# A Query Cache Tool for Optimizing Repeatable and Parallel OLAP Queries

[1] Ricardo Jorge Santos, [1,2] Jorge Bernardino

[1] CISUC – Centre of Informatics and Systems of the University of Coimbra – Portugal
[2] ISEC – Superior Institute of Engineering of Coimbra – Portugal
lionsoftware.ricardo@gmail.com, jorge@isec.pt

**Abstract.** On-line analytical processing against data warehouse databases is a common form of getting decision making information for almost every business field. Decision support information oftenly concerns periodic values based on regular attributes, such as sales amounts, percentages, most transactioned items, etc. This means that many similar OLAP instructions are periodically repeated, and simultaneously, between the several decision makers. Our Query Cache Tool takes advantage of previously executed queries, storing their results and the current state of the data which was accessed. Future queries only need to execute against the new data, inserted since the queries were last executed, and join these results with the previous ones. This makes query execution much faster, because we only need to process the most recent data. Our tool also minimizes the execution time and resource consumption for similar queries simultaneously executed by different users, putting the most recent ones on hold until the first finish and returns the results for all of them. The stored query results are held until they are considered outdated, then automatically erased. We present an experimental evaluation of our tool using a data warehouse based on a real-world business dataset and use a set of typical decision support queries to discuss the results, showing a very high gain in query execution time.

## 1 Introduction

Over the last decades, data warehouses have become excellent decision-support resources for almost every business area. Decision making information is mainly obtained through usage of tools performing On-Line Analytical Processing (OLAP) against data warehouse databases. Because these databases usually store the whole business history, they frequently have a huge number of rows, and grow to gigabytes or terabytes of storage size, making query performance one of the most important issues in data warehousing.

The author in [21] refers that standard decision making OLAP queries which are executed periodically at regular intervals are, by far, the most usual form of obtaining decision making information. This implies that this kind of information is usually based on the same regular SQL instructions. This makes it relevant and important to optimize the performance of predefined decision support queries, which would be executed repeatedly at any time, by a significant number of OLAP users.

Most research proposals for optimizing parallel and repeatable query execution focus on issues such as data and hardware balancing, to take advantage of multi-threading and multi-core processors [7, 11]. The proposed solutions are somewhat complex and expensive. In this paper, we propose a solution at the data and SQL level, which is farther more simple, understandable and inexpensive.

Our proposal consists on a method for speeding up the execution of two types of queries: periodically repeatable queries, which keep their original OLAP instruction; and two or more similar query instructions which are executed simultaneously. This is done by storing the latest results of the frequently used OLAP queries. Therefore, only the most recent factual data is used for processing incremental results, which will be joined with the previous results in order to supply the OLAP queries' response. Our proposal also avoids spending time and resources of the DBMS in processing simultaneously similar OLAP instructions. This is done by looking into the query cache, for every OLAP query to be executed, to see if there is any similar query being executed at the same time. If there is, the latest user is put on hold and will receive the results as soon as it finishes processing for the first user who started the execution. As it can be seen in the results provided in the experimental evaluation, this method provides very high gains in query response time and resource consumption for repeatable and parallel querying, for several number of simultaneous users.

The remainder of this paper is organized as follows. Section 2 presents our proposal, describing how the query caching method works and is used in the Query Cache Tool. In Section 3 we present an experimental evaluation and discuss its results. Section 4 presents related work on parallel query execution, query caching and other research related with the solutions used in our proposal. Finally, section 5 presents conclusions and future work.

## 2 The Query Cache Tool

Traditionally, it has been well accepted that data warehouse databases are updated periodically – typically in a daily, weekly or even monthly basis [28]. In our experience, the daily updates seem to be the most used approach. These updates consist on integrating new data into the data warehouse databases and rebuilding all the associated optimization data structures, such as indexes, materialized views, etc. While these update procedures are executed, the databases are offline, i.e., unavailable to end users such as decision makers and OLAP tools. Between these updates, i.e., while the databases are available, the existing data is static and suffers no changes in its contents and structures.

Now suppose that several decision makers need to execute the same queries among each other along the day, for instance, consulting how much was the total sales amount of the day before the current. During that same day, the existing data in the data warehouse databases does not change. This brings up a very relevant question: Why should we request the execution of similar queries more than once, between data warehouse updates, if the data is always the same? The results are also always the same! Therefore, if we store the results for the most recently executed queries, which decision makes will probably need to consult repeatedly, we already have fast direct

access to the results and do not need to process those queries once more. Furthermore, the new data which is integrated in the databases is always incremental, i.e., it adds new records and never changes previously stored data [12]. Therefore, if a repeatable query is executed before a data update, and a user requests its execution afterwards, in order to obtain its results we should only query the most recent added factual data and join the results with the previously stored ones from the query's prior execution.

It is also avoidable time and resource consumption if more than one user is requesting to execute similar queries at the same time. If we can compare real-time simultaneous query execution between the data warehouse users, we can also see if there are any similar queries which are being requested to execute at the same time. Therefore, if we consider a set of users which are trying to execute the same queries, and put the latest users on hold until the first conclude the query processing and return the results for all of them, we efficiently avoid overuse of resource consumption and processing time, minimizing query response time.

The Query Cache Tool deals with all of the mentioned issues, looking to optimize all repeatable and parallel querying. In the following subsections, we shall explain what data structure is used for managing the query execution history and how the query cache algorithm works.

## 2.1 The Query Cache Tool Data Schema

In order to store all the needed information for the query cache tool, we propose the data schema which is presented in Figure 1.

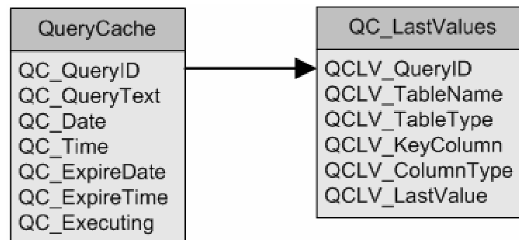



**Fig. 1.** Query Cache Tool data schema

Table `QueryCache` is the master table for the Query Cache Tool (QCT). It will store one row for each query which is executed by the QCT. Column `QC_QueryID` is a unique identifier for each SQL query instruction and column `QC_QueryText` stores a copy of the instruction. Columns `QC_Date` and `QC_Time` store the date and time when the respective query was first executed. Columns `QC_ExpireDate` and `QC_ExpireTime` allow defining when will the respective query's result become overdue or irrelevant. When this happens, the QCT will automatically delete all references and results to it, in what we call the QCT `QueryCacheCleanSweep` procedure, which we will explain further on in this paper. Column `QC_Executing` is a logical flag attribute which indicates if the respective query is currently being executed or not.

Table QC_LastValues is a detail table which will store the last values of the data in each dimension and fact tables which are needed for processing each query. Column QCLV_QueryID references the query identifier QC_QueryID for the query in the master table QueryCache. Column QCLV_TableName indicates the name of a table which is needed for query identified by QCLV_QueryID, and column QCLV_TableType indicates if that table is dimensional (D) or factual (F). Columns QCLV_KeyColumn, and QCLV_ColumnType respectively indicate the name of the key and type of a key column existing in table QCLV_TableName, while column QCLV_LastValue stores the greatest recorded value for that QCLV_KeyColumn in table QCLV_TableName.

For the QCT, each requested query execution generates a table denominated QCacheResponseX, which stores the corresponding result, where X is the value of the query's identifier QC_QueryID in the QueryCache table. For instance, if it receives a query to execute to which it associates QC_QueryID = 1, the corresponding results of its execution is stored in an isolated table QCacheResponse1, in the QCT database.

We shall now explain how our QCT algorithm uses this data schema in order to optimize repeatable and parallel OLAP query execution.

## 2.2 The Query Cache Tool Algorithm

As we mentioned before, the QCT assumes that if no new data has been added to the data warehouse database, the results for any query X which has already been executed is stored in one of the formerly saved QCacheResponseX tables. Therefore, there is no need to execute these queries again, just to supply the results by returning the rows in the correspondent QCacheResponseX table which relates to the desired query, saving time and resource consumption. This makes supplying results for repeated queries an extremely fast task for the QCT.

Suppose a certain user *A*, which starts the execution of an OLAP query *X*. If another user *B*, has previously started executing an OLAP query *Y*, similar to query *X*, and which is currently being processed, our method does not execute query *X*. Instead, it discards the execution of query *X* and puts user *A* on hold while query *Y* finishes being processed, and then returns the same results to both users *A* and *B*. This allows avoiding time and resource consumption for simultaneous similar query execution, speeding up response time for this type of parallel querying.

The algorithm also needs to insure the creation and storage of the results from the first execution of each different query, along with the latest values of each dimensional and factual table needed in processing those results, for identifying in the future if the data warehouse database data has changed or not. It also needs to define the validity of each query results, for automatically disposing those which become overdue.

The QCT algorithm for OLAP query execution is showed below. Due to space constraints, this algorithm is presented in a simple and summarized manner, for its complete code list is too long to include in this paper. However, further ahead we will explain more thoroughly how it works, using an example.

```
PROCEDURE ExecuteQuery(QueryN: SQL Query Instruction)
BEGIN
   IF THERE IS A ROW IN QueryCache WHERE QC_QueryText = QueryN THEN
      QID = QC_QueryID FOR QueryN
      IF QueryN IS ALREADY BEING PROCESSED (QC_Executing = TRUE) THEN
         WAIT
            DELAY Y SECONDS
            VERIFY QC_Executing VALUE FOR QueryN
         UNTIL QC_Executing FOR QueryN IS EQUAL TO FALSE
      ELSE
         SAVE QC_Executing = TRUE IN QueryCache FOR QC_QueryID = QID
         ReQuery = FALSE
         FOR EACH TABLE NEEDED IN QueryN
            LOOKUP LAST RECORDED VALUES IN EACH KEY COLUMN
            IF VALUES ARE DIFFERENT FROM
               RECORDED VALUES IN QC_LastValues FOR QueryN THEN
               LOOKUP LAST RECORDED VALUES IN EACH KEY COLUMN
               SAVE THOSE LAST RECORDED VALUES IN QC_LastValues
               ReQuery = TRUE
            END IF
         NEXT
         IF ReQuery = TRUE THEN
            FOR EACH FactTable IN QueryN
               BUILD TmpFactTable WITH ALL THE NEW ROWS INSERTED
                  SINCE LAST EXECUTION OF QueryN
            NEXT
            EXECUTE QueryN AGAINST TmpFactTables
            JOIN RESULTS WITH PREVIOUSLY STORED QCacheResponseX
               WHERE X = QC_QueryID FOR QueryN
            RECREATE QCacheResponseX WITH NEW RESULTS
            SAVE QC_Executing=FALSE IN QueryCache FOR QC_QueryID=QID
         END IF
      END IF
   ELSE
      DETERMINE A NEW QC_QueryID FOR QueryN
      INSERT A NEW ROW IN QueryCache FOR QueryN
         WITH QC_Executing = TRUE
      FOR EACH TABLE NEEDED IN QueryN
         LOOKUP LAST RECORDED VALUES IN EACH KEY COLUMN
         SAVE THOSE LAST RECORDED VALUES IN QC_LastValues
      NEXT
      EXECUTE QueryN AND SAVE RESULTS IN QCacheResponseX
         WHERE X = QC_QueryID FOR QueryN
      SAVE QC_Executing = FALSE IN QueryCache FOR QC_QueryID = QID
   END IF
   RETURN RESULTS BY SELECTING ALL ROWS FROM QCacheResponseX
      WHERE X = QC_QueryID FOR QueryN
END
```

We have highlighted the instructions which distinguish the major sections of our QCT algorithm. The first highlighted `IF` instruction verifies if the submitted query *QueryN* has already been executed earlier, meaning that it has already been stored in `QueryCache` and its results are stored in a corresponding `QCacheResponse` table. If *QueryN* exists in `QueryCache`, the second `IF` instruction checks if it is currently being executed on behalf of other user, and if this is true, waits until the execution finishes. Otherwise, it processes the query against the data which has been added to the database since it was last executed and joins those results to the previously stored ones, saving new results in the corresponding `QCacheResponse` table. The actual last recorded values of each key column for each table in the query are recorded in `QC_LastValues` with `QC_QueryID` of the each current query, for future comparison in data content updates. If the data has not changed since the query's last execution, no processing is needed, since the results are already stored in the corresponding

QCacheResponse table. If the first highlighted IF instruction is FALSE, this means this is the first time execution of *QueryN*. Consequently, a new QC_QueryID value is given to the query, which is recorded in a new row in QueryCache for identification, along with the query's features (complete SQL instruction, current execution date and time, expiring date and time, and QC_Executing flag attribute as TRUE). The actual last recorded values of each key column for each table in the query are recorded in QC_LastValues with QC_QueryID of the current query for future comparison in data content updates. The results of the query's execution are then stored in the corresponding QCacheResponse table. The results to all users which submitted the query are given by querying the QCacheResponse table, independently if it is a first time execution, a waiting process or an incremental join to previously stored results.

Joining the results from a previous execution of a query with new processed results requires taking several issues under consideration. Queries containing the SUM and COUNT aggregation functions do not need to be changed. The first stored results just need to be added to the new ones. The final results of the queries with aggregation functions is computed in a similar way as in data warehouse stripping, presented in [4, 5]. The average function AVG is calculated dividing a SUM by COUNT, and if there is a need for obtaining STDDEV and VARIANCE, they are determined by usage of COUNT, VARIANCE, SUM and COUNT functions, as shown in the previous mentioned papers.

As time goes by, the number of QCacheResponse tables and consequent storage space they take up need to be dealt with. This is done by looking for the results of queries which have been considered overdue or obsolete, checking the values of the QC_ExpireDate and QC_ExpireTime columns. To perform this, the QCT executes a procedure which we have called the QueryCacheCleanSweep. This procedure seeks for all the rows in the QueryCache referring to queries which are not currently being executed (QC_Executing = FALSE) and where the current server date/time considers overtime (already past the values of the QC_ExpireDate and QC_ExpireTime columns). The procedure is automatically executed every *X* seconds, where *X* should be defined by the Database Administrator after consulting with decision makers as to which is the minimum period of interestingness for any query. The simplified algorithm for the QueryCacheCleanSweep is shown below.

```
PROCEDURE QueryCacheCleanSweep
BEGIN
   FOR EACH Row IN QueryCache WHERE QC_Executing = FALSE
      QID = VALUE OF KEY COLUMN QC_QueryID IN CURRENT QueryCache ROW
      IF (CurrentDate() > QC_ExpireDate) OR
         (CurrentDate() = QC_ExpireDate AND
          CurrentTime() >= QC_ExpireTime) THEN
         DELETE ALL ROWS IN QC_LastValues WHERE QCLV_QueryID = QID
         DELETE CURRENT ROW IN QueryCache
         DROP TABLE QCacheResponseX WHERE X = QID
      END IF
   NEXT
END
```

## 2.3 Illustrating how the Query Cache Tool works

We shall now illustrate an example for explaining how the QCT works. Consider a data warehouse with a schema similar to Figure 2. This schema is based on a real-world data warehouse database, concerning a commercial sales business enterprise.

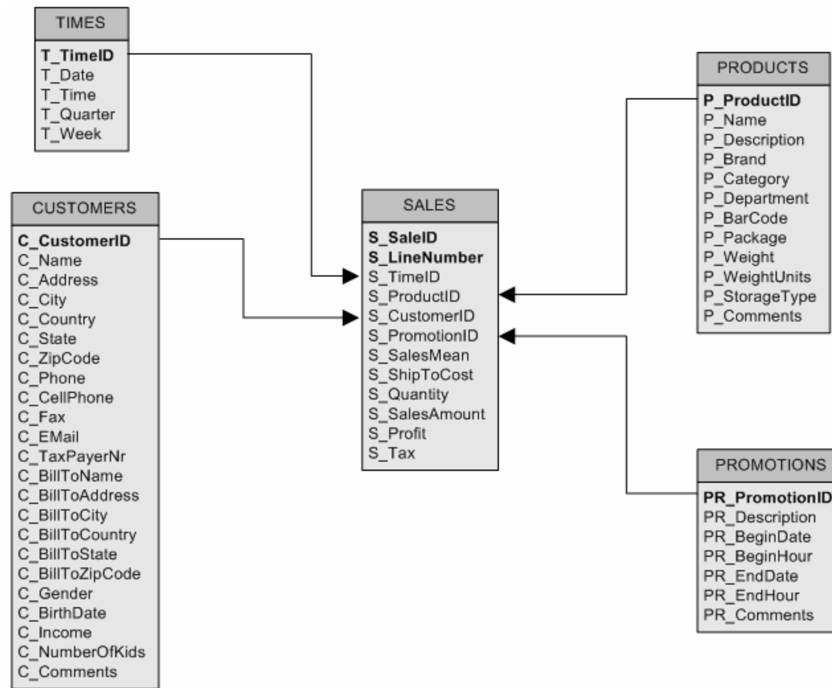



**Fig. 2.** Commercial Sales Business Enterprise Data Warehouse Star Schema

The shown schema represents a typical star schema [12], with one central fact table (Sales) and four dimensional tables, which are connected to the fact schema through their respective key columns. Now, suppose we want to use PCT for executing a query which tells us the total sales amount, total sales profit, and total shipping cost for the sales concerning month of December 2008. This can be done by submitting the following query for execution:

```
SELECT SUM(S_SalesAmount) AS TotalSalesAmount,
       SUM(S_Profit) AS TotalSalesProfit,
       SUM(S_ShipToCost) AS TotalShipToCost
FROM Sales, Times
WHERE S_TimeID = T_TimeID AND
      T_Date >= TO_DATE('01-12-2008','DD-MM-YYYY') AND
      T_Date <= TO_DATE('31-12-2008','DD-MM-YYYY');
```

Consider that this query was first executed at 30-12-2008, and that, up to this day, the latest recorded values for fact table `Sales` key columns `S_SaleID` and `S_LineNumber` is equal to 22 250 000 and 1, respectively. Supposing this query is the first query to be executed by QCT, an example of the consequent insertion of this information in the QCT database is similar to what can be seen in Tables 1 and 2.

**Table 1.** Content of the QueryCache table for an example

| QueryCache table contents | | | | | | |
|---|---|---|---|---|---|---|
| QC_QueryID | QC_QueryText | QC_Date | QC_Time | QC_ExpireDate | QC_ExpireTime | QC_Executing |
| 1 | 'SELECT …' | 31-12-2008 | 09:00 | 01-01-2009 | 09:00 | TRUE |

**Table 2.** Content of the QC_LastValues table for an example

| QC_LastValues table contents | | | | | |
|---|---|---|---|---|---|
| QCLV_QueryID | QCLV_TableName | QCLV_TableType | QCLV_KeyColumn | QCLV_ColumnType | QCLV_LastValue |
| 1 | 'Sales' | 'F' | 'S_SaleID' | 'N' | 22 250 000 |
| 1 | 'Sales' | 'F' | 'S_LineNumber' | 'N' | 1 |

After it has been processed, the results of this query are stored in `QCacheResponse1`, which is an isolated table in the QCT database. Now suppose new data has been added to the Sales fact table. This will imply the existence of a greater value for at least one of their key columns `S_SaleID` and `S_LineNumber`. The next time the query is executed by QCT, it will see that the original `QCLV_LastValue` for at least one of the key columns has changed, building a temporary auxiliary fact table for executing the query and joining these results with the previously stored ones. For this particular query, the set of instructions which may accomplish this can be similar to the following.

```
CREATE TABLE QCTempSales1 AS
   SELECT * FROM Sales
   WHERE (S_SaleID >   (SELECT QCLV_LastValue FROM QC_LastValues
                         WHERE QCLV_QueryID = 1 AND
                               QCLV_KeyColumn = 'S_SaleID') OR
         (S_SaleID=    (SELECT QCLV_LastValue FROM QC_LastValues
                         WHERE QCLV_QueryID = 1 AND
                               QCLV_KeyColumn = 'S_SaleID') AND
          S_LineNumber>(SELECT QCLV_LastValue FROM QC_LastValues
                         WHERE QCLV_QueryID = 1 AND
                               QCLV_KeyColumn = 'S_LineNumber));
CREATE TABLE QCTempQCacheResponse1 AS
   SELECT SUM(S_SalesAmount) AS TotalSalesAmount,
          SUM(S_Profit) AS TotalSalesProfit,
          SUM(S_ShipToCost) AS TotalShipToCost
   FROM QCTempSales1, Times
   WHERE S_TimeID = T_TimeID AND
         T_Date >= TO_DATE('01-12-2008','DD-MM-YYYY') AND
         T_Date <= TO_DATE('31-12-2008','DD-MM-YYYY');
CREATE TABLE QCTempQCacheFinalResponse1 AS
   SELECT SUM(TotalSalesAmount),
          SUM(TotalSalesProfit),
          SUM(TotalShipToCost)
   FROM (SELECT * FROM QCacheResponse1) UNION ALL
        (SELECT * FROM QCTempQCacheResponse1);
DROP TABLE QCacheResponse1;
DROP TABLE QCTempQCacheResponse1;
CREATE TABLE QCacheResponse1 AS
   SELECT * FROM QCTempQCacheFinalResponse1;
DROP TABLE QCTempQCacheFinalResponse1;
```

The results for the users which have requested the execution of this query are then returned as a `SELECT * FROM QCacheResponse1`. The next section presents an experimental evaluation of the QCT using a real-world data warehouse database.

## 3 Experimental Evaluation

To test the QCT, we have implemented a data warehouse similar to the one presented in Figure 2. As we mentioned before, this data warehouse is based on a real-life database, concerning a commercial sales business enterprise. The dimension features of the database, corresponding to one year of commercial data (2008), are shown in Table 3. To build the data warehouse, we used Oracle 10g DBMS on a 2.8 GHz Pentium IV CPU, with 1 GByte RAM and a 180 GByte 7200 rpm hard disk.

**Table 3.** Dimensional features of the Commercial Sales Business Enterprise Data Warehouse

| | Times | Customers | Products | Promotions | Sales |
|---|---|---|---|---|---|
| **Number of Rows** | 8 760 | 250 000 | 50 000 | 89 812 | 31 536 000 |
| **Storage Size** | 0,12 MB | 90 MB | 7 MB | 10 MB | 1 927 MB |

In order to obtain results for aiding decision making in this business, we have considered the following set of decision support OLAP queries:

1) Query Q1: For determining the sales quota for each product department, relating to the whole year's total sales amount;
2) Query Q2: For showing the total sales amount, total sales profit and total ship cost for a certain month of the year;
3) Query Q3: For listing the best 1000 clients, in total sales amount, since the beginning of the year;
4) Query Q4: For determining the total sales value for each product relating to each defined promotion in a certain day;
5) Query Q5: For showing the total sales amount for each area zip code, in a certain month of the year;
6) Query Q6: For listing the 10 most sold products, in sales amount, for each gender (male/female) and belonging to each class of income value (Minimum Income: below 600; Reasonable Income, at least 600 and below 1000; Medium Income: at least 1000 and below 1500; High Income: at least 1500 and below 2500; and Very High Income: at least 2500);
7) Query Q7: For showing the sales amount and sales quota for each country, concerning the total sales occurred in a certain month of the year;
8) Query Q8: For listing all the products which have not been sold at all in a certain week of the year;
9) Query Q9: For listing all the customers which have not purchased anything in a certain month of the year;
10) Query Q10: For listing how many customers are there for each area zip code;
11) Query Q11: For listing how many customers are there for each income value class (for the income value classes presented in query Q6);
12) Query Q12: For showing how many customers are there for each age class (Under Age: below 18 years old; Young Adult: at least 18 and below 30; Middle Age Adult: at least 30 and below 45; Mature Adult: at least 45 and below 65; Senior Adult: at least 65 years old).

The set of these twelve queries represents the workload which we used in each experimental scenario. We have tested the QCT for every day of December, 2008, considering four execution possibilities for each query:

a) *Standard Execution:* traditional execution of the query workload in the Oracle SQL*Plus interface in a standard manner;
b) *QCT First Execution:* execution of the query workload by the QCT, assuming that the query cache database is empty, i.e., each query is being executed for the first time by QCT;
c) *QCT Incremental Execution:* execution of the workload by the QCT, where a first execution has been previously made and their results are already stored in the query cache database, and after inserting an entire day of new data in the data warehouse fact table (which stands for an average of 86746 new rows in `Sales`), to join these new results with the previously stored ones;
d) *QCT Sequential Execution:* execution of the query workload by the QCT a second time after they have already been executed and their results are already stored in the query cache database, with no change in the data warehouse tables' contents.

Assuming that the data warehouse is updated in a daily fashion, first we shall present the results concerning the usage of the QCT during one day, against traditional query workload execution. Tables 4, 5, and 6 present the results for comparing standard query workload execution on 31-12-2008, against each of the three presented execution possibilities using the QCT on the same day.

**Table 4.** Standard workload execution time vs. QCT workload first exec. time on a day

| | **Standard Exec. Time** | **QTC First Exec. Time** | **Time Difference** | **Times Faster/Slower** |
|---|---|---|---|---|
| **1 User** | 764 s | 810 s | + 46 s | 1.06 times slower |
| **2 Users** | 1336 s | 832 s | **- 504 s** | **1.61 times faster** |
| **4 Users** | 2050 s | 1212 s | **- 838 s** | **1.69 times faster** |
| **8 Users** | 4206 s | 1868 s | **- 2338 s** | **2.25 times faster** |
| **16 Users** | 7807 s | 3797 s | **- 4010 s** | **2.06 times faster** |

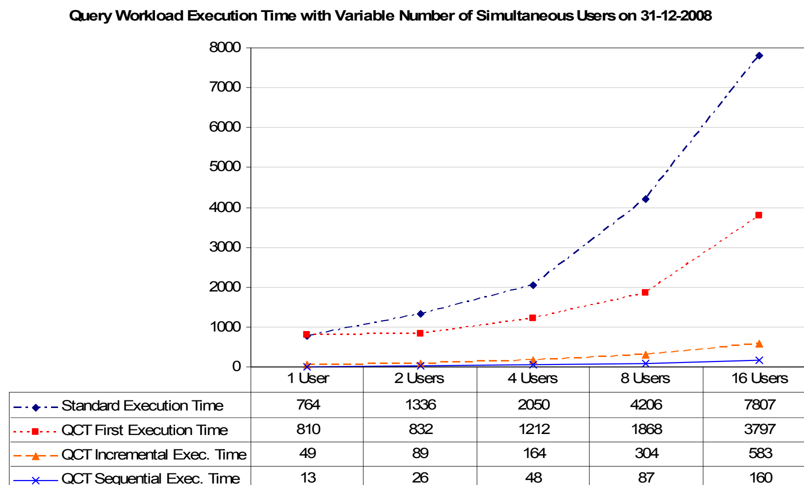
**Table 5.** Standard workload exec. time vs. QCT workload incremental exec. time on a day

| | **Standard Exec. Time** | **QTC Incremental Exec. Time** | **Time Difference** | **Times Faster/Slower** |
|---|---|---|---|---|
| **1 User** | 764 s | 49 s | **- 715 s** | **15.6 times faster** |
| **2 Users** | 1336 s | 89 s | **- 1247 s** | **15.0 times faster** |
| **4 Users** | 2050 s | 164 s | **- 1886 s** | **12.5 times faster** |
| **8 Users** | 4206 s | 304 s | **- 3902 s** | **13.8 times faster** |
| **16 Users** | 7807 s | 583 s | **- 7224 s** | **13.4 times faster** |

**Table 6.** Standard workload exec. time vs. QCT workload sequential exec. time on a day

| | **Standard Exec. Time** | **QTC Sequential Exec. Time** | **Time Difference** | **Times Faster/Slower** |
|---|---|---|---|---|
| **1 User** | 764 s | 13 s | **- 751 s** | **58.8 times faster** |
| **2 Users** | 1336 s | 26 s | **- 1310 s** | **51.4 times faster** |
| **4 Users** | 2050 s | 48 s | **- 2002 s** | **42.7 times faster** |
| **8 Users** | 4206 s | 87 s | **- 4119 s** | **48.3 times faster** |
| **16 Users** | 7807 s | 160 s | **- 7647 s** | **48.8 times faster** |

As it can be seen from the results, the QCT is much faster than the standard query execution for all cases, except for the first execution with only 1 user querying, showed in *QCT First Time Execution*. This happens because the QCT also has to execute the first workload with 1 user in a standard manner and still has to create and store the initial results. However, for more than 1 user, the QCT takes advantage of checking if there are any similar queries executing simultaneously, dismissing parallel querying for those queries, contrarily to the standard execution, which reexecutes all of the queries. This means that the more the users, the better QCT outperforms the standard execution. This can be confirmed by observing Figure 3.




| | 1 User | 2 Users | 4 Users | 8 Users | 16 Users |
|---|---|---|---|---|---|
| Standard Execution Time | 764 | 1336 | 2050 | 4206 | 7807 |
| QCT First Execution Time | 810 | 832 | 1212 | 1868 | 3797 |
| QCT Incremental Exec. Time | 49 | 89 | 164 | 304 | 583 |
| QCT Sequential Exec. Time | 13 | 26 | 48 | 87 | 160 |

**Fig. 3.** Query workload execution of standard execution vs. QCT execution for 31-12-2008

By analyzing the previous tables and figure, we can see that if a query which has been stored in the query cache database is repeated, showed by the *QCT Sequential Execution*, the QCT can supply the results around 50 times faster, for it only needs to access the previously stored results in order to process the query. It is also much faster to join new calculated results from new added data with previously stored ones to supply query results, showed by the *QCT Incremental Execution*, than reexecuting the queries against the whole amount of data.

Figures 4, 5, 6, 7 and 8 show the workload execution time for each day of December, 2008, for 1, 2, 4, 8 and 16 simultaneous users querying, for each possible type of query execution. All results confirm the previous analysis, which demonstrate that the QCT is always much more efficient than standard query execution.

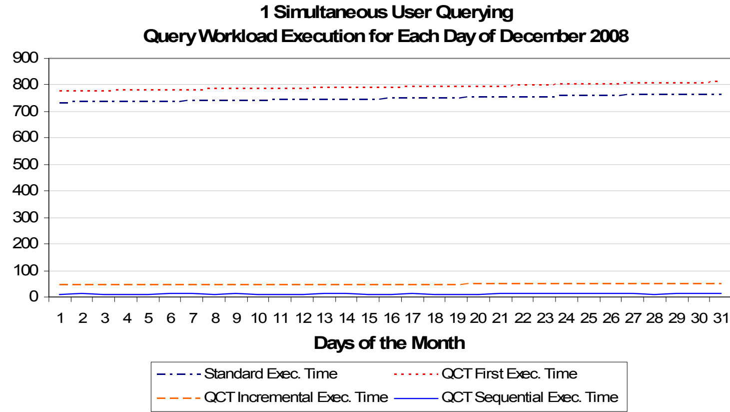



**Fig. 4.** Query workload execution with 1 user for December, 2008

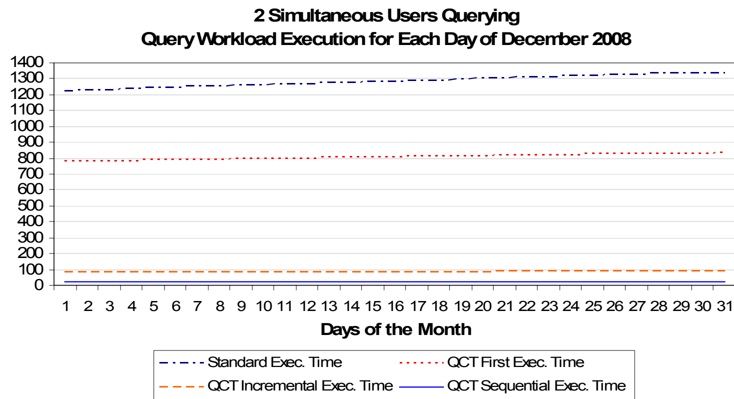



**Fig. 5.** Query workload execution with 2 simultaneous users for December, 2008

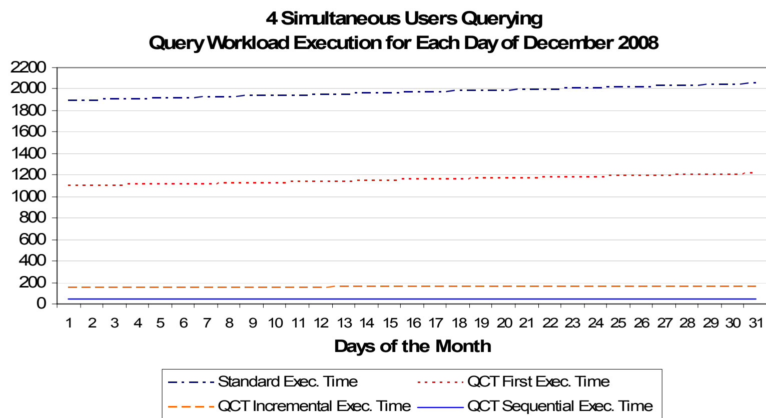



**Fig. 6.** Query workload execution with 4 simultaneous users for December, 2008

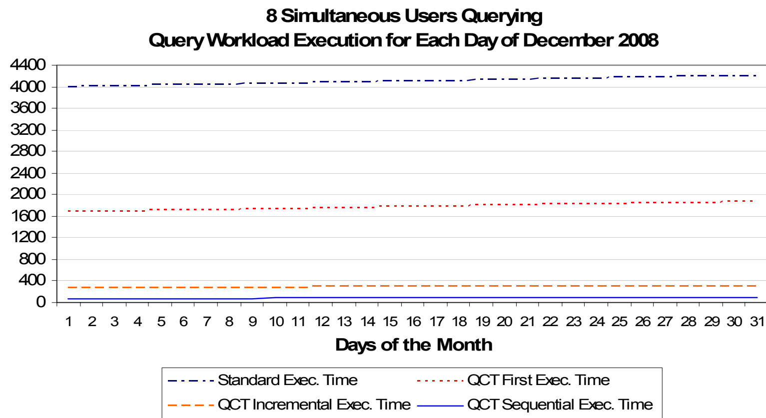



**Fig. 7.** Query workload execution with 8 simultaneous users for December, 2008

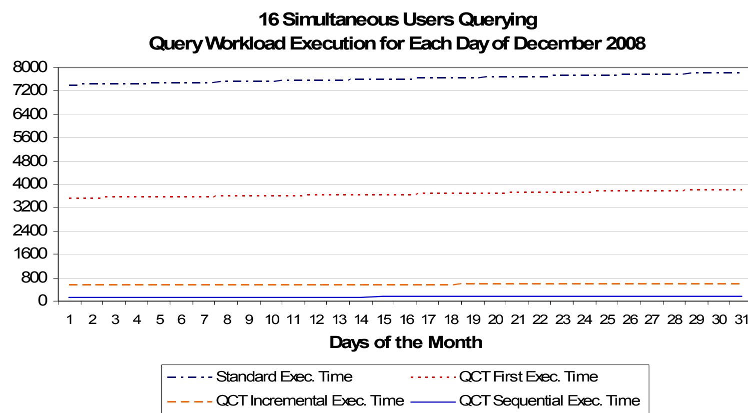



**Fig. 8.** Query workload execution with 16 simultaneous users for December, 2008

## 4 Related Work

Most of the research work done in this area is focused on optimizing data search methods and physical data distribution. Our method aims for the OLAP SQL instruction level. There is vast literature on query processing and load balancing in parallel database systems [e.g. 1, 16] and distributed databases [e.g. 20]. In [17], the potential of parallel processing in data warehouse loading processes and maintenance of materialized views is discussed. However, this paper does not cover the use of parallel technology for data warehouse analysis.

Many DBMS vendors claim to support parallel data warehousing to various degrees, e.g. Oracle10g R2 [19], IBM/Informix Red Brick [22], and the Microsoft SQL Server [11]. Most of these products, however, do not use dimensionality of data that exists in a data warehouse and it remains unclear to what extent multidimensional

fragmentation is exploited to reduce query work. None of the aforementioned vendors provide sufficient information or even tool support on how to determine an adequate data allocation for star schemas. The effective use of parallel processing in this environment can be achieved only if we are able to find innovative techniques for parallel data placement using the underlying properties of data in the warehouse.

One of the first works to propose a parallel physical design for the data warehouse was [8]. A data indexing strategy that suggests a vertical partitioning of the star schema to provide efficient data partitioning and parallel resource utilization is proposed. In this work the authors propose algorithms that split the data among N parallel processors and perform parallel join operations, but without quantifying potential gains. Recently was proposed a multidimensional hierarchical fragmentation and allocation method for star schemas in a parallel data warehouse environment [25]. This approach called MDHF (MultiDimensional Hierarchical Fragmentation) allows all star queries referencing at least one attribute from any fragmentation dimension to be confined to a subset of the fact table fragments. This approach assumes a shared disk parallel database system that exhibits near linear scalability with respect to the number of disks and processors, but in certain cases (partially filled bitmap indexes) it shows an increase in I/O load and has some administration overhead.

In order to properly handle large volumes of data, allowing to perform complex data manipulation operations, enterprises normally use high performance systems to host their data warehouses. The most common choice consists of systems that offer massive parallel processing capabilities [2, 26], as Massive Parallel Processing (MPP) systems or Symmetric MultiProcessing (SMP) systems. Due to the high price of this type of systems, some less expensive alternatives have already been proposed and implemented [6, 9, 18]. One of those alternatives is the Data Warehouse Stripping (DWS) technique [4, 5].

A large amount of research has been performed for processing and optimizing queries over distributed data (see, e.g. [3, 13, 14, 15, 23, 24, 27]). However, this research has focused mainly on distributed join processing rather than distributed computation. Only recently [7] we have a new architecture and optimizations for parallel SQL execution in the Oracle 10g database and a practical solution for parallelizing query optimization in the multi-core processor architecture, including a parallel join enumeration algorithm and several alternative ways to allocate work to threads to balance their load. This solution has been prototyped in PostgreSQL [11]. The approach we explore in this paper marries the concepts of distributed processing and parallel OLAP queries to provide a fast and reliable relational data warehouse.

## 5 Conclusions and Future Work

We have explained how our query cache tool works, optimizing the execution of repeatable OLAP queries and simultaneous similar query executions. We have also shown that our query cache tool is efficient, significantly reducing query execution time and processing resources. The presented results in the experimental evaluation show that the query cache method is much better than the standard query workload execution, for this type of queries.

As future work, we intend to enhance the method for including features which can deal with queries which represent incremental column results that can be added to the results of other previously processed queries. We also mean to work on similar query recognition, for identifying similar OLAP query instructions which are not written exactly the same way, but aim for similar results.

## References


[1] Abdelguerfi, M., Wong, K.: Parallel Database Techniques. IEEE Computer Society Press, 1988.

[2] Agosta, L.: Data Warehousing Lessons Learned: SMP or MPP for Data Warehousing", DM Review Magazine, 2002.

[3] Akinde, M. O., Bhlen, M. H., Johnson, T., Lakshmanan, L. V. S., Srivastava, D.: Efficient OLAP query processing in distributed data warehouses, Information Systems 28, pp. 111-135, 2003.

[4] Bernardino, J., Madeira, H.: Experimental Evaluation of a New Distributed Partitioning Technique for Data Warehouses, International Symposium on Database Engineering and Applications (IDEAS'01), Grenoble, France, 2001.

[5] Bernardino, J., Furtado, P., Madeira H.: Approximate Query Answering Using Data Warehouse Striping, Journal of Intelligent Information Systems – Integrating Artificial Intelligence and Database Technologies, Vol. 19, Issue 2, Elsevier Science Publication, (2002) 145-167.

[6] Critical Software SA, DWS, www.criticalsoftware.com.

[7] Cruanes, T., Dageville, B., Ghosh, B.: Parallel SQL Execution in Oracle 10g, ACM SIG International Conference on Management of Data (SIGMOD), 2004.

[8] Datta, A., Moon, B., Thomas, H.: A Case for Parallelism in Data Warehousing and OLAP, Proc. of the 9th Int. Conf. Database and Expert Systems Applications (DEXA'98), 1998.

[9] DATAllegro, DATAllegro v3™, www.datallegro.com.

[10] Galindo-Legaria, C. A., Grabs, T., Gukal, S., Herbert, S., Surna, A., Wang, S., Yu, W., Zabback, P., Zhang, S.: Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server, Int. Conf. on Data Engineering (ICDE'08), pp.1190-1199, 2008.

[11] Han, W. S., Kwak, W., Lee, J., Lohman, G. M., Markl, V.: Parallelizing Query Optimization, International Conference on Very Large Data Bases (VLDB), 2008.

[12] Kimball, R., Ross, M.: The Data Warehouse Toolkit, 2nd Ed., John Wiley & Sons, 2002.

[13] Kossman, D.: The state of the art in distributed query processing, ACM Computing Surveys 32 (4) (2000) pp.422–469.

[14] Kossman, D., Franklin, M., Drasch, G.: Cache Investment: Integrating Query Optimization and Dynamic Data Placement. ACM Transact. Database Systems, Dec. 2000, pp.517-558.

[15] Liu, B., Chen, S., Rundensteiner, E. A.: A Transactional Approach to Parallel Data Warehouse Maintenance. In Data Warehousing and Knowledge Discovery, Proceedings. Lecture Notes in Computer Science (LNCS) Springer Verlag, September 2002

[16] Lu, H., Ooi, B. C., Tan, K. L.: Query Processing in Parallel Relational Database Systems. IEEE Computer Society, May 1994.

[17] Garcia-Molina, H., Labio, W., Wiener, J., Zhuge, Y.: Distributed and Parallel Computing Issues in Data Warehousing, 18th ACM Symposium on Principles of Distributed Computing (PODC), Atlanta, USA, May 4-6, 1999.

[18] Netezza, The Netezza Performance Server® DW Appliance, www.netezza.com.

[19] Oracle Data Warehousing Guide 10g R2, available: http://downloadwest.oracle.com/docs/cd/B 19306 01 /server.102/b14223.pdf.

[20] Ozsu, M., Valduriez, P.: Principles of Distributed Database Systems. 2nd Edition, Prentice-Hall, New Jersey, 1999.

[21] Pedersen, T. B.: How is BI Used in Industry?, International Conference on Data Warehousing and Knowledge Discovery (DAWAK), 2004.

[22] RedBrick White Paper, ftp://ftp.software.ibm.com/-software/data/informix/pubs/whitepapers/redbrick wpO40904.pdf.

[23] Schewe, K. D., Zhao, J.: Balancing redundancy and query costs in distributed data warehouses -- an approach based on abstract state machines, 2nd Asia-Pacific Conference on Conceptual Modelling (ER), S. Hartmann and M. Stumptner, Eds., vol. 43 of CRPIT. Australian Computer Society, 2005, pp. 97—105.

[24] Stanoi, I., Agrawal, D., Abbadi, A.: Modeling and Maintaining Multi-View Data Warehouses. In Proc. of the 18th Int. Conf. on Conceptual Modeling (ER), November 1999.

[25] Sthor, T., Martens, H., Rahm, E.: Multi-Dimensional Database Allocation for Parallel Data Warehouses, Proc. 26th Intern. Conference on Very Large Databases (VLDB), 2000.

[26] Sun Microsystems, Data Warehousing Performance with SMP and MPP Architectures, White Paper, 1998.

[27] Vingralek, R., Breitbart, Y., Weikum, G.: Distributed File Organization with Scalable Cost/Performance, ACM SIGMOD Int. Conf. on Management of Data, 1994, pp.253-264.

[28] Zurek, T., Kreplin, K.: SAP Business Information Warehouse – From Data Warehousing to an E-Business Platform, International Conference on Data Engineering (ICDE), 2001.