

Practices for Designing and Improving Data Extraction in a Virtual Data Warehouses Project

Ion Lungu, Manole Velicanu, Adela Bara, Vlad Diaconiță, Iuliana Botha

Abstract: The problem of low performance in data extraction from data warehouse can be critical in the Business Intelligence projects because of the major impact in the using the data from data warehouse: if a BI report is taking a lot of time to run or the data displayed are no longer available for taking critical decisions, the project can be compromised. In this paper we present an overview of an implementation of a Business Intelligence project in a national company, the problems we confronted with and the techniques that we applied to reduce the cost of execution for improving query performance in this decisional support system. We'll present several techniques that we applied to reduce queries' execution time and to improve the performance of the BI analyses and reports.

Keywords: Business Intelligence, Virtual data warehouse, Data extraction, Query optimization, Partitioning techniques, Indexes, Analytical functions

1 Introduction

Business Intelligence Systems allows users to manipulate large sets of data in real time manner in order to analyze and take decisions. So, the main goal of Business Intelligence Systems (BIS) is to assist managers, at different levels in the organization, in taking decisions and to provide in real time representative information, to help and support them in their activities such as analyzing departmental data, planning and forecasting activities for their decision area [1]. In essence, managers at every departmental level can have a customized view that extracts information from transactional sources and summarizes it into meaningful indicators. These systems usually work with large sets of data and require a short response time and if you consider using analytical tools like OLAP against virtual data warehouses then you have to build your system through SQL queries and retrieve data directly from OLTP systems. In this case, the large amount of data in ERP systems may lead to an increase of responding time for BIS. That's why you should consider applying optimization techniques in order to improve the BI system's performance.

2 Problems in developing a BI project involving virtual data warehouses

In a research project that we conducted in one of the multinational companies from our country we applied the concepts mentioned in this paper and also tried to meet the requests from the executives and managers of the company. For the project life cycle we applied the framework described in the book Executive Information Systems [2], tailored for the specific needs of our project. With BI techniques, like data warehouse, OLAP, data mining, portal we succeeded to implement the BI system's prototype and to validate it with the managers and executives. The BIS gathers data, using the ERP system partially implemented in the organization to extract data from different functional areas or modules such as: financial, inventory, purchase, order management or production. Information from these functional modules within the ERP system is managed by a relational database management system - Oracle Database Enterprise Edition 10g. From the relevant data sources we applied an ETL (extract, transform and load) process to transport data from source to destination. In this step we had to choose between the two data warehouses solutions: stored data and virtual extraction. After a comparative analysis between these techniques we choose the second solution. The major elements in this choice were that the ERP system was not yet fully implemented and there are many more changes to do, the amount of data is not so large - 3 millions records from January 2007 to August 2007, and implementing a virtual data warehouse is fastest and needs a lower budget than a traditional data warehouse. We are also considering the development of a traditional data warehouse after testing and implemented the actual prototype. So, we build the virtual data warehouse based on a set of views that collects data from the ERP system based on an ETL process that we designed. After we developed the analytical decisional reports and test them in a real organizational environment with over 100 users, we measured the performance of the system and the main problem was the high cost of execution. These analytical reports consumed over 80 percent of the total resources allocated for the ERP and BI systems. Also, the critical moment when the system was breaking down was at the end of each month when all transactions from functional modules were posted to the General

Ledger module. Testing all parameters and factors we concluded that the major problem was in the data extraction from the views. First solution was to rewrite the views and build materialized views and semi-aggregate tables. Data sources are loaded in these tables by the ETL process periodically, at the end of the month after posting to the General Ledger or at user request. The ETL process contains a set of procedures grouped in three types of packages: extraction, transforming and loading. The target data are stored in tables used directly by the analytical reports. A benefit of this solution is that it eliminates the multiple joins from the views and also that we can use the ETL process to load data in future data warehouse. After these operations were completed we re-tested the systems under real conditions. The time for data extraction was again too long and the costs of executions consumed over 50 percent of total resources. Next we considered using some optimization techniques like: table partitioning, indexing, using hints and using analytical functions instead of data aggregation in some reports. In the following section we describe these techniques and provide a comparative analysis of some of our testing results.

3 Applying optimization techniques

3.1 Partitioning techniques

The main objective of the partitioning technique is to radically decrease the amount of disk activity and to limit the amount of data to be examined or operated on and to enable parallel execution required to perform Business Intelligence queries against virtual data warehouses. Tables are partitioning using a partitioning key that is a set of columns which will determine by their conditions in which partition a given row will be store. Oracle Database 10g on which the ERP is implemented provides three techniques for partitioning tables:

Range Partitioning - specify by a range of values of the partitioning key;

List Partitioning - specify by a list of values of the partitioning key;

Hash Partitioning - a hash algorithm is applied to the partitioning key to determine the partition for a given row;

Also, there can be use sub partitioning techniques in which the table in first partitioned by range/list/hash and then each partition is divided in sub partitions:

Composite Range-Hash Partitioning - a combination of Range and Hash partitioning techniques, in which a table is first range-partitioned, and then each individual range-partition is further sub-partitioned using the hash partitioning technique;

Composite Range-List Partitioning - a combination of Range and List partitioning techniques, in which a table is first range-partitioned, and then each individual range-partition is further sub-partitioned using the list partitioning technique;

Index-organized tables can be partitioned by range, list, or hash. [5] In our case we consider evaluating each type of partitioning technique and choose the best method that can improve the BI system's performance.

Thus, we create two tables based on the main table which is used by some analytical reports and compare the execution cost obtained by applying the same query on them. First table BALANCE_RESULTS_A contained non partitioned data and is the target table for an ETL sub-process. It counts 55000 rows and the structure is shown below in the scripts. The second table BALANCE_RESULTS_B is a range partitioned table by column ACC_DATE which refers to the accounting date of the transaction. This table has four partitions as you can observe from the script below. Then, we create a third table which is partitioned and that contained also for each range partition four list partitions on the column "Division" which is very much used in data aggregation in our analytical reports.

TABLE: QUERY:	table A	table B		table C		
	Not partitioned	Partition range by date on column "ACC_DATE"		Partition range by date with four list partitions on column "DIVISION"		
		Without partition clause	Partition (QT1)	Without partition clause	Partition (QT1)	Sub-partition (QT1_AFO)
Select * from	100	121	-	224	-	-
where extract (month from acc_date) =1	103	124	42	227	72	72
... and division='h.AFO divizi'	101	122	42	10	8	72
select sum(acc_d) TD, sum(acc_c) TC from balance_results_a a where extract (month from acc_date) =1 and division='h.AFO divizi'	101	122	42	10	8	72
... and management_unit = 'MTN'	101	122	42	225	72	72
select /*- USE_HASH (a n)*/ a *, a.location a.country, a.region from balance_results_a a management_units a where a.management_unit=a.management_unit and extract (month from acc_date) =1	100	127	48	231	76	76
... and a division = 'h.AFO divizi'	105	126	45	14	8	75
/*- USE_NL (a n)*/	191	212	71	100	8	101
/*- USE_NL (a n)*/	131	172	58	80	8	88
... WITH INDEXES						
/*- USE_MERGE (a n)*/	104	125	45	13	8	75
... and v.management_unit = 'MTN'	104	125	45	75	8	75
/*- USE_NL (a n)*/	104	125	45	11	8	75
/*- USE_NL (a n)*/	102	123	43	13	8	73
... WITH INDEXES						
/*- USE_MERGE (a n)*/	102	123	43	13	8	73
... WITH INDEXES						

Figure 1: Comparative analysis results - the grey marked ones have the best execution cost of the current query CodeSection 1. create table balance_results_b (ACC_DATE date not null, PERIOD varchar2(15) not null, ACC_D number, ACC_C number, ACCOUNT varchar2(25), DIVISION varchar2(50), SECTOR varchar2(100), MANAGEMENT_UNIT varchar2(100)) partition by range (ACC_DATE) (partition QT1 values less than (to_date('01-APR-2007', 'dd-mon-yyyy')), partition QT2 values less than (to_date('01-JUL-2007', 'dd-mon-yyyy')), partition QT3 values less than (to_date('01-OCT-2007', 'dd-mon-yyyy')), partition QT4 values less than (to_date('01-JAN-2008', 'dd-mon-yyyy')));

create table balance_results_C (ACC_DATE date not null, PERIOD varchar2(15) not null, ACC_D number, ACC_C number, ACCOUNT varchar2(25), DIVISION varchar2(50), SECTOR varchar2(100), MANAGEMENT_UNIT varchar2(100)) partition by range (ACC_DATE) subpartition by list (DIVISION) (partition QT1 values less than (to_date('01-APR-2007', 'dd-mon-yyyy')) (subpartition QT1_OP values ('a.MTN', 'b.CTM', 'c.TRS', 'd.WOD', 'e.DMA'), subpartition QT1_GA values ('f.GA op', 'g.GA corp'), subpartition QT1_AFO values ('h.AFO div', 'i.AFO corp'), subpartition QT1_EXT values ('j.EXT', 'k.Imp')), partition QT2 values less than (to_date('01-JUL-2007', 'dd-mon-yyyy')) (subpartition QT2_OP values ('a.MTN', 'b.CTM', 'c.TRS', 'd.WOD', 'e.DMA'), subpartition QT2_GA values ('f.GA op', 'g.GA corp'), subpartition QT2_AFO values ('h.AFO div', 'i.AFO corp'), subpartition QT2_EXT values ('j.EXT', 'k.Imp')), partition QT3 values less than (to_date('01-OCT-2007', 'dd-mon-yyyy')) (subpartition QT3_OP values ('a.MTN', 'b.CTM', 'c.TRS', 'd.WOD', 'e.DMA'), subpartition QT3_GA values ('f.GA op', 'g.GA corp'), subpartition QT3_AFO values ('h.AFO div', 'i.AFO corp'), subpartition QT3_EXT values ('j.EXT', 'k.Imp')), partition QT4 values less than (to_date('01-JAN-2008', 'dd-mon-yyyy')) (subpartition QT4_OP values ('a.MTN', 'b.CTM', 'c.TRS', 'd.WOD', 'e.DMA'), subpartition QT4_GA values ('f.GA op', 'g.GA corp'), subpartition QT4_AFO values ('h.AFO div', 'i.AFO corp'), Subpartition QT4_EXT values ('j.EXT', 'k.Imp')));

Analyzing the decision support reports we choose a sub-set of queries that are always performed and which are relevant for testing the optimization techniques. We run these queries on each test table: A, B and C and compare the results in figure 1.

In conclusion, the best technique in our case is to use table C instead table A or table B, that means that partitioning by range of ACC_DATE with type DATE and then partitioning by list of DIVISION with type VARCHAR2

is the most efficient method. Also, we obtained better results with table B partitioned by range of ACC_DATE than table A non-partitioned.

3.2 Indexing and applying hints

Oracle uses indexes to avoid the need for large-table, full-table scans and disk sorts, which are required when the SQL optimizer cannot find an efficient way to service the SQL query.

The oldest and most popular type of Oracle indexing is a standard b-tree index, which excels at servicing simple queries. The b-tree index was introduced in the earliest releases of Oracle and remains widely used with Oracle. While b-tree indexes are great for simple queries, they are not very good for the following situations:

Low-cardinality columns with less than 200 distinct values do not have the selectivity required in order to benefit from standard b-tree index structures.

No support for SQL functions. The B-tree indexes are not able to support SQL queries using Oracle's built-in functions. Oracle 9i provides a variety of built-in functions that allow SQL statements to query on a piece of an indexed column or on any one of a number of transformations against the indexed column.

Oracle bitmap indexes are very different from standard b-tree indexes. In bitmap structures, a two-dimensional array is created with one column for every row in the table being indexed. Each column represents a distinct value within the bit mapped index. This two-dimensional array represents each value within the index multiplied by the number of rows in the table. At row retrieval time, Oracle decompresses the bitmap into the RAM data buffers so it can be rapidly scanned for matching values. These matching values are delivered to Oracle in the form of a Row-ID list, and these Row-ID values may directly access the required information. The real benefit of bit mapped indexing occurs when one table includes multiple bit mapped indexes. Each individual column may have low cardinality. The creation of multiple bit mapped indexes provides a very powerful method for rapidly answering difficult SQL queries.

One of the most important advances in Oracle indexing is the introduction of function-based indexing. Function-based indexes allow creation of indexes on expressions, internal functions, and user-written functions in PL/SQL and Java. Function-based indexes ensure that the Oracle designer is able to use an index for its query.

Oracle indexes can greatly improve query performance but there are some important indexing concepts to review: index clustering and index block sizes. Indexes that experience lots of index range scans of index fast full scans (as evidence by multi block reads) will greatly benefit from residing in a 32 k block size. Today, most Oracle tuning experts utilize the multiple block size feature of Oracle because it provides buffer segregation and the ability to place objects with the most appropriate block size to reduce buffer waste [6].

The optimizer decision to perform a full-table vs. an index range scan is influenced by the clustering factor (located inside the dba_indexes view), db_block_size, and avg_row_len. It is important to understand how the optimizer uses these statistics to determine the fastest way to deliver the desired rows. Conversely, a high clustering_factor, where the value approaches the number of rows in the table (num_rows), indicates that the rows are not in the same sequence as the index, and additional I/O will be required for index range scans. As the clustering factor approaches the number of rows in the table, the rows are out of sync with the index.

When a SQL statement is executed the query optimizer determines the most efficient execution plan after considering many factors related to the objects referenced and the conditions specified in the query. The optimizer estimates the cost of each potential execution plan based on the statistics available in the data dictionary for the data distribution and storage characteristics of the tables, indexes, and partitions accessed by the statement and it evaluates the execution cost. This is an estimated value depending on resources used to execute the statement which includes I/O, CPU, and memory [4]. This evaluation is an important factor in the processing of any SQL statement and can greatly affect execution time.

We can override the execution plan of the query optimizer with hints inserted in SQL statement. A SQL statement can be executed in many different ways, such as *full table scans*, *index scans*, *nested loops*, *hash joins* and *sort merge joins*. We can set the parameters for query optimizer mode depending on our goal. For BIS, time is one of the most important factor and we should optimize a statement with the goal of best response time. To set up the goal of the query optimizer we can use one of the hints that can override the OPTIMIZER_MODE initialization parameter for a particular SQL statement [5]. The optimizer first determines whether joining two or more tables having UNIQUE and PRIMARY KEY constraints and places these tables first in the join order. The optimizer then optimizes the join of the remaining set of tables and determinate the cost of a join depending on the following methods:

Hash joins are used for joining large data sets and the tables are related with an equality condition join. The optimizer uses the smaller of two tables or data sources to build a hash table on the join key in memory and then

it scans the larger table to find the joined rows. This method is best used when the smaller table fits in available memory. The cost is then limited to a single read pass over the data for the two tables.

Nested loop joins are useful when small subsets of data are being joined and if the join condition is an efficient way of accessing the second table.

Sort merge joins can be used to join rows from two independent sources. Sort merge joins can perform better than hash joins if the row sources are sorted already and a sort operation does not have to be done.

We compare these techniques using hints in SELECT clause and based on the results in table 1 we conclude that the Sort merge join is the most efficient method when table are indexed on the join column for each type of table: non-partitioned, partitioned by range and partitioned by range and sub partitioned by list. The significant improvement is in sub partitioned table in which the cost of execution was drastically reduce at only 6 points compared to 102 points of non-partitioned table. Without indexes, the most efficient method is hash join with best results in partitioned table and sub partitioned table.

3.3 Re-write aggregate queries in an analytical mode

Aggregate functions applied on a set of records return a single result row based on groups of rows. Aggregate functions such as SUM, AVG and COUNT can appear in SELECT statement and they are commonly used with the GROUP BY clauses. In this case the set of records is divided into groups, specified in the GROUP BY clause. Aggregate functions are used in analytic reports to divide data in groups and analyze these groups separately and for building subtotals or totals based on groups.

Analytic functions process data based on a group of records but they differ from aggregate functions in that they return multiple rows for each group. In the latest versions in addition to aggregate functions Oracle implemented analytical functions to help developers building decision support reports [5]. The group of rows is called a window and is defined by the analytic clause. For each row, a sliding window of rows is defined and it determines the range of rows used to process the current row. Window sizes can be based on either a physical number of rows or a logical interval, based on conditions over values [3]

Analytic functions are performed after completing operations such joins, WHERE, GROUP BY and HAVING clauses, but before ORDER BY clause. Therefore, analytic functions can appear only in the select list or ORDER BY clause [4].

Analytic functions are commonly used to compute cumulative, moving and reporting aggregates. The need for these analytical functions is to provide the power of comparative analyses in the BI reports and to avoid using too much aggregate data from the virtual data warehouse.

Thus, we can apply these functions to write simple queries without grouping data like the following example in which we can compare the amount of current account with the average for three consecutive months in the same division, sector and management unit, back and forward:

CodeSection 2. select period, division, sector, management_unit, acc_d, avg(acc_d) over (partition by division, sector, management_unit order by extract (month from acc_date) range between 3 preceding and 3 following) avg_neighbors from balance_results_a

Or we can obtain the previous three months cumulative accountings applying analytical sum function:

CodeSection 3. select period, division, sector, acc_d, sum(acc_d) over (partition by division, sector order by extract (month from acc_date) range between 3 preceding and current row)d3_months_preceding, acc_c, sum(acc_c) over (partition by division, sector order by extract (month from acc_date) range between 3 preceding and current row)c3_months_preceding from balance_results_a where extract (month from acc_date) =6

Analyzing the execution cost obtained in each sample table (A, B and C) we can observe that the lowest cost is o table A (104) followed by B (125) and the highest cost in on table C (225). So, the conclusion is that it is not a very good solution to partition tables when applying analytical functions.

4 Summary and Conclusions

The virtual data warehouse that we designed and tested is based on a set of views that extracts, joins and aggregates rows from many tables from an ERP system. In order to develop this decisional system we have to build analytical reports based on SQL queries. But as the data extraction is the major time and cost consuming job we had to search for a solution. So, we tested several techniques that improved the performance. In this paper we presented some of these techniques, like table partitioning, using hints, loading data from the online views

in materialized views or in tables in order to reduce multiple joins and minimized the execution costs. Also, for developing reports an easy and important option is to choose analytic functions for predictions (LAG and LEAD), subtotals over current period (SUM and COUNT), classifications or ratings (MIN, MAX, RANK, FIRST_VALUE and LAST_VALUE). Another issue that is discussed in this article is the difference between data warehouses in which the data are stored in aggregate levels and virtual data warehouse. Our conclusion is that classical warehouse provides performance as virtual warehouse provides ease in development and keeps the costs down. Even if the purpose of a project is developing a classical warehouse, it's better to create a virtual warehouse before doing that.

References

- [1] Lungu Ion, Bara Adela, Fodor Anca, "Business Intelligence tools for building the Executive Information Systems," *5thRoEduNet International Conference, Lucian Blaga University, Sibiu, June, 2006*.
- [2] Lungu Ion, Bara Adela, *Executive Information Systems*, ASE Publisher, 2007.
- [3] Lungu Ion, Bara Adela, Diaconita Vlad, "Building Analytic Reports for Decision Support Systems - Aggregate versus Analytic Functions," *Economy Informatics Review*, nr.1-4, pg. 17-20, ISSN 1582-7941, 2006.
- [4] Oracle Corporation, "Database Performance Tuning Guide 10g Release 2 (10.2)" *Oracle Publications*, Part Number B14211-01, 2005.
- [5] Oracle Corporation, "Oracle Magazine" *Oracle Publications*, 2006.
- [6] Donald K. Burleson, *Oracle Tuning*, ISBN 0-9744486-2-1, 2006.
- [7] Oracle Corporation Documentation, www.oracle.com

Ion Lungu, Manole Velicanu, Adela Bara, Vlad Diaconiță, Iuliana Botha
Academy of Economic Studies
Economic Informatics Department
Bucharest, Romania
E-mail: ion.lungu@ie.ase.ro, manole.velicanu@ie.ase.ro, bara.adela@ie.ase.ro
diaconita.vlad@csie.ase.ro iuliana.botha@ie.ase.ro