# Improving Query Performance in Distributed Database

**Alexandru Boicea, Florin Radulescu, Ciprian-Octavian Truica, Loredana Urse**

*Faculty of Automatic Control and Computers, University Politehnica of Bucharest, Bucharest, Romania (e-mail: alexandru.boicea@cs.pub.ro, florin.radulescu@cs.pub.ro, ciprian.truica@cs.pub.ro, loredana.urse@cti.pub.ro)*

**Abstract**: In a rapidly growing digital world there is the possibility to query and discover data, but the most important issue is what resources are needed and how quickly data can be accessed. For several years ago, the grid systems, cloud systems and distributed database systems have replaced independent databases, because their computing power is much higher. In the case of distributed databases, stored in different nodes of a network, there may be chosen more channels of communication between nodes and therefore different time costs. In this paper a method is presented for selecting optimal routes between the nodes that are distributed to the system, depending on the system parameters, network characteristics, available resources and the volume of data that is to be transferred. Also, a method is shown to improve the time cost for multiple queries in distributed databases using the caching technique. To test and validate the method, a database to a web application was used in order to manage a chain of stores. Several scenarios were created for queries and the execution time for each scenario in part was measured through an interface designed specifically for testing.

*Keywords:* Distributed database, node selection, feasible distance, query execution, caching method.

## 1. INTRODUCTION

Distributed computing technology has been used in all sorts of database applications. It provides transparent access to distributed computing resources like processing capabilities and storage capacity. A user sees a large, single computer even if the resources and databases are geographically distributed in the world-wide networks.

Query processing is an important concern in the field of distributed databases and also grid databases. The main problem is if a query can be decomposed into sub-queries that require operations in geographically separated databases, the sequence and the sites must be determined for performing this set of operations, such that the operating cost (communication cost and processing cost) for processing this query should be minimized (Alom et al., 2009).

Along with the expanding applications data queries are increasingly complex, so query optimization in both of them becomes a difficult task to accomplish. Many people have tried to solve this issue by proposing different algorithms in order to minimize the costs and the response time which are associated with obtaining the answer to queries.

In a distributed relational database more strategies are applicable for processing and integrating data, e.g.: data centralizing, data partitioning (fragmentation), full data replication and partial data replication.

Although using data replication strategy increases the database efficiency, this benefit comes with some costs, which could potentially be high: storage cost and communication cost (Rahimi et al., 2010).

In their work, Rahimi et al., arrived to the conclusion that optimizing queries in distributed database is different from the one in database grid systems. The main differences are consequences of autonomy and heterogeneity of databases in distributed systems. Therefore optimizing queries in distributed database systems is more challenging than optimizing queries in grid systems.

A query execution plan consists of operators and their allocation to servers. Standard physical operators, usually implementing the data model's algebra, are used to process data and to consolidate intermediary and final results. Communication operators realize the transfer, by sending and receiving data from one server to the other (Bressan, 2009).

In Section 2 of this paper a method is presented to estimate the cost of data transfer between nodes of a distributed database system and also to choose the optimal route based on minimum cost.

Section 3 shows a method to optimize the queries in a distributed database using the distributed caching. This method is based on temporary replication of data from different database partitions to process on the client machine.

In the last section the conclusions are presented based on experimental results shown in the previous sections.

## 2. COST ESTIMATING FOR DATA TRANSFER IN DISTRIBUTED DATABASES

To process a distributed query, some data in the distributed database must be transferred to the local machine. Data packets transmitted on the network are time consuming, especially for large volumes of data. The data transfer

between nodes of a network is a cost-recovery, so there must be chosen routes with minimal costs for data transfer.

*2.1 Criteria selecting of the nodes*

In distributed systems Query Optimizer it must be decided at which site each operation is to be executed using node selection algorithm, an issue which is presented in the next section. To accomplish this, the following semi-dynamic optimization criteria were taken into consideration:

- Nodes selection of the distributed system;
- Pre-optimizing the query;
- Testing if the execution runs as expected during the optimization in query execution;
- If execution shows severe deviations, a new query plan should be computed for all parts that haven't been executed;
- The plans should be created in multiple stages;
- Global query optimizer should focus on data transfer, where the different intermediate results should be computed and then shipped;
- Local query optimizers decide on local query algorithms, indexes, partitioned tables for delivering the intermediate result;
- The query can be optimized in two steps: at compile time (join order, methods, access, etc) and/or during query execution.

The final step is sending the plan to the execution engines at the sites decided by the optimizer and also being enhanced by the local optimizer.

Plan specifies precisely how the query is to be executed. The nodes are operators, and every operator carries out one particular operation. The edges represent consumer-producer relationships of operators.

Plan Refinement transforms the plan into an executable one. This transformation involves the generation of an assembler-like code to evaluate expressions and predicates.

*2.2 Estimating cost of data transfer*

To minimize the cost of data transfer between nodes in a distributed database, it is necessary to calculate the costs of transfer between neighbouring nodes, after which there will be chosen the route with minimum cost.

The most important factors to take into consideration are: bandwidth, delay, load, access and transmission costs, CPU and memory capabilities and free system capabilities.

An adaptive algorithm is proposed to achieve this cost minimization. This algorithm is similar with Diffusing Update Algorithm (DUAL), used by the EIGRP routing protocol, with a combination of backtracking or min-max algorithm to search in the depth for the optimal overall cost (Garcia, 2014).

The formula uses three separate tables for route calculation:

- *Neighbour table*: contains information on all the directly connected nodes and a timer is used to check whether the node is available or not.
- *Topology table*: contains the cost information of all the nodes to any destination. This information is received from the neighbour table. The primary (successor) and secondary (feasible successor) routes to a node will be determined from this table. Moreover, each entry in this table contains the following fields:
  - FD (Feasible Distance) = the cost to the desired node;
  - RD (Reported Distance) = the cost to the desired node reported by a neighbour node (this is used to calculate the FD);
  - The status of the route to that node:
    - Active : the route is being calculated (down);
    - Passive: the route is up.
  - CPU - available CPU;
  - Memory - available memory;
  - Free memory - the free percent of memory available (it is a combination of CPU and MEMORY from that node);
  - Access cost;
  - Transmission cost.
- *Routing table*: contains the best route(s) to a destination (the lowest metric cost). It is populated from the successors from the topology table.

Those tables are locally created and updated periodically (from hour to hour, for example) using the information exchanged between the nodes.

The node selection algorithm evaluates the data received from other routers in the topology table and calculates the primary (successor) and secondary (feasible successor) routes.

The primary path is usually the path with the lowest metric to reach the destination, and the redundant path is the path with the second lowest cost (if it meets the feasibility condition). There may be multiple successors and multiple feasible successors. Both successors and feasible successors are maintained in the topology table, but only the successors are added to the routing table and used to route packets.

Feasible Distance (FD) is sum of Reported Distance (RD) of all successors between start node and final node from a route.

For a route to become a feasible successor, its RD must be smaller than the FD (RD<FD). If this feasibility condition is met, there is no way that adding this route to the routing table could cause a loop. If all the successor routes of a destination fail, the feasible successor becomes the successor and is immediately added to the routing table. If there is no feasible successor in the topology table, a query process is initiated to look for a new route.

The formula used to calculate the best metric is the following:

$$Metric = \left[K1 * Bandwidth + \frac{K2 * Bandwidth}{256 - Load} + K3 * Delay\right] * K4 * \left[\frac{(Access * K5 + Transmission * K6)}{10\text{^}3} + \frac{(CPU * Memory * Free)}{10\text{^}8}\right] \quad (1)$$

where:

- Bandwidth - the minimum bandwidth along the path is selected;
- Load - a number between 1 and 255, meaning how saturated is the network;
- Delay - the number of nodes that must be traversed to the destination node;
- Access - how expensive are access costs to access the data;
- Transmission - how expensive are the transmission costs for transmitting the data from one node, to the neighbour node;
- CPU (capabilities) - the performance of the node's CPU;
- Memory (capabilities) - how fast it perform I/O operations for read/write the data;
- Free (system capabilities) - how much of the system is free;
- K1 - K6 are various constants that can be changed in the entire network of distributed database. At the initial moment the values are: K1 = 1, K2 = 0, K3 = 1, K4 = 1, K5 = 20, K6 = 2.

The formula is based on a cost-based algebraic optimization.

The node selection algorithm can support unequal cost load balance (like DUAL) if a variable (variance) L is taken into consideration.

Using this variable to match a feasible condition, the successor cost multiplied by the variable L should be greater than the other nodes feasible distance (FD * L >= FD successor). The L variable is an integer number greater or equal than 1.

The nodes from which to receive the data (it is known that the data is present there) and to send the data to the nearest node can be chosen using the given metric for the elements mentioned above.

The data received must be processed in parallel, using the unequal cost load balance method, based on the feasible successors metric as follows: the data requested can be found on multiple node, is fragmented (partitioned, sub-partitioned, has snapshots, indexes, etc.) and can be split easily using the proportion from the metrics.

*2.3 Example of estimating the cost of data transfer*

For exemplification, the following example of a distributed database with a user node (A0), processing a query, and three nodes of remote databases (A1, A2, A3) was used.

The percentage of resources allocated for the three nodes are: P1 = 60%, P2 = 25% and P3 = 15%.

Fig.1 shows the possible routes between nodes and connections associated RD.

The result returned from the one node is afterward transmitted to another where it can find additional data to be joined, by the global optimizer and the algorithm recursively repeats itself until the final result is returned.

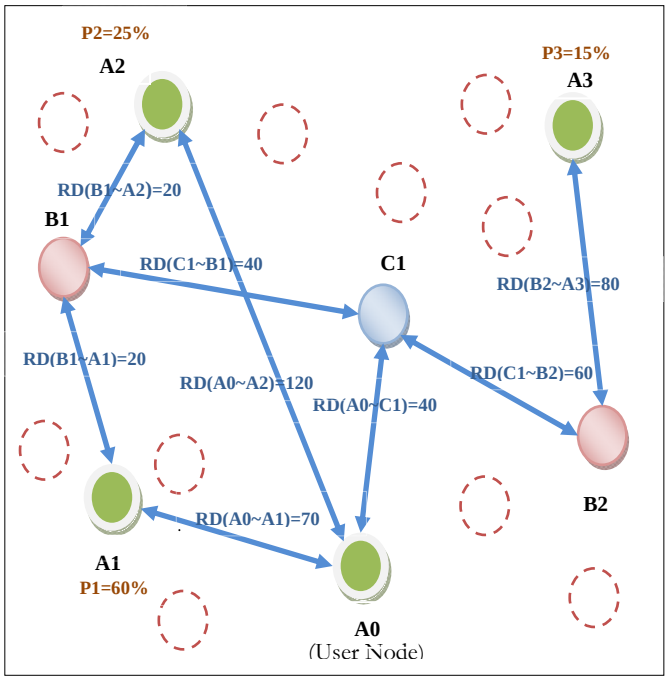


Fig. 1. Possible routes and RD costs for a distributed database.

By applying the above formula, there can be built a RD cost matrix between nodes, as shown in Table 1.

**Table 1. Reported Distance (RD) between nodes.**

| Nodes | A0 | A1 | A2 | A3 | B1 | B2 | C1 |
|---|---|---|---|---|---|---|---|
| **A0** | 0 | 70 | 120 | * | * | * | 40 |
| **A1** | 70 | 0 | * | * | 20 | * | * |
| **A2** | 120 | * | 0 | * | 20 | * | * |
| **A3** | * | * | * | 0 | * | 80 | * |
| **B1** | * | 20 | 20 | * | 0 | * | 40 |
| **B2** | * | * | * | 80 | * | 0 | 60 |
| **C1** | 40 | * | * | * | 40 | 60 | 0 |

The Reported Distance and Feasible Distance are calculated from the Topology Table by the global optimizer scheduler, which says on which nodes the interest data can be found.

To find all possible routes between two nodes it can use Breadth-First Search (BFS) algorithm. The BFS begins at a root node and inspects all the neighbouring nodes.

Calculating Feasible Distances from user node (A0) to nodes A1, A2, A3, according to possible routes, are following:

✓ A0=>A1

Route(A0-A1) : FD= 70;

Route(A0-C1-B1-A1) : FD= 40+40+20= 100;

Route(A0-A1) has the minimum cost route with FD=70.

✓ A0=>A2

Route(A0-A2) : FD= 120;

Route(A0-A1-B1-A2) : FD= 70+20+20= 110;

Route(A0-C1-B1-A2) : FD= 40+40+20= 100;

Route(A0-C1-B1-A2) has the minimum cost route with FD=100.

- ✓ A0=>A3

Route(A0-C1-B2-A3) : FD= 40+60+80= 180;

Route(A0-A1-B1-C1-B2-A3) : FD= 70+20+40+60+80= 270;

Route(A0-C1-B2-A3) has the minimum cost route with FD=180;.

Fig. 2 show the graph of connections between nodes and minimum cost routes between user node (A0) and remote database nodes (A1, A2, A3).

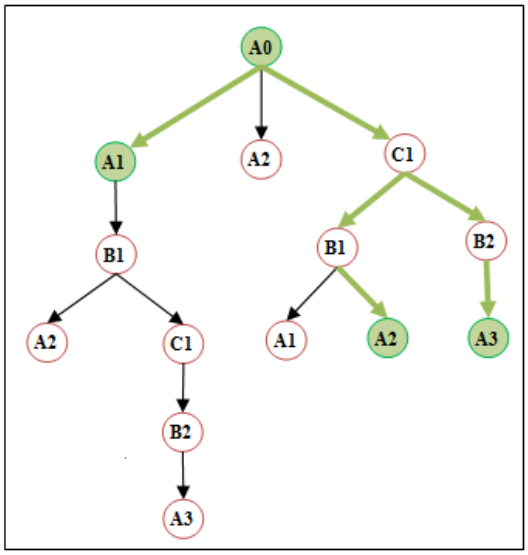


Fig. 2. Graph of connections between nodes.

After the node selection and local query execution, the next step is to go back to the global optimizer and search the next available nodes to find the interest data.

In Fig. 3 there are marked routes with minimal costs for transferring data from distributed databases on the node that processes the user query.

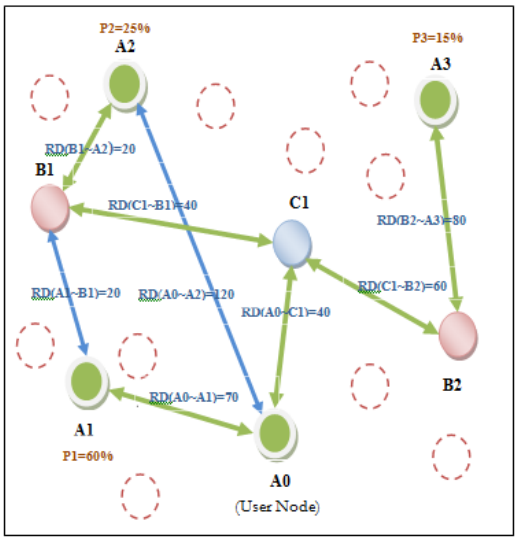


Fig. 3. The routes with minimal cost of data transfer.

Routes that have minimal costs are show in Table 2.

**Table 2. Selected routes with minimal costs.**

| Nodes | Route | Total Cost |
|---|---|---|
| **A0 =>A1** | A0-A1 | 70 |
| **A0 =>A2** | A0-C1-B1-A2 | 100 |
| **A0 =>A3** | A0-C1-B2-A3 | 180 |

The above example is a simple one, but this can be more complicated if different data sets are required from different nodes/sources, different proportions of allocation is necessary to be taken into consideration and the queries have many join conditions.

The proposed method is time consuming and is suited for large amounts of data requested. All the tables, neighbour, topology and route table are updated periodically (e.g. every hour).

## 3. QUERY PROCESSING IN DISTRIBUTED DATABASE

One of the most relevant cases to query processing on the distributed databases is when multiple organisations agree to share their date to perform a common task. For example, a multinational company has offices in different continents so it needed to exchange data between them, in order to strengthen the central level.

This is achieved by using well-established techniques like fast access remote resources and parallel query processing.

In general, query processing in grid systems is similar to query processing in distributed databases in the sense that in both domains the queries are submitted over a global schema and the data is fetched from the sources through packaging.

The main steps for query processing are:

- query decomposition
- query optimization
- data localization
- query execution

Queries are expressed using different declarative query languages like SQL, OQL, XQUERY. Once a query is submitted, the first step is to parse it and check the syntax. Then the query is decomposed into an internal representation using algebra expressions.

Query Rewrite transforms a query in order to carry out optimizations that are good regardless of the physical state of the system (elimination of redundant predicates, interpretation of sub-queries, expressions simplification).

Query Optimizer carries out optimizations that depend on the physical state of the system. It decides which index, which method, and in which order to execute operations of a query.

Data localization determines which fragments are involved in the query and thereby transforms the distributed query into a fragment query in order to localize its data.

The last step is to find the most efficient candidate execution plan and execute the query.

*3.1 Improving query performance through caching method*

Caching method is used to create a system for queries processing that manipulate temporary database to optimize multiple queries in a distributed database.

The terms "buffer" and "cache" tend to be used interchangeably but they represent different things. A buffer is a temporary memory and is used traditionally as an intermediate temporary store for data between a fast and a slow entity.

Fundamentally, caching provides a performance increase by repeated data transfers from one server to another. While a caching system may realize a performance increase upon the initial transfer of a data item, this performance increase is due to buffering occurring within the caching system.

Caching and replication is an effective approach for lowering overhead and increasing accessibility and performance in distributed systems. It is used in a range of environments including web page distribution, multimedia distribution and distributed object location. Large scale multimedia distribution and video-streaming, in particular, can gain significant advantages from effective caching and replication due to their high bandwidth, low latency requirements (Tysona et al., 2009).

Replication solution also provides independence of the servers from different locations, which can work independently with local data until the synchronization is done (Truica et al., 2013).

The basic idea of caching is to create and implement a strategy of splitting the distributed data, transfer and processing them on the local machine, and storing them for a certain period of time in view of subsequent queries. Thus, the cost of data transmission will not have much impact in running queries or query response time becomes significantly lower. For this, temporary tables or materialized views that contain the columns desired from the users and which can be updated on request can be created.

Synchronizing data between tables in distributed databases and temporary tables which are on the local machine can be done at predefined time intervals, through background processing.

For example, triggers or stored procedures can be created that make real-time data updating and execute at required time interval.

The connection between a remote database and local database is done through a database link.

Caching is particularly useful in distributed systems because it reduces network traffic and improves response time.

Detailing of the caching scheme for three distributed databases is:

- The user connects to the application on the local database (DB1) and can manually enter a query that requires information from tables on other databases (DB2, DB3);
- Depending on the criteria entered by the user, the query is built;
- Query is executed on the local database (an initial non-optimized query);
- The execution time of the query is measured (non-optimized);
- Then, on the local machine, temporary tables/ materialized views are created, with data from the remote database tables;
- Run the query again with the data from the temporary tables created (optimized query);
- Execution time of the new query is measured (optimized query);

*3.2. Experimental results for caching method*

For testing, there was used an architecture that consists of a local server on which was installed a local database (DB_local), two servers with the local network connection on which were installed two databases (DB2, DB3) and a server with remote connection on which it was installed a database having Internet access (DB_remote).

The Windows 7 operating system was chosen for development. The tests were implemented in PHP5 and run on an Apache Web Server. All the DBMSs in the testing scenario were Oracle Database 11g.

A PHP application was created that contains an interface for query editing and for executing the test scenarios.

The test databases were taken from a web application for managing a chain of shops and for storing the following information:

DB_local: Sales

DB1: Users, Roles

DB2: Customers

DB_remote: Suppliers, Products, Stores

The entity-relationship diagram for the test database is presented in Fig. 4.

Caching algorithm has the following inputs/outputs:
Input: Query
Output: Data query

For each query the following steps are executed:

1. It checks if the entered query can be executed on the local database tables (using PHP and Oracle);

2. If the query can run on tables in the local database, the data is selected from tables and the results returned to the user (via PHP);

3. If the query cannot be executed entirely on the tables in the local database, then queries are run on data that are not found in local tables and results are stored in the local database tables (via Apache sever), after which the entire query is executed and the results returned to the user (via PHP);

4. If the query cannot be executed on any of the tables in the local database, then queries are run for tables of distributed databases, the results are stored in the local database tables (via Apache sever) and also the results are returns to the user (via PHP);

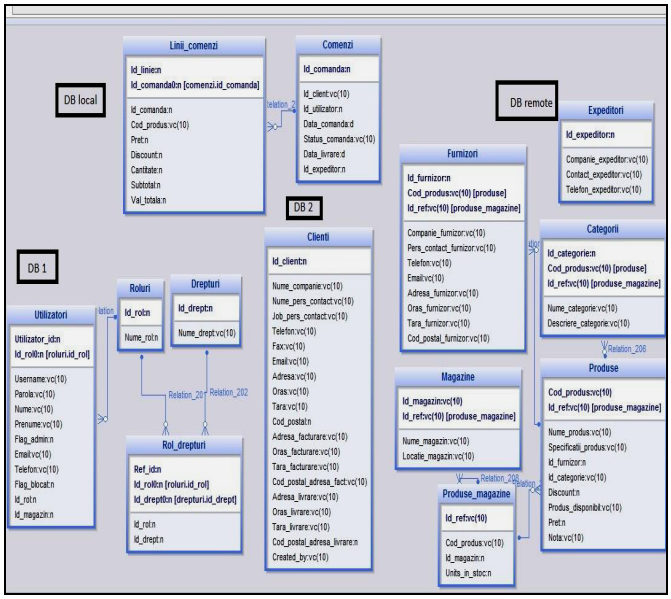


Fig. 4. Entities relationship diagram for the distributed database.

In the application were created several test scenarios for queries that return between 1000-500000 records. For each scenario were performed a minimal 10 tests for better measuring execution time.

The main test scenarios were:

S1: list of sales made by a user (DB_local) and information about products sold (DB_ remote) => it returns on the local database (DB_local) temporary tables that contain all columns created on the tables in remote database (DB_remote).

S2: list of sales made by a user (DB_local), information on products sold (DB_ remote) and information about the user (DB1) => it returns on the local database (DB_local) temporary tables with data on products sold.

S3: list of sales made by a user (DB_local), customer information (DB2) and information about the user (DB1) => it returns on the local database (DB_local) temporary tables with data about products and users.

S4: list of sales made by a user (DB_local), information on products sold (DB_remote), information about the user (DB1) and information about clients (DB2) => it returns on the local database (DB_local) only a temporary table with data about products.

S5: list of sales made by a user (DB_local), information on products sold (DB_remote), information about the user (DB1) and information about customers (DB2) => it returns on the local database (DB_local) a temporary table with data about user.

S6: list of sales made by a user (DB_local), information about products sold (DB_remote), information about the user (DB1) and information about customers (DB2) => it returns on the local database (DB_local) a temporary table with data about customers.

S7: list of sales made by a user (DB_local), information about products sold (DB_remote), information about the user (DB1) and information about clients (DB2) => it returns on the local database (DB_local) temporary tables with data about products and customers.

S8: list of sales made by a user (DB_local), information about products sold (DB_remote), information about the user (DB1) and information about customers (DB2) => it returns on the local database (DB_local) temporary tables with data about products and customers.

S9: list of sales made by a user (DB_local), information about products sold (DB_remote), information about the user (DB1) and information about customers(DB2) => it returns on the local database (DB_local) temporary tables with data about users and customers.

S10: list of sales made by a user (DB_local), information about products sold (DB_remote), information about the user (DB1) and information about customers (DB2) => it returns on the local database (DB_local) temporary tables with data about products, users and customers.

Depending on the scenario, the query was fragmented vertically in more queries on distributed database, as shown in Table 3.

**Table 3. Fragmentation matrix on distributed database queries.**

| Scenario/DB | DB_local | DB1 | DB2 | DB_remote |
|---|---|---|---|---|
| **S1** | 1 | 0 | 0 | 1 |
| **S2** | 1 | 1 | 0 | 1 |
| **S3** | 1 | 0 | 1 | 1 |
| **S4** | 1 | 0 | 1 | 1 |
| **S5** | 1 | 1 | 1 | 1 |
| **S6** | 1 | 1 | 1 | 1 |
| **S7** | 1 | 1 | 1 | 1 |
| **S8** | 1 | 1 | 1 | 1 |
| **S9** | 1 | 1 | 1 | 1 |
| **S10** | 1 | 1 | 1 | 1 |

Function that measures the execution time of the query is:

```
CREATE OR REPLACE FUNCTION  T_TIME (P_SQL IN
VARCHAR2)
RETURN NUMBER
IS
    timestart  NUMBER;
    v_cnt      NUMBER;
BEGIN
timestart:=dbms_utility.get_time();

EXECUTE IMMEDIATE ('SELECT COUNT(*) FROM
('||P_SQL||')') INTO  v_cnt;
RETURN round((dbms_utility.get_time()-timestart)
          +300/100,2);
END T_TIME;
```

*3.3 Example of the SELECT command*

For scenario no. 10, on DB_local database, it was created the following SQL command:

- ✓ Without caching method:

```
SELECT  u.username, u.nume, u.prenume, u.email,
u.id_magazin, c.id_comanda, t.nume_companie,
t.nume_persoana_contact, t.telefon,t.email, t.adresa,
t.oras,t.tara, t.cod_postal, c.status_comanda, c.data_livrare,
l.cod_produs, l.pret, l.cantitate,l.subtotal, l.val_totala,
s.nume_produs, s.specificatii_produs, s.produs_disponibil,
s.nota

FROM comenzi c,linii_comenzi l,lore.produse@db_remote s,
utilizatori@db1 u, clienti@db2 t

WHERE  c.id_comanda=l.id_comanda and
c.id_utilizator=u.utilizator_id and u.utilizator_id=833 and
l.cod_produs=s.cod_produs and c.id_client = t.id_client ;
```

- o Execution time: 16.42 sec

- ✓ With caching method:

```
CREATE  TABLE utilizatori_temp@DB_local as SELECT *
FROM utilizatori@db1;
CREATE  TABLE  clienti_temp as SELECT * FROM
clienti@db2;
CREATE  TABLE  produse_temp as SELECT * FROM
lore.produse@db_remote ;

SELECT u.username, u.nume,u.prenume, u.email,
u.id_magazin, c.id_comanda, t.nume_companie,
t.nume_persoana_contact, t.telefon,t.email, t.adresa, t.oras,
t.tara, t.cod_postal  ,c.status_comanda, c.data_livrare,
l.cod_produs, l.pret,l.cantitate, l.subtotal, l.val_totala ,
s.nume_produs, s.specificatii_produs, s.produs_disponibil,
s.nota

FROM  comenzi c, linii_comenzi l, produse_temp s,
utilizatori_temp u, clienti_temp t

WHERE  c.id_comanda=l.id_comanda and
c.id_utilizator=u.utilizator_id and u.utilizator_id=833 and
l.cod_produs=s.cod_produs and c.id_client = t.id_client;
```

- o Execution time: 2.56 sec

*3.4 Example of a scenario using materialized view*

It will be created a materialized view on local database (DB_local) with data about products taken from the remote database (DB_remote):

```
CREATE MATERIALIZED VIEW
db_remote.PRODUSE_ONLINE

QUERY REWRITE AS

SELECT a.id_ref, a.units_in_stoc, b.cod_produs, b.discount,
b.nota, b.nume_produs, b.pret, b.pret- b.discount as
pret_discount, b.produs_disponibil, b.specificatii_produs,
c.id_magazin, c.nume_magazin,
SUM(b.pret*a.units_in_stoc) OVER (PARTITION BY
c.id_magazin ) as venit_magazin, c.descriere_magazin,
d.descriere_categorie, d.id_categorie, d.nume_categorie,
SUM(b.pret*a.units_in_stoc) OVER (PARTITION BY
d.id_categorie) as venit_categorie, e.adresa_furnizor,
e.cod_postal_furnizor, e.companie_furnizor,
SUM(b.pret*a.units_in_stoc) OVER (PARTITION BY
e.id_furnizor) as datorie_furnizor, e.contact_furnizor,
e.email_furnizor, e.id_furnizor, e.oras_furnizor,
e.telefon_furnizor

FROM lore.produse_magazine@db_remote a ,
INNER JOIN lore.produse@db_remote b ON
(a.cod_produs=b.cod_produs)
INNER JOIN lore.magazine@db_remote c ON
(a.id_magazin=c.id_magazin)
 INNER JOIN lore.categorii@db_remote d ON
(b.id_categorie=d.id_categorie)
INNER JOIN lore.furnizori@db_remote e ON
(b.id_furnizor=e.id_furnizor);
```

Then it was tested a complete query using the created view on local database (DB_local):

```
SELECT * FROM produse_online;
```

The average time of execution of the query, without creating the view, was 20.3 sec and the average time obtained by taking the data from the view was 7.68 sec. For all the 10 scenarios the average time obtained during execution without /with caching method is presented in Table 4.

**Table 4. Query execution time before/after optimization.**

| Scenarios | BOP ( Before Optimization ) (sec) | AOP ( After Optimization ) (sec) |
|---|---|---|
| **S1** | 15.16 | 2.31 |
| **S2** | 17.28 | 1.59 |
| **S3** | 15.36 | 10.12 |
| **S4** | 17.79 | 2.11 |
| **S5** | 16.4 | 10.47 |
| **S6** | 16.26 | 10.47 |
| **S7** | 16.77 | 2.27 |
| **S8** | 16.44 | 2.54 |
| **S9** | 16.79 | 11.59 |
| **S10** | 16.42 | 2.56 |

Graphic representation of the experimental results is shown in Figure 5.

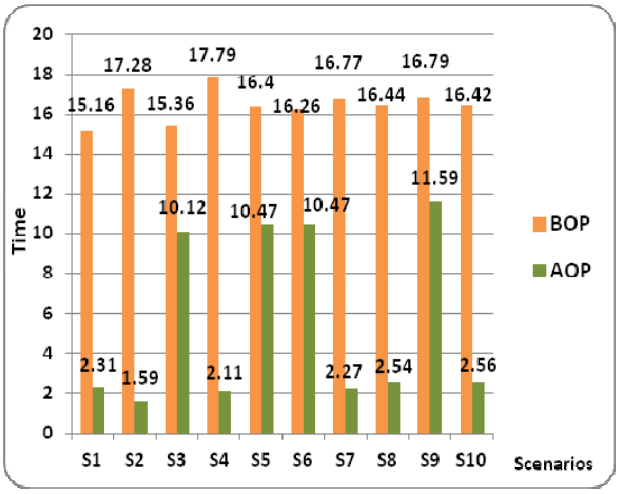


Fig. 5. Query execution time before/after optimization.

## 6. CONCLUSIONS

The optimization of queries in distributed systems is a complex activity that depends on many factors. In a certain percent the optimization is performed by the DBMS but there are situations when the user applications must contain algorithms for improving the queries. Depending on the frequency of query execution in applications with distributed database, it is necessary to improve execution costs of the queries, by transferring data from partitions of the database and processing them on the local machine.

Improving of queries on distributed database through the query caching method involves transferring of the most frequently accessed data from the remote databases to the local database or to another database stored on the same machine. However, additional costs arise by operations of refreshing the cache data, so the method is recommended for databases where data have low cardinality, e.g. for processing data archives.

For the choice of an execution plan for a query, there must be considered the time required for data transfer between different nodes of the distributed database system. It is recommended that the choice of an execution plan to be made by the DBMS because the time performance is not significantly improved by applying some of the node selection algorithms, especially for small data volume.

Using the node selection algorithm has some limitations because of high CPU utilization for calculating the routing. Also, the CPU plays a major role because the execution time for processing the queries is influenced by the processor's frequency and by the number of cores (Truica et al., 2014). For increasing the processing power, a GPU (Graphics Processing Unit) can be used even for non-video or non-graphics applications that imply a serious amount of parallel processing (Munteanu et al., 2015).

Based on the dynamic resource information, which sometimes need to be predicted using prediction algorithms, a scheduler can choose the combination of resources from the available resource pool that is expected to maximize performance (Pop et al., 2011).

## REFERENCES


Alom, B.M., Henskens F. and .Hannaford, M..(2009), Query processing and optimization in distributed database systems, *International Journal of Computer Science and Network Security (IJCSNS'09);*

Bressan, S.,(2009), Distributed query optimization, *Encyclopedia of Database Systems*, pp 908-912

Garcia Luna Aceves, J.J.(2014), *Diffusing Update Algorithm, Wikipedia;*

Munteanu, G., Mocanu, S. and Saru, D.(2015), GPGPU optimized parallel implementation of AES using C++AMP, *Journal of Control Engineering and Applied Informatics (CEAI),* Vol.17, No.2, 2015;

Pop, F. and Cristea V.(2011), Scheduling optimization based on resources state prediction in large scale distributed systems, *Journal of Control Engineering and Applied Informatics (CEAI),* Vol.13, No.4, 2011.

Rahimi, S.K., Haug, F.S.,(2010), Distributed database management systems: a practical approach, *Wiley Publication*, ISBN: 047040745X, IEEE Computer Society;

Truica, C.O. , Boicea, A. and Radulescu, F. (2013), Asynchronous replication in Microsoft SQL Server, PostgreSQL and MySQL, *International Conference on Cyber Science and Engineering (CyberSE'13);*

Truica, C.O., Boicea, A. and Radulescu, F. (2014), Performance time for e-learning applications with multiple databases, *International Scientific Conference eLearning and software for Education (ELSE'14);*

Tysona, G., Mauthea, A., Kauneb S., Mua M. And Plagemann T.,(2009), Corelli:a peer-to-peer dynamic replication service for supporting latency dependent content in community networks, *Multimedia Computing and Networking Conference(MMCN'09);*