIM\_CURRENCY – holds the data about the majority of currencies of the world. The data here is loaded in the table from an external file with SQL Loader utility.

3. DIM\_COUNTRY – holds the data about the countries of the world. The data here is loaded in the table from an external file with SQL Loader utility.

4. DIM\_CUST\_GROUP – holds the data about the customers groups. Each customer can be part of a group (for example members of a family can be in the same group, but they are different individuals and each one of them can have more than an account). Each group has an owner and all the accounts owned by customers from the same group create a container which is unique. The data is randomly generated.

5. DIM\_CUSTOMER - holds the data about the customers. The data is randomly generated.

6. DIM\_ACCOUNT – holds the data about the accounts. The data is randomly

generated.

7. DIM\_OPERATION – holds the data about the operations a customer can make. The data is manually inserted.

8. DIM\_COMMISION – holds the data about the commission retained, based on the operations made. The data is manually inserted.

9. FACT\_DAY\_RATES – holds the data about the exchange rates between currencies, in each day. For each date and each currency, the exchange rate is randomly generated.

10. FACT\_TRANSACTIONS – holds the data about the transactions made by customers. The data is randomly generated.

For generating data, we’ve created two packages: generateDimData and generateFactData. Each table is loaded using a procedure from these packages. However, because of the limitation of our system, we had to limit also the size of the tables and because of that, each procedure can be called with the number of rows we want to be in the tables. In real life situation the main problem here would be how to load the data coming from feeds, in a timely manner. Our problem here is how to load random data in our tables in a timely manner.

We loaded DIM\_COUNTRY and DIM\_CURRENCY tables from external files, using SQL\*Loader utility. The process is running as bulk and is inserting ~200 rows in each table in 1-2 seconds. Next loaded table is DIM\_DATE. Since we were able to generate the data directly in SQL without switching to PL/SQL, ~10000 rows are inserted as bulk in a couple of seconds.

The problem appeared when we had to generate random data. Since we want rows

to be different, we had to generate each row individually. Since it cannot be done only in SQL, we switched to PL/SQL and process each row in a for loop. Since we wanted to have distinct rows, we chose the most expensive way – row by row processing.

**SABDUSER** schema is a frontend schema, though to be used by a reporting application. The feed of this schema is SABDADM, which holds the data. The data is spread across following tables:

1. CUSTOMERS - holds the data about the customers. The data here is merged from sabdadm.dim\_customers table.

2. ACCOUNTS - holds the data about the accounts. The data here is merged from

sabdadm.dim\_accounts table.

3. TRANSACTIONS - holds the data about the transactions. The data here is merged from sabdadm.fact\_transactions table. There are a lot of calculations that are made and inserted in this table, the data been retrieved from sabdadm schema tables.

In order to populate the sabduser schema, we’ve created procedures: load\_customers,load\_accounts and load\_transactions. First two are merging the data from dimensions tables, so taking in consideration that usually in these types of tables we don’t have a lot of data, they will run quite fast. However, we identified here two problematic cases:

* Do not crash or return an exception when one row cannot be merged or inserted and also know the reason for it.
* When loading transaction table by running load\_transactions, the process is slow because of all the calculations and joins.

1. **Results**

**Test one**: Bulk processing, implicit and explicit cursors

**The bulk processing** features of PL/SQL are designed specifically to reduce the number of context switches required to communicate from the PL/SQL engine to the SQL engine.

Use the BULK COLLECT clause to fetch multiple rows into one or more collections with a single context switch.

Use the FORALL statement when you need to execute the same DML statement repeatedly for different bind variable values.

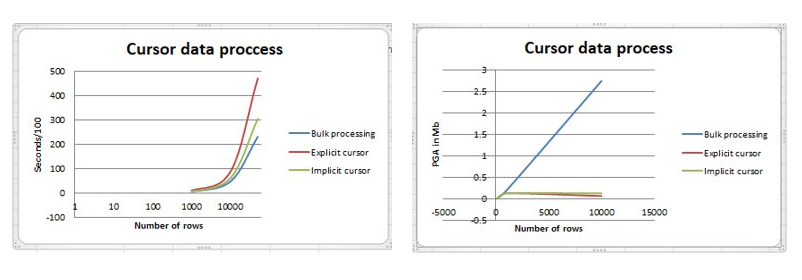
A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

* Implicit cursors
* Explicit cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.



Conclusions:

* Bulk processing is faster because is doing a single context switching and process all rows at once;
* Bulk processing is “eating” our PGA because all rows are kept in memory, to the difference of explicit and implicit cursors;
* Explicit and implicit cursors are running slower because for each processed row, are doing a context switching;
* Implicit cursor is running faster than explicit cursor because operations of “open”, “fetch” and “close” are not performed and process the row as soon as is retrieved from the database;

**Test two**: Pipelined function versus non pipelined functions

Basically, when you would like a PLSQL (or java or c) routine to be the «source»

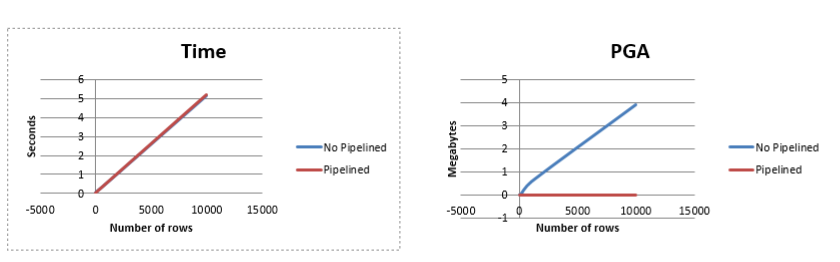
of data -- instead of a table -- you would use a pipelined function.

PIPELINED functions will operate like a table.

A PL/SQL function may be used in a data warehouse database to transform large amounts of data. This might also involve massaging the data in a series of transformations, each performed by different functions. Prior to Oracle Database 9, large transformations required either significant memory overhead, or storing the data in intermediate tables between each stage of the transformation. The loading process caused immense performance degradations in both cases.

With non-pipelined table functions, the entire collection returned by a table function must be constructed and returned to the server before the query can return a single result row. Pipelining enables rows to be returned iteratively, as they are produced. This also reduces the memory that a table function requires, as the object cache does not need to materialize the entire collection.

Results:



Conclusions:

None of these options were viable for what we needed, generating data via functions is taking 10x longer for a row by row processing.

The main difference between the two of them, is that as soon as a row is retrieved, the pipelined functions are pipe-ing them to stdout. Pipelined function finish faster, because it will process only 10 rows. Non pipelined function will process 1000 rows in this case, load them in memory and afterwards, will limit the output to 10 rows.

**Test three**: performance boost using horizontal table partitioning

There are two types of partitioning:

* Partitioning by list;
* Partitioning by range;

Using partitioning, we avoid a full index scan, or worse, a full table scan and we only looking into specific partition(s). In Oracle database, in order to look into a specific partition, we can either select from that partition or use in the “WHERE” the column that is used for partitioning:

***SELECT ….***

***FROM <table> PARTITION (“<partition\_name>”);***

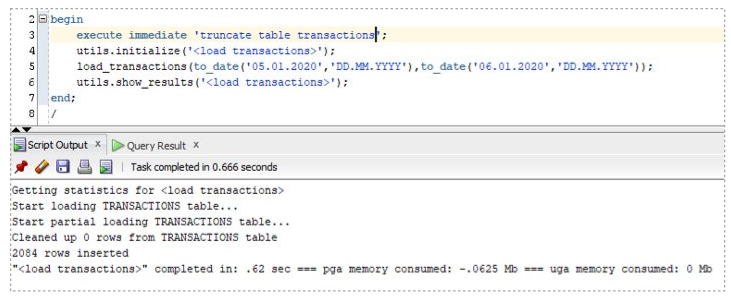
***SELECT …***

***FROM <table>***

***WHERE <column\_used\_for\_partition> <condition> …***

Conclusions:

After the two tables were partitioned, loading data process improved considerably from ~80 seconds for 2084 rows to less than a second:



**Test four**: performance boost using materialized views

We also analyzed the benefits of using materialized views. Materialized views are a database object, now available in the most of database systems. They were first implemented by Oracle, followed by the others RDBMS as well.

The main difference between materialized views and normal views, is that in the materialized views case the result of the query that forms it is cached. It will not be changed, until the view is refreshed. They basically represent a snapshot of the data present at moment x in the database.

In SQL Server, materialized views are implemented in a different way and they are known more under the “Indexed views” term. The difference in this case is that the views are always synchronized with the data from the based tables that creates the view.

**Conclusions**:

After creating a materialized view from the select statement that populate our table and a procedure that loads the data from created materialized view, the results were even better: After partitioning and materialized view changes, we managed to do a full transaction load (~35k rows) in 1.4 seconds, instead of ~1900 seconds as it was initially.