Solutions for improving data extraction from virtual data warehouses

Adela BÂRA
Economic Informatics Department, Academy of Economic Studies
Bucharest, ROMANIA
bara.adela@ie.ase.ro

Abstract: The data warehousing project's team is always confronted with low performance in data extraction. In a Business Intelligence environment this problem can be critical because the data displayed are no longer available for taking decisions, so the project can be compromised. In this case there are several techniques that can be applied to reduce queries' execution time and to improve the performance of the BI analyses and reports. Some of the techniques that can be applied to reduce the cost of execution for improving query performance in BI systems will be presented in this paper.
***Keywords**: Virtual data warehouse, Data extraction, SQL tuning, Query performance*

1 Introduction

The Business Intelligence (BI) systems manipulate data from various organizational sources like files, databases, applications or from the Enterprise Resource Planning (ERP) systems. Usually, data from these sources is extracted, transformed and loaded into a primary target area, called staging area which is managed by a relational database management system. Then, in order to build analytical reports needed in a BI environment, a second ETL (extract, transform and load) process is applied to load data into a data warehouse (DW). There are two ways to implement a DW: to store data in a separate repository which is the traditional method and to extract data directly from the relational database which manages the staging repository. The last solution is usually applied if the ERP system is not yet fully implemented and the amount of data is not as huge as the queries can be run in a reasonable time (less than 30 minutes). The implementation of a virtual data warehouse is fastest and requires a low budget then a traditional data warehouse. Also this method can be applied in a prototyping phase but after the validation of the main functionalities, data can be extracted and loaded into a traditional data warehouse. It's a very good practice to use the staging area already build for the second ETL process to load data into the data warehouse.

This paper presents some aspects of the implementation of a virtual data warehouse in a national company where an ERP was recently setup and a set of BI reports must be developed quickly. Based on a set of views that collects data from the ERP system, a virtual data warehouse based on an ETL process was designed. The database management system (DBMS) is Oracle Database 10g Release 2 and the reports were developed in Oracle Business Intelligence Suite (OBI). After the development of the analytical BI reports, the project team run several tests in a real organizational environment and measured the performance of the system. The main problem was the high cost of execution. These reports were over 80% resource consuming 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 the month when all transactions from functional modules were posted to the General Ledger module. After testing all parameters and factors, the team concluded that the major problem was in the data extraction from the relational database. So, in order to improve the query performance, some of the main optimization techniques are considered.

2 An overview of the SQL execution process

The query performance depends on one side on the technology and the DBMS that are used and on the other side on the way queries are executed and data are processed. So, first let's take a look on the way Oracle Database manages the queries. There are two memory structures which are responsible with SQL processing: the System Global Area (SGA) - a shared memory area that contains data and control information for the instance and the Program Global Area (PGA) - a private memory region containing data and control information for each server process.

The main component in the SGA that affect query optimization process is the Shared pool area which caches various SQL constructs that can be shared among users and contains shared SQL areas, the data dictionary cache, and the fully parsed or compiled representations of PL/SQL blocks. A single shared SQL area is used by multiple users that issue the same SQL statement. The size of the shared pool affects the number of disk reads. When a SQL statement is executed, the server process checks the dictionary cache for information on object ownership, location, and privileges and if it is not present, this information is loaded into the dictionary cache through a disk read. The disk reads and parsing are expensive operations; so it is preferable that repeated executions of the same statement find required information in memory, but this process require a large amount of memory. So in conclusion, the size of the shared pool leads to better SQL management by reducing disk reads, shared SQL queries, reducing hard paring and saving CPU resources and improving scalability.

The PGA is a non-shared memory area that is allocated for each server process that can read and write to it. The Oracle Database allocates a PGA when a user connects to an Oracle database. So, a PGA area contains the information about: the user session that initiated it, the cursor that is executed in the PGA and the SQL work areas. The main components which affect the query execution are the SQL work areas. A SQL query is executed in a SQL work area based on an execution plan and algorithm: hash, sort, merge. Thus, the SQL work area allocates a hash area or a sort area or a merge area in which the query is executed. These algorithms are applied depending on the SQL operators, for example a sort operator uses a work area called the sort area to perform the in-memory sort of a set of rows. A hash-join operator uses a work area called the hash area to build a hash table from its left input. If the amount of data to be processed by these two operators does not fit into a work area, then the input data is divided into smaller pieces. This allows some data pieces to be processed in memory while the rest are spilled to temporary disk storage to be processed later. But the response time increases and it affects the query performance. The size of a work area can be controlled and tuned, but in general bigger database areas can significantly improve the performance of a particular operator at the cost of higher memory consumption. The best solution that can be applied is to use Automated SQL Execution Memory (PGA) Management which provides an automatic mode for allocating memory for SQL working. Thus the working areas that are used by memory-intensive operators (sorts and hash-joins) can be automatically and dynamically adjusted. This feature of Oracle Database offers several performance and scalability benefits for analytical reports workloads used in a BI environment or mixed workloads with complex queries. The overall system performance is maximized, and the available memory is allocated more efficiently among queries to optimize both throughput and response time [1].

Another important component of the Oracle Database is the Query optimizer that creates the execution plan for a SQL statement. The execution plan can greatly affect the execution time of a SQL query

and it consists in a series of operations that are performed in sequence to execute the specified statement. The Query optimizer considers many factors related to the objects referenced and the conditions specified in the statement such as: statistics gathered for the system related to the I/O operations, CPU resources and schema objects; information in the data dictionary; conditions in WHERE clause; execution hints supplied by the developers. Based on the evaluation of these factors the Query optimizer decides which is the most efficient path to access data and how to join tables (full-scan, hash, sort, and merge algorithms). In conclusion the execution plan contains all information of a SQL statement execution and in order to improve query performance we have to analyze this plan and to try to eliminate some of the factors that affect the performance.

3 Optimization solutions

3.1. Materialized views

To reduce the multiple joins between relational tables in a virtual data warehouse, the first solution was to rewrite the views and build materialized views and semi-aggregate tables on the staging area. Data sources are loaded in these tables by the ETL (extract, transform and load) process periodically, for example at the end of the week or at the end of the month after posting to the General Ledger. A benefit of this solution is that it eliminates the joins from the views and the ETL process can be used to load data in a future data warehouse that will be implemented after the prototype validation.

After re-write the queries in terms of materialized views, the project team re-test the system under real conditions. The time for data extraction was again too long and the costs of executions consumed over 50% of total resources. So, on these materialized views and tables some of optimization techniques must be applied. These techniques are: table partitioning, indexing, using hints and using analytical functions instead of data aggregation in some reports.

3.2 Partitioning

The main objective of the partitioning technique is to decrease the amount of disk activity and limiting the amount of data to be examined or operated on and enabling parallel execution required to perform 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 our ERP is implemented provides three techniques for partitioning tables [1]:

- 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;

Sub partitioning techniques can be applied and first tables are 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.

In our case we consider evaluating each type of partitioning technique and choose the best method that can improve the queries' performance. Some of our research can be found also in [2] and [3].

For the loading process we created two tables based on the main table and compare the execution cost obtained by applying the same query on them. First table TEST_A

contained un-partitioned data and is the target table for an ETL process. It counts 100000 rows and the structure is shown below in the scripts. The second table TEST_B is a range partitioned table by column T_DATE which refers to the date of the transaction. This table has four partitions as you can observe from the script below:

```
create table test_b
( T_DATE     date not null,
  PERIOD varchar2(15) not null,
  DEBIT  number,
  CREDIT  number,
  ACCOUNT     varchar2(25),
  DIVISION  varchar2(50),
  SECTOR  varchar2(100),
  UNIT varchar2(100))
partition by range (T_DATE)
(partition QT1  values less than
(to_date('01-APR-2009', 'dd-mon-
yyyy')),
partition QT2 values less than
(to_date('01-JUL-2009', 'dd-mon-
yyyy')),
partition QT3  values less than
(to_date('01-OCT-2009', 'dd-mon-
yyyy')),
partition QT4  values less than
(to_date('01-JAN-2010', 'dd-mon-
yyyy')));
```

Then, we create the 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. The script is showed below:

```
  create table TEST_C
(  T_DATE     date not null,
  PERIOD varchar2(15) not null,
  DEBIT  number,
  CREDIT  number,
  ACCOUNT     varchar2(25),
  DIVISION  varchar2(50),
  SECTOR   varchar2(100),
  UNIT varchar2(100))
partition by range (T_DATE)
subpartition by list (DIVISION)
(partition QT1  values less than
(to_date('01-APR-2009', 'dd-mon-
yyyy'))
(subpartition QT1_OP values
('a.MTN','b.CTM','c.TRS','d.WOD','e.DM
A'),
 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-2009', 'dd-mon-
yyyy'))
(subpartition QT2_OP values
('a.MTN','b.CTM','c.TRS','d.WOD','e.DM
A'),
 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-2009', 'dd-mon-
yyyy'))
(subpartition QT3_OP values
('a.MTN','b.CTM','c.TRS','d.WOD','e.DM
A'),
 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-2010', 'dd-mon-
yyyy'))
(subpartition QT4_OP values
('a.MTN','b.CTM','c.TRS','d.WOD','e.DM
A'),
 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')));
```

After loading data in these two partitioned tables we gather statistics with the package DBMS_STATS. 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 table 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 T_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 T_DATE than table A non-partitioned.

Table 1 Comparative analysis results for simple queries

TABLE:	TES	TEST_B	TEST_C

	T_A					
QUERRY:	Not partitioned	Partition range by date on column "T_DATE"		Partition range by date with four list partions on column "DIVISION"		
		Without partition clause	Partition (QT1)	Without partition clause	Partition (QT1)	Sub-partition (QT1_AFO)
Select * from TEST_	**170**	184	-	346	-	-
where extract (month from T_date) =1;	183	197	91	357	172	172
… and division='h.AFO divizii'	173	199	91	**25**	**12**	172
select sum(debit) TD, sum(credit) TC from test_ where extract (month from t_date) =1 and division='h.AFO divizii'	173	199	91	**25**	12	172
… and unit ='MTN'	173	199	91	350	172	172

Note: The grey marked ones have the best execution cost of the current query

3.3 Using hints and indexes

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 statistics 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 [1]. 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 BI systems, 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 [1]. 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 determinates 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 2 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.

Table 2. Comparative analysis results using hints

TABLE:	**TEST_A**	**TEST_B**		**TEST_C**		
QUERRY**:**	Not partitioned	Partition range by date on column "T_DATE"		Partition range by date with four list partions on column "DIVISION"		
		Without partition clause	Partition (QT1)	Without partition clause	Partition (QT1)	Sub-partition (QT1_AFO)
select /*+ *USE_HASH (a u)*/* a.*, u.location,u.country, u.region from TEST_t a , d_units u where a. unit=u. unit and extract (month from T_date) =1	176	182	95	294	152	152
… and a.division = 'h.AFO divizii'	175	181	94	28	19	150
…/*+ *USE_NL (a u)*/*	281	287	151	170	18	171
…/*+ *USE_NL (a u)*/* *--WITH INDEXES*	265	235	110	120	12	143
…/*+ *USE_MERGE (a u)*/* *--WITH INDEXES*	174	180	94	27	18	150
…and u. unit ='MTN'	174	180	94	151	19	150
…/*+ *USE_NL (a u)*/*	174	180	94	21	18	150
…/*+ *USE_NL (a u)*/* *--WITH INDEXES*	172	178	86	21	12	144
…/*+ *USE_MERGE (a u)*/* *--WITH INDEXES*	172	178	86	21	12	144

The significant improvement is in sub partitioned table in which the cost of execution was drastically reduce at only 12 points compared to 176 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 Using analytical functions

In the latest versions in addition to aggregate functions Oracle implemented analytical functions to help developers building decision support reports [1]. 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 Oracle divides the set of records 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. 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 [4]

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 [1].

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:

```
   select period, division, sector,
unit, debit,
   avg(debit) over (partition by
division, sector, unit
   order by extract (month from
t_date)
   range between 3 preceding and 3
following) avg_neighbours
   from test_a
```

3.4. Object oriented implementation

A modern RDBMS environment, such as Oracle Database 10g, supports the object type concepts that can be used to specify the multidimensional models (MD) constrains. An object type differs from native SQL data types in that it is user-defined, and it specifies both the underlying persistent data (attributes) and the related behaviours (methods).

The object type is an object layer that can map the MD model over the database level, but data is still stored in columns and tables. Internally, statements about objects are still basically statements about relational tables and columns, and you can continue to work with relational data types and store data in relational tables. But we have the option to take advantage of object-oriented features too. Data persistency is assured by the object tables, where each row of the table corresponds to an instance of a class and the table columns are the class's attributes. Every row object in an object table has an associated logical object identifier. There can be use two types of object identifiers: a unique system-generated identifier of length 16 bytes for each row object assigned by default by Oracle in a hidden column, and primary-key based identifiers specified by the user and in which we have the advantage of enabling a more

efficient and easier loading of the object table [1].

The object oriented implementation can be used to reduce the execution cost by avoid the multiple joins between the fact and the dimension tables. For exemplification we'll present here only the classes of management unit dimension (table unit in our previous examples) and the fact table – balance_R.

We'll use a super type class to define the management unit dimension. We'll call it as UnitSpace_OT. For hierarchical levels of the dimension, as you can observe, there are two major hierarchies:

- geographical locations (H1): zone->region->country->location-> unit
- organizational and management (H2): division->sector-> unit.

So, the final object in both hierarchies is unit which will have two REFs, one for H1 and one for H2 hierarchies. The script is shown below:

For the first hierarchy (H1):

```
create or replace type unitspace_ot as object (unitspace_id number, unitspace_desc varchar2 (50), unitspace_type varchar2 (50)) not instantiable not final;
create or replace type zone_o under unitspace_ot (/*also add other attributes and methods*/) not final;
create or replace type region_o under unitspace_ot (zone ref zone_o /*also add other attributes and methods*/) not final;
create type country_o under unitspace_ot (region ref region_o /*also add other attributes and methods*/) not final;
create type location_o under unitspace_ot (country ref country_o /*also add other attributes and methods*/) not final;
The second hierarchy (H2):
create or replace type division_o under unitspace_ot (/*also add other attributes and methods*/) not final;
create type sector_o under unitspace_ot (division ref division_o /*also add other attributes and methods*/) not final;
create or replace type unit_o under unitspace_ot (sector ref sector_o, location ref location_o /*also add other attributes and methods*/) final;
```

The orders' fact is implemented also as an object type INSTANTIABLE and NOT FINAL:

```
create or replace type balance_r as object
    ( T_DATE date not null,
      PERIOD varchar2(15) not null,
      DEBIT number,
      CREDIT number,
      ACCOUNT varchar2(25),
      DIVISION varchar2(50),
      SECTOR varchar2(100),
      UNIT_ID varchar2(100)));
```

The methods of each MD object type are implemented as object type bodies in PL/SQL language that are similar with package bodies. For example the unit_o object type has the following body:

```
create or replace type body unit_o as
static function f_unit_stat(p_tab varchar2,p_gf varchar2, p_col_gf varchar2, p_col varchar2, p_val number) return number as
  /* the function return the aggregate statistics from fact tables for a specific unit */
  v_tot number;
  text varchar2(255);
  begin
   text:= 'select '|| p_gf || ' ('||p_col_gf||') from '||p_tab||' cd where '||p_col||'= '||p_val;
   execute immediate text into v_tot;
    return v_tot;
  end f_unit_stat;
/*others functions or procedures*/
end;
/
```

We can use this function to get different aggregate values for a specific unit, such as the total amount of quantity per unit or the average value per unit. Data persistency is assured with object tables that will store the instances of that object type, for example:

```
CREATE TABLE unit_t OF unit_o;
```

For example the static function *f_unit_stat* from the *unit_o* class can be use to retrieve the total debit value for each unit:

```
(1) select unit_id, description,
unit_o.f_unit_stat('balance_rt', 'SUM',
'debit', 'unit_id', unit_id) total
from unit_t;
```

instead of using the join between the unit_t and the balance_r tables:

```
(2) select t.unit_id, t.description,
SUM(debit) total_debit
from unit t, balance_r b
where t.unit_id=b.unit_id
```

We'll use for testing two types of tables: object tables (unit_t and balance_rt) and relational tables (unit and balance_r). The cardinality of these tables is about 100000 records in balance and about 100 in unit tables.

We analyze the impact of calling the function in the SQL query in different situations, as we present in the following table:

Table 3. The execution costs of the queries

No	Query	Cost
	select unit_id, description, unit_o.f_unit_stat('balance_rt', 'SUM', 'debit', 'unit_id', unit_id) total from unit_t	35
	select t.unit_id, t.description, SUM(debit) total from unit t, balance_r b where t.unit_id=b.unit_id	171
	Example (1) with an index on unit_id on both object tables	30
	Example (2) with an index on unit_id on both relational tables	171
	Example (1) with an index on unit_id on both relational tables with *use_nl* hint	79

We analyze the execution cost of the function's query, it has 35 units, but the SQL Tuning Advisor makes a recommendation: "consider collecting statistics for this table and indices". We used DBMS_STATS package to collect statistics from both object and relational tables. Then we re-run the queries and observe the execution plans; there is no change and the Tuning Advisor doesn't make any recommendation.

By introducing these types of functions we have the following *advantages*:

- The function can be used in many reports and queries with different types of arguments, so the code is re-used and there is no need to build another query for each report;
- The amount of joins is reduced; the functions avoid the joins by searching the values in the fact table;
- Soft parsing is used for the function's query execution instead of hard parsing in the case of another SQL query.

The main *disadvantage* of the model is that the function needs to open a cursor to execute the query which can lead to an increase of PGA resources if the fact table is too large. But the execution cost is insignificant and does not require a full table scan if an index is used on the corresponding foreign key attribute.

Through an ETL (extract, transform and load) process data is loaded into the object tables from the transactional tables of the ERP organizational system. This process can be implemented also through object types' methods or separately, as PL/SQL packages. Our recommendation is that the ETL process should be implemented separately from the object oriented implementation in order to assure the independency of the MD model.

4 Conclusions

The virtual data warehouse is based on a set of objects like views, packages and program units that extracts, joins and aggregates rows from the ERP system's database. In order to develop a BI system we have to build analytical reports based on this virtual data warehouse. But the performance of the whole system can be affected by the data extraction process which is the major time and cost consuming job. A possible solution is to apply the optimization techniques that can improve the performance. Some of these techniques are presented in this paper. The results that we've obtained are relevant for decreasing the execution cost. Also, for developing BI reports an important option is to choose analytic functions for predictions, subtotals over current period, classifications and ratings. Another issue discussed was the OO implementation which is very flexible and offer a very good representation of the business aspects that are essential for BI projects. Classes like dimensions or fact tables can be implemented together with the attributes and methods, which can be a very good and efficient practice concerning both the performance and business modelling.

References

[1] Oracle Corporation - *Database Performance Tuning Guide 10g Release 2 (10.2),* Part Number B14211-01, 2005

[2] Ion Lungu, Manole Velicanu, Adela Bâra, Vlad Diaconita, Iuliana Botha – Practices for designing and improving data extraction in a virtual data warehouses project, Proceedings of ICCCC 2008, International conference on computers, communications and control, Baile Felix, Oradea, Romania, 15-17 May 2008, pag 369-375, published in International Journal of Computers, Communications and Control, volume 3, 2008, ISSN 1841-9836

[3] Adela Bâra, Ion Lungu, Manole Velicanu, Vlad Diaconița, Iuliana Botha – Improving query performance in virtual data warehouses, WSEAS TRANSACTIONS ON INFORMATION SCIENCE AND APPLICATIONS, May 2008, ISSN: 1790-0832

[4] Ion Lungu, Adela Bara, Anca Fodor - Business Intelligence tools for building the Executive Information Systems, 5thRoEduNet International Conference, Lucian Blaga University, Sibiu, June 2006

[5] Donald K. Burleson, Oracle Tuning, ISBN 0-9744486-2-1, 2006