On Using an Online, Automatic and Non-Intrusive Approach for Rewriting SQL Queries

Arlino H. M. de Araújo[1,2], José Maria Monteiro[2], José Antônio F. de Macêdo[2], Júlio A. Tavares[3], Angelo Brayner[3], Sérgio Lifschitz[4]

[1] Federal University of Piauí, Brazil
[2] Federal University of Ceará, Brazil
{arlino,monteiro,jose.macedo}@lia.ufc.br
[3] University of Fortaleza, Brazil
{julio,brayner}@unifor.br
[4] Pontifical Catholic University of Rio de Janeiro
sergio@inf.puc-rio.br

Abstract. Database applications have become very complex, dealing with a huge volume of data and database objects. Concurrently, low query response time and high transaction throughput have emerged as mandatory requirements. Among other possible interventions regarding database performance, SQL query rewriting has been shown efficient. The idea is to rewrite a new SQL statement equivalent to the statement initially formulated, where the new SQL statement provides performance gains w.r.t. query response time. In this article, we propose an online, automatic and non-intrusive approach for rewriting SQL queries, denoted ARe-SQL. ARe-SQL rewrites SQL statements, using a set of 11 heuristics. Based on ARe-SQL, two different strategies for SQL-query rewriting, denoted assisted and automatic, were implemented. These strategies were evaluated in three different DBMSs considering three distinct scenarios. The results indicate that both strategies can provide performance gains, which, in some cases, reach 92.5%.

Categories and Subject Descriptors: H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Retrieval models*

Keywords: Query processing, Database tuning, Query rewriting

1. INTRODUCTION

Database queries are expressed by means of high-level declarative languages, such as SQL (Structured Query Language), for instance. Such queries are submitted to the query engine, which is responsible for processing queries in database management systems (DBMSs). Thus, query engines should implement four main activities: query parsing, logical execution plan (LEP) generation, physical execution plan (PEP) generation and PEP execution. LEP and PEP generation and PEP execution are often called the query optimization.

The rewriting technique consists in writing a new SQL statement Q_b equivalent to the statement Q_a initially formulated. Two queries are equivalent if and only if their execution produces the same result [Ramakrishnan and Gehrke 2002]. Thus, the executions of Q_a and Q_b return the same result. However, Q_b execution provides performance gains w.r.t. Q_a execution. In order to illustrate performance gains achieved by means of query rewriting, consider the queries Q_a and Q_b depicted in Figures 1 and 2, respectively. In order to execute those queries, the TPC-H benchmark's database has been created in a PostgreSQL server.

Looking more closely to Figures 1 and 2, one can observe that Q_a and Q_b are equivalent. Nonetheless, Q_a execution lasts $13.685ms$, while Q_b is executed in $1.825ms$. In fact, Q_b was obtained by removing the ALL operator from Q_a and including the MIN operator in subquery of Q_a.

Therefore, we may formulate the following hypothesis, the way a query Q is formulated in terms of SQL commands induces the query optimizer to produce a given PEP for Q. To show the veracity of our hypothesis, consider the execution plans for queries Q_a and Q_b depicted in Figures 3 and 4. Those plans have been yielded by PostgreSql's query engine. The PEP showed in Figure 4 is more efficient (1,825 ms) than PEP depicted in Figure 3 (13,685 ms). This fact stems from the least amount of data materialized in the "Materialize" operation, which is performed after the "Aggregate" operation (in the PEP illustrated in Figure 4).

```
SELECT *
FROM orders
WHERE o_totalprice < ALL (SELECT o_totalprice
                          FROM orders
                          WHERE o_orderpriority = '2-HIGH')

Execution time = 13,685 ms
```

Fig. 1. SQL expression for Q_a.

```
SELECT *
FROM orders
WHERE o_totalprice < (SELECT MIN(o_totalprice)
                      FROM orders
                      WHERE o_orderpriority = '2-HIGH')

Execution time = 1,825ms
```

Fig. 2. SQL expression for Q_b.

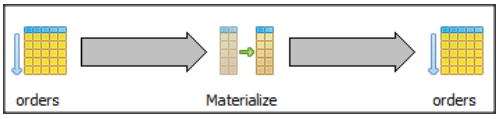

Fig. 3. PEP P_{Q_a} yielded by PostgreSQL for Q_a

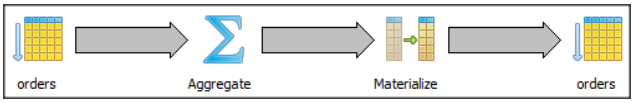

Fig. 4. PEP P_{Q_b} yielded by PostgreSQL for Q_b

Database query tuning tools explores these aspects in order to help query optimizers to produce better PEPs. These tools may be categorized as *(i)* online or offline; *(ii)* automatic or non-automatic; and *(iii)* intrusive or non-intrusive. Offline approaches rely on human intervention for specifying the time window for capturing the workload. In addition, they do not allow the system to react to workload dynamics [Weikum et al. 2002; Chaudhuri and Weikum 2006]. On the other hand, an online tuning tool is able to dynamically capture database workload. Automatic query tuning tools have the ability of automatically triggering adjustments to improve query performance. Conversely, non-automatic tools transfer to the DBA (Database Administrator) the responsibility for making such adjustments. This include the definition of the most appropriate moment to apply the recommended query changes and to choose, among the suggested actions, those that in fact should be executed and in which order. Intrusive solutions are those that require changes in the DBMS code and, therefore, are tightly coupled to a particular DBMS. Non-intrusive solutions are those that are disconnected from a particular DBMS implementation and that are usually not affected by any new DBMS release.

In fact, there are some tools that aim at tuning database query performance by providing rewriting recommendations. However, they adopt an offline approach, since they provide recommendations only when an human (e.g., DBA) requires such recommendations. Most of them are intrusive, which means that they are DBMS-specific. Additionally, they present a strong issue, for a given query they may provide several rewriting recommendations. In this case, the DBA is in charge to choose one of them without any indication (from the tool) that the chosen recommendation will provide the best performance gains regarding response time.

In this article, we propose an online, automatic and non-intrusive approach for rewriting SQL queries, denoted ARe-SQL. The proposed approach provides the necessary support to implement two different strategies for SQL-query rewriting. In the first strategy, denoted automatic, ARe-SQL autonomously captures SQL queries submitted by users or applications and identify which queries should be rewritten. Thereafter, queries are sent to the database query engine. The second strategy, denoted assisted, works as an autonomous advisor, which analyze previously executed SQL statements in order to recommend (through alerts, reports and wizards) SQL tuning opportunities. In both strategies, the rewritten queries profit from performance gains as we show in Section 4.

The key goal of ARe-SQL is to assist optimizers in build efficient query physical execution plan. In a proactive manner, ARe-SQL rewrites SQL statements which will induce query engine to produce better plans than those that would be built by executing the original SQL statements. We advocate that the main features presented by ARe-SQL are:

—Implementation of a non-intrusive strategy. Due to this feature ARe-SQL may be applied to different database systems. In other words, it is not DBMS-specific.

—Autonomy. ARe-SQL has the ability of automatically triggering query rewritings or recommendations. This include the definition of the most appropriate moment to apply the recommended rewritings and to choose, among the suggested actions, those that in fact should be executed and in which order.

—Low-overhead tuning activity. By using our approach, the database system keeps running while our automatic tool runs continuously capturing the workload, in a lightweight manner.

We have implemented the proposed approach, including automatic and assisted strategies. These strategies were evaluated in three different DBMSs (Oracle, SQL Server and PostgreSQL) considering three different scenarios, comprising the TPC-H benchmark. The experimental results indicate that both strategies can provide performance gains. In some cases, such performance gains reach 92.5%.

The rest of the article is structured as follows. Section 2 discuss most relevant related work. The proposed approach is presented and analyzed in Section 3. In turn, Section 4 brings and analyzes experimental results. Finally, Section 5 concludes this article.

2. RELATED WORK

Bruno *et al* propose in [Bruno et al. 2009] a framework, called Power Hints, which enables the creation and use of hints. In the proposed framework, hints are created by means of regular expressions, making it easy and flexible to create restrictions for query execution plans, allowing more precise tuning.

The work proposed in [Herodotou and Babu 2009] presents a tool called zTuned whose purpose is to facilitate experiments related to tune SQL queries. The tool produces sets of plans with operators that have the same cardinality (called neighborhood plans) and chooses the optimal plan among the best plans for each neighborhood, using the mechanism of cost estimative of the DBMS. The tool works decoupled from the query optimizer and can potentially be used with any DBMS that has a cost-based optimizer.

In [Krishnaprasad et al. 2004], the author describes a technique to transform an XML query in an equivalent query in the relational model or object-relational through rewritten at compile time.

In Oracle 10g is introduced a component called Automatic SQL Tuning Advisor that provides recommendations to creating indexes based on predicates present in SQL statements and recommendations of opportunities to rewrite queries using semantic equivalences [Dageville and Dias 2006].

Karl *et al* propose in [Dias et al. 2005] a way to perform automatic performance diagnosis and tuning on Oracle 10g. The proposed solution uses and Automatic Database Diagnostic Monitor (ADDM) that automatically diagnoses the bottlenecks affecting the total database throughput and provides actionable recommendations to alleviate them.

In [Belknap et al. 2009], the authors describe a new Oracle 11g database feature, called Automatic SQL Tuning. This tool extends the SQL Tuning Advisor (present on Oracle 10g) fully automating the SQL tuning workflow and solving some SQL performance problems without any DBA intervention.

Lohman *et al* discuss in [Markl et al. 2003] an autonomic query optimizer, called LEO - LEarning Optimizer for DB2, that automatically self-validates its model without requiring any user interaction to repair incorrect statistics or cardinality estimates. By monitoring queries as they execute, the autonomic optimizer compares the optimizer's estimates with actual cardinalities at each step in a QEP, and computes adjustments to its estimates that may be used during future optimizations of similar queries.

The tool IBM Optim Development Studio [Studio 2010] collects query performance metrics in DB2. The metrics are query execution frequency, cost and time. Thus, with such metrics, it is possible to identify queries, for which query rewriting recommendation are worthwhile. It is important to mention that this tool collects metrics for a given period of time (defined by the DBA).

Embarcadero DB Optimizer XE [Optimizer 2010] has the functionality of identifying hints which should be encoded in a given SQL statement, i.e., it does not propose query rewritings. Its goal is mainly to eliminate unnecessary outer joins and cartesian products. Additionally, Embarcadero may provide recommendation for

improving index configuration. Quest SQL Optimizer for Oracle [Quest 2010] provides semantically equivalent SQL statements for a given query. In this case, the DBA should pass to the tool which query should be analyzed. Moreover, it is up to the DBA to choose one of the several recommended SQL statements.

Therefore, most of the investigated SQL tuning tools (Automatic SQL Tuning Advisor [Dageville and Dias 2006], IBM Optim Development Studio [Studio 2010], Embarcadero DB Optimizer XE [Optimizer 2010] and Quest SQL Optimizer for Oracle [Quest 2010]) adopt an offline approach. In this sense, they transfer to the DBA the responsibility for defining the set of queries to be evaluated for choosing one of the several alternatives provided by them. Observe that DBAs work in a reactive way, i.e., they only trigger a tool or an advisor when the problem already exists. Furthermore, after identifying the problem (a time consuming query) and a possible solution, a DBA should rewrite the SQL statement, test and send it to a programmer to change the application code which is using the SQL statement. This whole process is rather time consuming. On the other hand, Oracle Automatic SQL Tuning is an online tool. However, it's an intrusive solution and runs only with Oracle 11g. The approach proposed in this article differs from previous work since ARe-SQ is an online, automatic and non-intrusive approach. Furthermore, most of these tools are proprietary, involving high acquisition costs. These aspects hinder the performance comparison between ARe-SQL and previous solutions.

3. ARE-SQL: AN ONLINE, AUTOMATIC AND NON-INTRUSIVE APPROACH FOR REWRITING SQL QUERIES

ARe-SQL's main functionality is to influence query optimizer to choose an effectual query execution plan. ARe-SQL proactively rewrites SQL statements which will induce query engine to produce better plans. In order to achieve its goal, ARe-SQL works directed by a set of heuristics. The heuristics consist of rules to identify potential opportunities for tuning SQL statements and the ways to rewrite the statements. Table I brings the eleven heuristics applied by ARe-SQL. Furthermore, it indicates whether or not each heuristic is currently implemented by three major DBMSs: PostgreSQL 8.3, Oracle 11g and SQL Server 2008.

Table I. Heuristics for SQL Tuning.

	Heuristcs for SQL Tuning	PostgreSQL	Oracle	SQL Server
H1	Transform queries which create and use temporary table into an equivalent sub-query.	No	No	No
H2	Eliminate unnecessary GROUP BY.	No	No	No
H3	Remove having clause whose predicates do not have any aggregate function. The predicates should be moved to a WHERE clause.	No	No	No
H4	Change query with disjunction in the WHERE to a union of query results.	No	Yes	No
H5	Remove ALL operation with greater/less-than comparison operators by including a MAX or MIN aggregate function in the subquery.	No	Yes	No
H6	Remove SOME operation with greater/less-than comparison operators by including a MAX or MIN aggregate function in the subquery.	No	Yes	No
H7	Remove ANY operation with greater/less-than comparison operators by including a MAX or MIN aggregate function in the subquery.	No	Yes	No
H8	Replace IN set operation by a join operation.	No	Yes	No
H9	Eliminate unnecessary DISTINCT.	No	No	No
H10	Move function applied to a column index to another position in the expression.	No	No	No
H11	Move arithmetic expression applied to a column index to another position in the expression.	No	No	No

3.1 ARe-SQL Architecture

In order to implement ARe-SQL, two different approaches were employed: assisted and automatic. For each approach, a different tool was implemented. However, both approaches present the following features: *(i)* Non-intrusive: completely decoupled from the source code of the DBMS. This allows that the conceived solution can be used with any DBMS and *(ii)* Independent of location: it can run on a machine different than that used to host the DBMS, not consuming server resources where the DBMS is hosted.

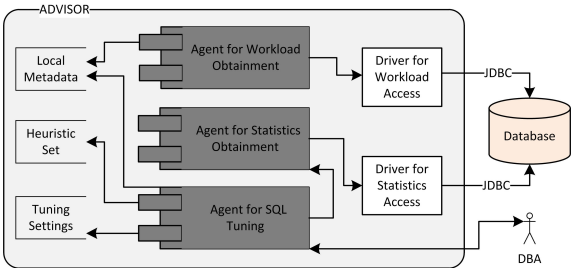

Fig. 5. ARe-SQL Advisor's Architecture.

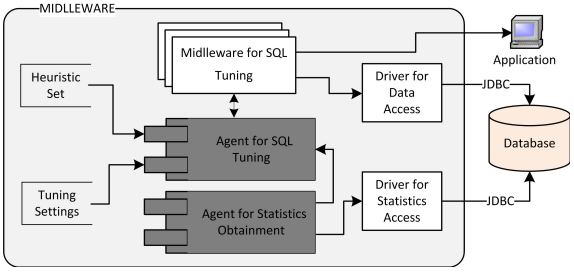

Fig. 6. Architecture of ARe-SQL's Automatic Approach.

3.1.1 *Assisted Approach.* A query-rewriting tool based on assisted approach consists of an advisor which has the ability: *(i)* to capture the previously SQL statements executed by DBMS; *(ii)* to analyze these statements, and; *(iii)* to recommend (through alerts, reports or wizards) SQL tuning opportunities. Thus, the advisor can identify SQL statements that could be rewritten. Additionally, such a type of tool allow the DBA to interact with the tuning process. For instance, a DBA may select a subset of available heuristics to be applied for the SQL-query tuning process. In other words, the DBA can specify that some heuristics are unnecessary or inappropriate for a given database system or a given statement.

We have implemented a tool, called ARe-SQL Advisor, which implements an assisted approach for ARe-SQL. Figure 5 depicts an abstract model of ARe-SQL Advisor architecture. To utilize ARe-SQL Advisor in a given DBMS, it is necessary to instantiate two specific drivers: a driver for workload access (DWA) and a driver for statistics access (DSA). These drivers are the only components ofARe-SQL Advisor which are DBMS-specific. Nevertheless, these drivers are implemented in a non-intrusive fashion. We have so far implemented drivers for the following DBMSs: PostgreSQL 8.4, Oracle 11g and SQL Server 2008 (see Section 4). The main components of the architecture illustrated in Figure 5 are the following:

—Agent for Workload Obtainment (AWO): This agent observes the operations submitted to the DBMS and retrieves the SQL statements and then stores them in the local metabase (Local MetaData - LM). This agent can be configured to run continuously (On-the-fly).

—Local Meta Data (LM): Database that stores workloads captured by AWO.

—Driver for Workload Access (DWA): This component enables ARe-SQL Advisor to access metabase (catalog) of a given DBMS.

—Agent for Statistics Obtainment (ASO): This component is in charge of accessing statistics information of the target DBMS, such as table cardinality, the amount of disk pages required to store a database table, the height of (B^+ tree) index structures and so forth.

—Driver for Statistics Access (DSA): driver that allows the ASO to retrieve statistics for a specific DBMS.

—Heuristic Set (HS): set of heuristics used by the agents to identify SQL statements with oportunities of tuning and in order to rewrite them. HS is composed of 11 heuristics, which are depicted in Table I. However, new heuristics can be defined and inserted into HS.

—Agent for SQL Tuning (AST): Component responsible for tuning a particular SQL statement using the set of heuristics (HS).

—Tuning Settings (TS): This component is preference file containing pairs <SQL statement Q, subset of heuristics H> defined by the DBA. Each pair indicates a heuristic subset chosen by DBA from HS that he/she wants to be applied for tuning Q. If there isn't an entry in this file for a given statement Q, all 11 heuristics are applied for tuning Q (in both, Assisted and Automatic Approaches).

3.1.2 *Automatic Approach.* The automatic approach consists of a middleware, ARe-SQL Mid, that works between the application and the DBMS. This middleware is responsible for: *(i)* automatically receiving SQL statements sent by applications; *(ii)* analyzing and rewriting them whenever necessary, and; *(iii)* submitting the

statements (rewritten or not) to the DBMS. The architecture designed for the automatic approach is illustrated in Figure 6. The main components of this architecture, in addition to components that are also present in Figure 5, are:

—Middleware for SQL Tuning (MST): is responsible for receiving SQL statements from the applications; send them to the agent AST examine them; and receive the statements rewritten (or not) and send them to the DBMS; receive the result of executing these queries and send them to the applications.

—Driver for Data Access (DDA): driver that allows the engine of the middleware (Middleware for SQL Tuning) to send the SQL statements to the target DBMS.

3.2 Implementing ARe-SQL

Given a particular SQL statement Q, the rewriting process, by means of heuristics presented in Table I, needs to sweep the text of Q searching for tuning problems or opportunities (eg, the existence of an unnecessary *DISTINCT* operator) and then changes the syntax of Q, generating a new command Q'. For this, we use a SQL parser called *ZQL* [Thuraisingham et al. 2010].

The *ZQL* parser is written in Java and has been implemented from the JavaCC (Java Compiler Compiler) [Kodaganallur 2004]. The JavaCC is a parser generator for Java applications. A parser generator is a tool that reads a grammar specification and converts them into a program capable of recognizing instructions built with this grammar.

The *ZQL* parser was developed using JavaCC with a SQL grammar. In addition to recognizing SQL statements, *ZQL* stores the elements of a SQL statement in Java data structures. Thus, for a given SQL statement is generated objects that store the column list in *SELECT* clause, the table list in *FROM* clause, arithmetic or logical expressions, among others. These objects were used to implement the heuristic set presented in Table I.

Some other SQL parsers have also been investigated and analyzed, for example, *General SQL Parser* [Merlo et al. 2007] and *SQL Query Parser* [Elliott et al. 2009]. However, the *ZQL* parser was selected to be an easy tool to use, free (open-sourse) and that allows the inclusion of new rules in their SQL grammar.

The *ZQL* receives as input a string containing a SQL statement Q (select, insert, update, delete, commit, rollback or set transaction) and creates a Java data structure to represent Q. For this, *ZQL* uses the Java Class *ZStatement* or one of its subclasses, as: *ZQuery* and *ZUpdate*. Figure 7 shows a piece of the *ZQL* class diagram. It's important to note that SQL queries are represented by the *ZQuery* class.

To illustrate the process of parsing a SQL statement, consider the SQL query Q_c depicted in Figure 8. Now, assume that Q_c was sent to the *ZQL*. Then, to represent Q_c, *ZQL* will create an instance of the *ZQuery* class, denoted by o_c. The method *getSelect()* of o_c will return a Vector of objects V containing one object (instance of the class *ZColumn*) for each column in the *SELECT* clause of Q_c: c_custkey, c_name, c_address and c_phone. Similarly, the method *getFrom()* will return a Vector of objects containing one object for each table in the *FROM* clause of Q_c. It is noteworthy that, in this case, the vector will include a *ZQuery* instance to represent the sub-query SELECT * FROM customer and two objects (instances of the class *ZTable*) to depict the tables nation and region. The method *getWhere()* will return an object of the *ZExpression* class. A *ZExpression* object can contain instances of the *ZQuery* class. The method *getGroupby()* will return an object of the *ZGroupby* class. A *ZGrouby* object can contain instances of the *ZQuery* class.

For each heuristic in HS (Table I) we have implemented an algorithm. Figure 9 describes the process of applying the heuristic H5. This algorithm receives as input a SQL statement Q_j and produces as output a SQL statement Q'_j), a new SQL statement equivalent to Q_j produced by heuristic H5.

The algorithm illustrated in Figure 10 describes the process of rewriting a SQL query Q_j, using the heuristic set presented in Table I. This algorithm receives as input a SQL statement Q_j and produces as output a SQL statement Q'_j), a new SQL statement equivalent to Q_j).

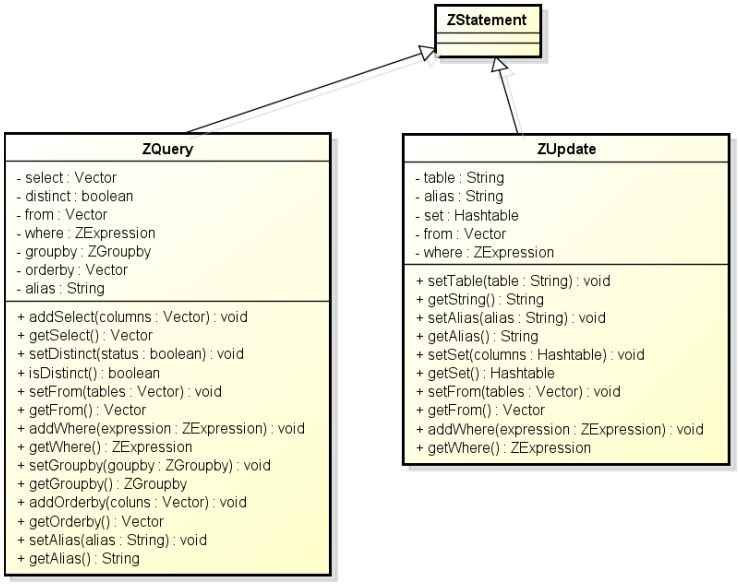

Fig. 7. A Piece of ZQL Class Diagram.

```
SELECT c_custkey, c_name, c_address, c_phone
FROM (SELECT * FROM customer), nation, region
WHERE c_nationkey = n_nationkey AND
n_regionkey = r_regionkey
GROUP BY c_custkey, c_name, c_address, c_phone
```

Fig. 8. SQL expression for Q_c.

4. EXPERIMENTAL RESULTS

In order to show the potentials of ARe-SQL, several experiments have been conducted. The main results achieved so far are presented and discussed in this section. Thus, we first provide information on how the experimentation environment was set up. Thereafter, empirical results are quantitatively presented and qualitatively discussed.

4.1 Simulation Setup

In order to validate both approaches, assisted and automatic, implemented by Are-SQL, we have evaluated ARe-SQL in three different scenarios. In the first scenario, TPC-H benchmark has been used, including its database and workload, which is composed of 23 queries. In the second scenario, a workload of 30 synthetic queries has been executed on TPC-H database. In both scenarios, we have used TPC-H benchmark with scale factor of 2 GB. Finally, in the third scenario, we exploit the database of a system used in several brazilian universities, called Integrated Management System (IMS), and another synthetic workload. The synthetic workloads are designed using the following strategy. Each synthetic workload is formed by 30 SQL queries. Each SQL query was written in order to enable the application of one of the heuristics presented in the Table I.

All experiments were run on a Core i3-2100 (3.10GHz) server, with 4GB RAM and 500 GB HD. PostgreSQL 8.1, Oracle 11g and SQL Server 2008 have been used as database systems. For each experimentation scenario,

```
Function heuristicH5(Q_j)
  oq_j = a ZQuery instance representing Q_j
  Vector operators = oq_j.getWhere().getOperators()
  For each operator ∈ operators do
    If ((operator == ">=") or (operator == ">") or (operator == "<=") or (operator == "<"))
    leftText = text on the left side of the operator
    rightText = text on the right side of the operator
    If rightText contains "ALL"
      sq = the sub-query on rightText
      remove "ALL" from sq
      If ((operator == ">=") or (operator == ">"))
        add "MAX" function in sq
      End If
      If ((operator == "<=") or (operator == "<"))
        add "MIN" function in sq
      End If
    End If
  End If
  End For
  Return Q'_j //The SQL Statement produced by applying H5
End Function
```

Fig. 9. Algorithm for Heuristic H5.

```
Function runHeuristics(Q_j)
  For each heuristic h_i ∈ HS do
    run h_i(Q_j)
  End For
  For each sub-query Q_s in the WHERE clause of Q_j do
    runHeuristics(Q_s)
  End For
  For each sub-query Q_s in the FROM clause of Q_j do
    runHeuristics(Q_s)
  End For
  For each sub-query Q_s in the SELECT clause of Q_j do
    runHeuristics(Q_s)
  End For
  For each sub-query Q_s in the HAVING clause of Q_j do
    runHeuristics(Q_s)
  End For
  Return Q'_j //The SQL Statement produced by the rewriting process
End Function
```

Fig. 10. Recursive algorithm for SQL rewriting.

three different experiments have been performed: i) The workload containing the original queries has been submitted to a database system. The results of this experiment have been be used as baseline; ii) The workload submitted to ARe-SQL advisor (assisted approach). Thereafter, the workload with tuned SQL queries were manually submitted to a database system, and; iii) Each query from original workload has been sent to the automatic approach of ARe-SQL. In this case, after tuning SQL queries, ARe-SQL automatically submits those queries to a database system. Then, in the experimental results, the execution time for the assisted strategy includes the overhead needed to rewriting the queries (that is, to apply the rewriting heuristics). For each test we have executed the set of queries belonging to used workload once, twice, 4, 8, 16, and 32 times. For each different number of executions (iterations), the sum of execution time of the whole set of queries was

Table II. Execution times of the TPC-H queries 18 and 20 in their original formats and after being rewritten.

	PostgreSQL		SQL Server	
	Original	**Rewritten**	**Original**	**Rewritten**
Query TPC-H 18	134.807ms	111.933ms	19s	14s
Query TPC-H 23	912ms	712ms	15s	11s

computed. For instance, in scenario 1 (Full TPC-H Benchmark), to get the execution time for 32 executions, we have execute the set of TPC-H queries 32 times, measured the time interval for executing all queries and computed the sum of the 32 measured execution times.

4.2 Scenario 1: Full TPC-H Benchmark

Figures 11 and 12 bring the result of using the assisted and automatic approach of ARe-SQL running on PostgreSQL and SQL Server, respectively. It is important to observe that from the 23 queries that comprise the TPC-H benchmark, two (18 and 23) have been rewritten by ARe-SQL advisor (see Section 3.1.1). Regarding the results presented in Figures 11 and 12, the assisted approach had a small decrease in workload runtime. This can be explained by the fact that only two TPC-H queries have presented opportunities for tuning. Table II shows the execution time of TPC-H queries 18 and 23 in their original formats and after being rewritten. Is important to emphasize that this is not a bad result, since the TCP-H queries were written by experts in SQL.

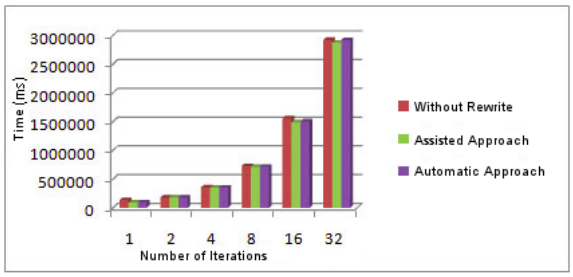

Fig. 11. Benchmark TPC-H on PostgreSQL

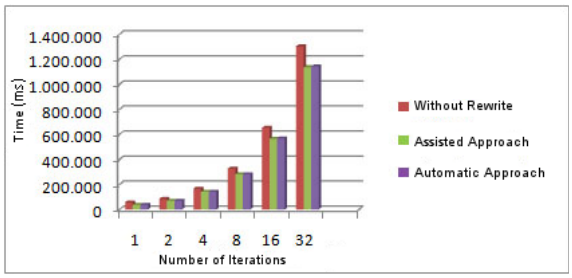

Fig. 12. Benchmark TPC-H on SQL Server

On Oracle, however, TPC-H queries 18 and 23 were not rewritten by ARe-SQL advisor, since Oracle implements heuristic H8. So, the ARe-SQL automatic approach had a slight worsening compared to baseline (24.95%). This is because the automatic approach had the overhead of trying to rewrite both queries.

4.3 Scenario 2: TPC-H Database with Synthetic Workload

Figures 14, 15 and 16 show that using ARe-SQL advisor ensures a significant reduction in query response. On the other hand, ARe-SQL's automatic approach presents smaller benefits than ARe-SQL advisor, since it involves the overhead of tuning the SQL statements received in runtime.

It's important to note that in the test on Oracle (Figure 16) automatic approach presents a small decrease in the execution time of the workload. This is explained by the fact that from the total of eleven heuristics five are

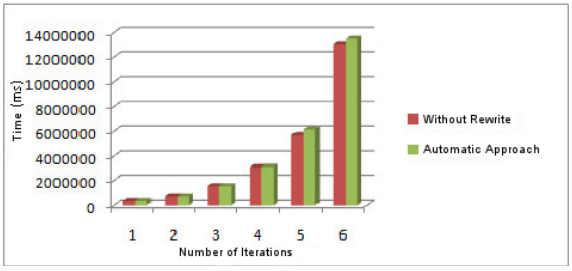

Fig. 13. Benchmark TPC-H on Oracle

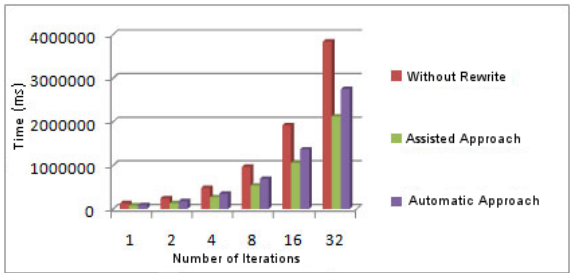

Fig. 14. Synthetic queries in TPC-H database on PostgreSQL

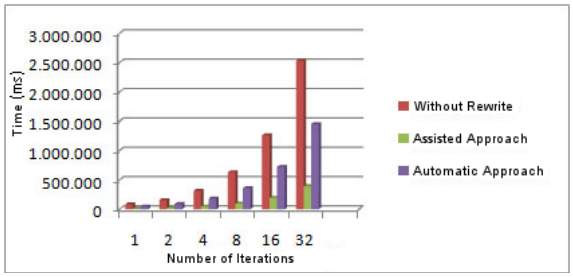

Fig. 15. Synthetic queries in TPC-H database on SQL Server

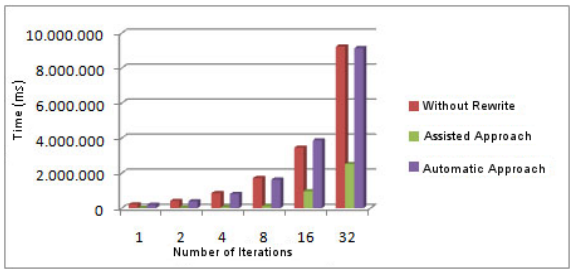

Fig. 16. Synthetic queries in TPC-H database on Oracle

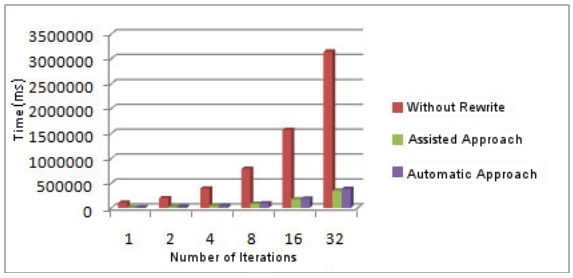

Fig. 17. Synthetic queries in IMS database on PostgreSQL.

already implemented by Oracle (H4, H5, H6, H7 and H8). Besides, from the six remaining heuristics, three of them (H9, H10 and H11) make use of statistical information, and, therefore, require additional access to DBMS, which significantly increased the overhead to rewrite SQL queries.

In the experiment with TPC-H Database and Synthetic Workload on Oracle, the runtime for the 32 iterations of the original workload (without rewriting) took 673.28s, while the runtime for the 32 iterations of the workload generated after applying the assisted approach lasted 49.43 seconds (Figure 17). In this case, assisted approach yielded a gain of 623 seconds (92.5%).

4.4 Scenario 3: IMS Database with Synthetic Workload

For this scenario, only PostgreSQL has been used, as the IMS database is only available for that DBMS. Again, the proposed approaches have provided a high reduction in time execution of the submitted workload (Figure 17). In the experiment with IMS Database and Synthetic Workload on PostgreSQL, the runtime for the 32 iterations of the original workload (without rewriting) took 314.32s, while the runtime for the 32 iterations of the workload generated after applying the assisted approach lasted 34.72 seconds. In this case, assisted approach yielded a gain of 280 seconds (89%).

4.5 Overhead Analysis

To analyze the overhead in SQL rewriting process we made an experiment using TPC benchmark. The idea was measure the time needed to rewrite each one of the 23 SQL queries in the TPC benchmark workload. For this, we have used as a baseline the rewritten time produced by *Quest SQL Optimizer for Oracle*. This tool was chosen because it is widely used in real scenarios.

Table III shows a piece of the report generated by *Quest SQL Optimizer for Oracle* using TPC-H queries with input. This report indicated that from the total of 23 SQL queries 16 could be rewritten successfully. Only 7 SQL queries couldn't be tuned (1, 6, 10, 13, 17, 19 and 23). However, after to perform (with empty cache), for each one of these 16 SQL queries, the best SQL command (w.r.t cost estimate) generated by the *Quest SQL Optimizer for Oracle* tool, we noted that only 5 SQL queries had reduced their runtimes (queries 2,

Table III. *SQL Optimizer for Oracle* using TPC-H *benchmark*.

Query	No. of generated SQL commands	Rewriting Overhead	No. of SQL clauses with cost estimate less than the original SQL	No. of SQL clauses with response time estimate less than the original SQL	Original SQL response time	Execution time for the best rewritten SQL
2	115	116s	3	3	310ms	128ms
3	53	58s	3	1	6393ms	4900ms
4	14	15s	1	0	4204ms	1 min
5	228	229s	5	0	4658ms	4666ms
7	110	111s	1	0	4658ms	7808ms
11	139	140s	8	4	600ms	259ms
12	20	21s	1	1	1995ms	1993
14	5	6s	1	0	512ms	538ms
16	35	36s	2	0	278.244ms	280.190 ms
18	160	161s	1	0	17.384ms	22.306 ms
20	92	93s	8	3	3.261ms	3.212ms
21	120	121s	5	0	8.660ms	8.787ms
22	1	1s	1	0	407ms	418ms

3, 11, 12 and 20). In practice, queries 4, 5, 7, 8, 9, 14, 16, 18, 21 and 22 had a worsening in their execution times. It's important to note that in some cases, like queries 4 and 7, the rewritten SQL response time was much greater than the original query runtime. Additionally, *Quest SQL Optimizer for Oracle* tool is DBMS-specific (running only with Oracle). Besides, in some cases, this tool may generate tens or hundreds of rewritten SQL instructions for tune the same SQL statement, such as, for example, for query 5, 228 different rewritten SQL instructions were generated. This fact could increase so much the rewritten time, producing a high overhead, and making this tool unfeasible for online scenarios, that is, those where the SQL statement needs to be adjusted automatically (at runtime). On the other hand, we submitted the synthetic workload used at the scenario 2 to ARe-SQL and measured the time needed to rewrite each one of the 30 SQL queries. ARe-SQL spent in average 29.83s to rewrite a query on Oracle, 19.33s on SQL Server and 40.66 on PostgreSQL.

4.6 Additional Discussion

It's important to note that the heuristics provide different gains. Then, a particular heuristic can provides a much greater benefit than another. However, in all queries that formed the synthetic workloads (used in scenarios 1 and 2), the used heuristics provided some gain (a positive profit). That is, any query produced by ARe-SQL had a response time greater than the original query execution time.

Additionally, the DBA has the ability to interact with the tuning suggestions provided by the *advisor*, in the assisted strategy. For example, the DBA can apply only a subset of the heuristics in the heuristic set (HS), the DBA can force the application of heuristics that have not been effectively used by the *advisor*, the DBA can choose the order in which the heuristics are performed, among other options. Besides, the DBA can choose and set a subset of the available heuristics so that they are always executed in the rewriting process for queries in general or for SQL statement in particular. In this case, this information is stored in the Tuning Settings (TS) repository.

So, before rewriting a SQL statement Q, the AST agent checks, at the Tuning Settings (TS) component, if there is a subset of heuristics selected (defined) for rewriting Q. If not, all available heuristics are used to rewriting Q.

5. CONCLUSIONS

In this article, we propose an online, automatic and non-intrusive approach for rewriting SQL queries, denoted ARe-SQL. The key goal of ARe-SQL is to assist optimizers in building efficient query physical execution plan. For this, ARe-SQL rewrites SQL statements, using a set of 11 heuristics. Based on ARe-SQL, two different strategies for SQL-query rewriting, denoted assisted and automatic, were implemented. These strategies were evaluated in three different DBMSs considering three different scenarios. The experimental results indicate that both strategies can provide performance gains. In some cases, such performance gains reach 92.5%. The proposed solutions are applicable to situations where the query optimizer cannot produce optimal plans, even using access methods and assessment strategies supported by the DBMS. ARe-SQL is even more important in contexts in which SQL statements are written by non-experts. As future work we intend to investigate and implement heuristics based on query hints to tune SQL queries.

REFERENCES

BELKNAP, P., DAGEVILLE, B., DIAS, K., AND YAGOUB, K. Self-tuning for sql performance in oracle database 11g. In *Proceedings of the IEEE International Conference on Data Engineering*. ICDE '09. IEEE Computer Society, Washington, DC, USA, pp. 1694–1700, 2009.

BRUNO, N., CHAUDHURI, S., AND RAMAMURTHY, R. Power hints for query optimization. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, pp. 469–480, 2009.

CHAUDHURI, S. AND WEIKUM, G. Foundations of automated database tuning. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, Seoul, Korea, pp. 1265–1265, 2006.

DAGEVILLE, B. AND DIAS, K. Oracle's self-tuning architecture and solutions. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. IEEE Computer Society, Oracle, USA, 2006.

DIAS, K., RAMACHER, M., SHAFT, U., VENKATARAMANI, V., AND WOOD, G. Automatic performance diagnosis and tuning in oracle. In *Biennial Conference on Innovative Data Systems Research*. pp. 84–94, 2005.

ELLIOTT, B., CHENG, E., THOMAS-OGBUJI, C., AND OZSOYOGLU, Z. M. A complete translation from sparql into efficient sql. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*. IDEAS '09. ACM, New York, NY, USA, pp. 31–42, 2009.

HERODOTOU, H. AND BABU, S. Automated sql tuning through trial and (sometimes) error. In *Proceedings of the Second International Workshop on Testing Database Systems*. ACM, New York, NY, USA, 2009.

KODAGANALLUR, V. Incorporating language processing into java applications: A javacc tutorial. *IEEE Software* 21 (4): 70–77, 2004.

KRISHNAPRASAD, M., LIU, Z. H., MANIKUTTY, A., WARNER, J. W., ARORA, V., AND KOTSOVOLOS, S. Query rewrite for xml in oracle xml db. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, Toronto, Canada, pp. 1134 – 1145, 2004.

MARKL, V., LOHMAN, G. M., AND RAMAN, V. Leo: An autonomic query optimizer for db2. *IBM Syst. J.* 42 (1): 98–106, Jan., 2003.

MERLO, E., LETARTE, D., AND ANTONIOL, G. Automated protection of php applications against sql-injection attacks. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. CSMR '07. IEEE Computer Society, Washington, DC, USA, pp. 191–202, 2007.

OPTIMIZER, E. D. Embarcadero db optimizer xe, 2010. Available: http://www.embarcadero.com/products/db-optimizer-xe. June 2011.

QUEST, S. O. F. O. Quest sql optimizer for oracle, 2010. Available: http://www.quest.com/SQL-Optimizer-for-Oracle. June 2011.

RAMAKRISHNAN, R. AND GEHRKE, J. *Database Management Systems*. McGraw Hill, 2002.

STUDIO, I. O. D. Ibm optim development studio, 2010. Available: http://www-01.ibm.com/software/data/optim/development-studio. June 2011.

THURAISINGHAM, B., KHADILKAR, V., GUPTA, A., KANTARCIOGLU, M., AND KHAN, L. Um framework para a avaliaçãoo de desempenho de bancos dedados. In *Proceedings of the International Conference on Networking, Applications and Worksharing*. Texas, USA, 2010.

WEIKUM, G., MOENKEBERG, A., HASSE, C., AND ZABBACK, P. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Proceedings of the International Conference on Very Large Data Bases*. pp. 20–31, 2002.