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

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. We have two schemas here: SABDADM and 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 spread across 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).

*group --> group member1 --> account1*

*...*

*accountN*

*group member2 --> account1*

*...*

*accountN*

Each group has an owner and all the accounts owned by customers from the same group create a container which is unique. The data here is randomly generated.

1. DIM\_CUSTOMER - holds the data about the customers. The data here is randomly generated.
2. DIM\_ACCOUNT – holds the data about the accounts. The data here is randomly generated.
3. DIM\_OPERATION – holds the data about the operations a customer can make. The data here is manually inserted.
4. DIM\_COMMISION – holds the data about the commission retained, based on the operations made. The data here is manually inserted.
5. FACT\_DAY\_RATES – holds the data about the exchange rates between currencies, in each day. For example, in 01.01.1990 we will have something like:

*EUR -> USD – 1.31*

*USD -> EUR – 1/1.31*

*EUR – EUR – 1*

For each date and each currency, the exchange rate is randomly generated.

1. FACT\_TRANSACTIONS – holds the data about the transactions made by customers. The data here 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 timely manner.

***Generate random data in main schema 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;

Analyzed a couple of ways we can achieve that and we chose to use implicit cursors in the procedures to generate the data.

**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 type 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.

***Logging errors***

We had to come with a way of not intreruping a loading when a row cannot be merged or inserted and also know the reason for it. We avoided using additional PL/SQL blocks with PRAGMA\_AUTONOMOUS\_TRANSACTION and we chose to use a 11.2 Oracle package called DBMS\_ERRLOG. We created an error table for each table using:

DBMS\_ERRLOG.CREATE\_ERROR\_LOG(<table\_name>, <error\_table\_name>)

which is having columns with information about encountered errors. Also we added at the end of each merge and insert statement line:

LOG ERRORS INTO <error\_table\_name> (<tag>) REJECT LIMIT UNLIMITED;

making each rejected row to populate the error table accordingly.

We also added a table that contains the merged errors called ERROR\_TABLE\_LOG. We populate this table dynamically, by calling a procedure in the others, if there are rows rejected.

***Load transactions in a timely manner***

To solve this problem we chose to partition tables FACT\_DAY\_RATES and FACT\_TRANSACTIONS tables. After that, we created a materialized view from the select that populate the sabduser.transaction table. We managed to do a full load in ~1.4 seconds, dropping down from ~2000 seconds.