SQOPI: Semantic Query Optimization Framework

Mohamed Mounir Hassan
Faculty of Computers and Informatics,
Zagazig University,
Egypt

Ahmed Mohammed Sultan
Faculty of Computers and Informatics,
Zagazig University,
Egypt

ABSTRACT
Semantic query optimization uses semantic knowledge in databases to rewrite queries and logic programs for the purpose of more efficient query evaluation. There has been a large body of work in the area of semantic query optimization. But, unfortunately, till now no commercial application of sematic query optimization techniques has received wide attention. In this paper, we address this problem by developing a unified framework (Application Programming Interface) called SQOPI that could be used by any application developer to semantically optimize queries executed against relational database regardless of DBMS type used. Our results show that SQOPI improves both time and I/O efficiency.

General Terms
Databases, DBMS

Keywords
Semantic Query Optimization, Query Rewrite.

1. INTRODUCTION
It has become clear that relational database systems became the predominant technology for storing, handling, and querying data only after a great improvement in the efficiency of query evaluation in such systems. The key factor in this improvement was the introduction and development of query optimization techniques. However, the traditional types of optimization exploit to a limited extent the semantic information about the stored data. Researchers [1-5] recognized that such information could be used for further query optimization, and developed a set of techniques called semantic query optimization (SQO). SQO uses the integrity constraints associated with the database to improve the efficiency of query evaluation. The techniques most often discussed in literature included the following:

- **Join Elimination:** for some joins the result of the query is known in advance. It may be efficient not to evaluate these queries at all.
- **Join Introduction:** Adding a join can help if relation is small compared to original relations and highly selective.
- **Predicate Elimination:** If predicate known to be always true, can be eliminated from query (DISTINCT clause on Primary Key)
- **Predicate Introduction:** New predicates on indexed attributes can result in a much faster access plan.
- **Detecting the Empty Answer Set:** If query predicates inconsistent with integrity constraints, the query does not have answer.

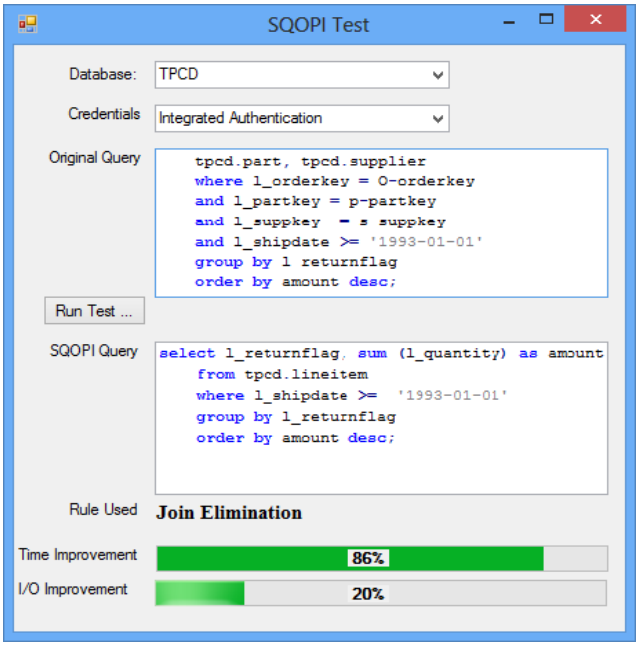

Figure 1: SQOPI Demo Application

SQO is a largely unutilized technique, despite the prevailing view that SQO is useful. It is clear that Semantic optimization could potentially provide much greater performance improvements than more traditional algebraic optimizers, but that few commercial products, if any, do much in the way of semantic optimization.

In [6], the authors put forward two reasons why SQO has never caught on in the commercial world where most databases are RDBMS:

- SQO is designed for deductive databases where the relatively high cost of applying complex rules (in comparison to much less complex rules in relational databases) is more likely to make the extra computational effort of implementing SQO worthwhile;
- CPU speeds are not high enough for the extra computational cost of SQO to be acceptable.

In [7], the authors consider the role of schema constraints in capturing business rules and identify four reasons for SQO techniques not being employed:

- The potential for using schema constraints to capture business rules is only now being realized, so opportunities for SQO have until now seemed limited.

- The expense of checking schema constraints at data insert or update time has limited the use of schema constraints, so opportunities for SQO have until now seemed limited.
- Many semantic rules which could potentially be used as schema constraints are simply not discovered.
- Even if a semantic rule is discovered there may be no justification for making it a schema constraint.

2. RELATED WORK

This section summarize and describe the related work on main types of SQO as they have been classified by various researchers [1, 4, 6, 8-12] and then put our system in context. There are four broad categories of Semantic Query Optimization:

Detection of Unsatisfiable Queries

A query is unsatisfiable if it cannot logically return any rows. The detection of such queries is identified as a major advantage by all researchers into SQO [13-17]. For example, [18] reports savings made by detecting unsatisfiable queries are an order of magnitude greater than other optimizations. The advantage arises simply because an unsatisfiable query need not be posed to the database at all, resulting in a 100% saving, neglecting the cost of detecting that unsatisfiability.

Restriction Removal

A query restriction may be deduced to be redundant and its elimination simplifies the query by eliminating the need to process that restriction.

Restriction Introduction

A query restriction may imply an additional (redundant) restriction which, when introduced, increases efficiency. This typically occurs when the introduced restriction involves an indexed attribute [10, 19].

Join Removal

Join removal occurs when a redundant table join is detected and avoided. The Join operation in RDB is typically the most expensive [20, 21], so it is reasonable to expect its elimination could greatly increase query efficiency.

Our system departs from the previously mentioned work in that it doesn't investigate new ways for applying semantic query optimization. SQOPI tries to make use of all these techniques to provide a framework that carries the previously mentioned SQO techniques to the commercial domain. In [22], the authors comment that relational query optimizers ignore many semantic optimization opportunities arising from a knowledge of the schema semantics. In [21], the author comments that semantic optimization could potentially provide much greater performance improvements than more traditional algebraic optimizers, but that few commercial products, if any, do much in the way of semantic optimization. SQOPI comes here as a tool to bridge this gap.

3. MOTIVATION

Despite the negative comments provided by previous authors, semantic query optimization is worthwhile and should be revisited. We now make some observations concerning changes in computer hardware typically employed in the database industry. We argue that these changes have made SQO significantly more attractive than in the last decade when much foundational work in SQO was done.

- First, average data volumes have increased by several orders of magnitude, driven by the rising use of data warehouses and the falling cost of disk storage. Therefore even small increases in query efficiency offered by SQO may now be worthwhile.
- Second, available RAM has increased, typically by a factor of three or four, driven by the falling cost of RAM. In DBMS, the impact of increasing available main memory is seen in the increasing proportion of the database that runs in memory. Ultimately, when sufficient main memory is made available, most of the database runs in memory most of the time and, crucially, disk activity is minimized. In RDBMS, this equates to most queried table data plus most procedural and SQL code being held in cache most of the time. In this environment, any query optimization that reduces disk activity is likely to be significant.
- Third, Distributed databases, where a single logical database comprises several geographically distant nodes, are now commonly deployed by businesses across their WANs. Distributed databases introduce delays in query answering, primarily because of the cost of transporting data between physical nodes. In this environment, data is typically partitioned across physical nodes according to simple semantic rules. These rules may then be utilized by a simple semantic query optimizer to minimize communication costs. For example, suppose the database is distributed across three physical nodes located in Cairo, Riyadh and Istanbul. Each node holds data strictly for its own Sales Office. A simple semantic query optimizer is constructed which determines which Sales Office is being queried and routes the query to the correct nodes while preventing the query from being passed to the remaining nodes.

4. SQOPI ARCHITECTURE

SQOPI extends the traditional classic relational database access APIs by adding the semantic query optimizer.

Our C# version of the framework builds on the top of the ADO.NET [23]. Mainly because ADO.NET is an integral part of the .NET Framework, providing access to relational data, XML documents, and application data. ADO.NET supports a variety of development needs. And thus our framework could be used for creating database-client applications and middle-tier business objects used by applications.

Of the tens of objects available in ADO.NET, we chose only three objects (i.e. Connection, Command and DataReader objects) to be included and extended for our framework model and to be exposed to the application developer. Our decision is backed by two reasons: first: these objects represents the core objects that formulates the whole framework inner workings and second, and second: they are fairly enough for developing complete data manipulation Apps.

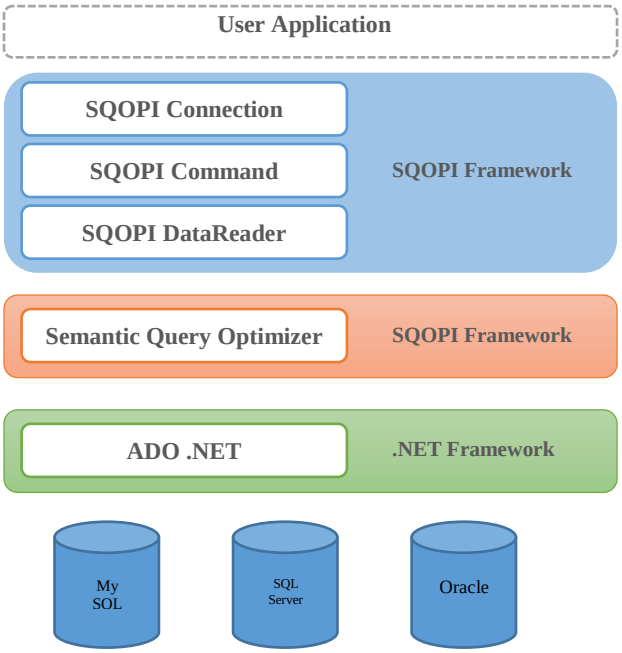

Figure 2: Proposed Framework Architecture

SQOPI connection objects follow the same behavior as those provided by ADO.NET framework. Actually they inherit from these objects. Each SQOPI connection consumes a certain amount of resources on the database server and these resources are not infinite. Connection providers implement connection pooling. If you create database connections, they are held in a pool. When the programmer want a connection for an application, the provider extracts the next available connection from the pool. When the application closes the connection, it returns to the pool and makes itself available for the next application that wants a connection. This means that opening and closing a database connection is no longer an expensive operation. If you close a connection, it does not mean you disconnect from the database. It just returns the connection to the pool. If you open a connection, it means it's simply a matter of obtaining an already open connection from the pool.

The Command object is used to execute a single query against a database. The query can perform actions like creating, adding, and retrieving, deleting or updating records. If the query is used to retrieve data, the data will be returned as a DataReader object. This means that the retrieved data can be manipulated by properties, collections, methods, and events of the DataReader object. The major feature of the Command object is the ability to use stored queries and procedures with parameters.

It's worthy to mention that semantic query optimization subcomponent of the framework is not exposed to the application developer and has no corresponding API objects for working with it. Although the whole framework is dependent on this component, we decided to make it invisible form the application programmer point of view. Since the optimizer produces an equivalent query with hopefully less execution cost, it would be better to call it implicitly and automatically for every query. However, the application developer can set a binary flag (*OPTIMIZER_ON*) to switch the semantic optimization on or off for test and debugging purposes.

5. PROGRAMMING PATTERN

The programming pattern of our framework APIs follows the same line as ADO.NET. We intended to design it this way and make it compatible, because it would much easier to convert the data access code to SQOPI-enabled code by just renaming the core objects names. This in fact may encourage developers to take this step without worrying about redesigning their software. Below is a sample code for calling query against the TPCD database and getting the results back via the SQOPIDataReader object, looping through the records returned by and displaying each record as String (actually comma separated string).

```
void executeQuery(DateTime date) {
using( SQOPIConnection con =
 new SSQOPIConnection (CONN_STRING) ) {
 con.Open();
 SQOPICommand cmd = con.CreateCommand();
 cmd.CommandText =
        @"select l_returnflag, sum(l_quantity) as
        amount
        from tpcd.lineitem,  tpcd-orders,
        where l_orderkey = O_orderkey
        and l_partkey = p_partkey
        and l_suppkey  = s_suppkey
        and l_shipdate >= @date
        group by l_returnflag
        order by amount desc; ";
 cmd.Parameters.Add("@date",date);
 SQOPIDataReader r = cmd.ExecuteReader();
 while(r.Read()) {
 Console.WriteLine(r.ToString());
} } }
```

While this code sample depends on the SQOPI version developed in C# and utilizes ADO.NET, our framework model and programming pattern applies to any other relational database access API like JDBC.

6. EXPERIMENTAL RESULTS

All of our experiments were done on Oracle RBMS.

Experimental Setup

We do not simply measure elapsed time in order to judge the cost of a query batch. Rather, we use the Oracle system itself to take its own measurements, which are very precise. We use a software tool, **TKPROF**, to take these measurements and this enables us to look at the query cost in a number of different ways. For example, we may look at the CPU time separately from the number of disk blocks physically fetched from disk, or the number of query rows fetched. We use three query metrics (**Table 1**) inspired by authors in [24] to measure the true query cost. The metrics we use are all statistics output by the Oracle database tool **TKPROF**. The software tool **TKPROF** reports each SQL statement executed along with the resources it has consumed, the number of times it was called, and the number of rows which it processed. This information may be automatically accumulated in an operating system file over an arbitrary period of time and may include the resources utilized by one or many simultaneous sessions accessing the target database. TKPROF is the main method by which computational cost is measured in our own.

Oracle further distinguishes between three phases or calls when an SQL statement is processed: PARSE, EXECUTE and FETCH.

- **PARSE** Translates the SQL statement into an execution plan, including checks for proper security authorization and checks for the existence of tables, columns and other referenced objects.
- **EXECUTE** Actual execution of the statement by Oracle. For INSERT, UPDATE, and DELETE statements, this modifies the data. For SELECT statements, this identifies the selected rows.
- **FETCH** Retrieves rows returned by a query. Fetches are only performed for SELECT statements.

We use R_{com} to report the sum of these three calls as a single metric. R_{com} is calculated as following:

$$R_{com} = \frac{1}{3}\sum_{i=1}^{3}\frac{COST_{opt}^{i}}{COST_{norm}^{i}}$$

Note that $COST^{i}$represents each of the three cost metrics we have shown in the **Table 1**.

Table 1 : Query Evaluation Metrics

Metric	*Meaning*	$R_{cost} = \frac{COST_{opt}}{COST_{norm}}$
CPU	Total CPU time in seconds for all parse, execute, or fetch calls for the statement.	R_{cpu}
ELAPSED	Total elapsed time in seconds for all parse, execute, or fetch calls for the statement.	$R_{elapsed}$
DISK	Total number of data blocks physically read from the data files on disk for all parse, execute, or fetch calls.	R_{disk}
COMBINED	The average of the other three metrics. This metric is only ever reported as a ratio	R_{com}

Results

Keeping in mind that SQOPI is data manipulation framework targeted primarily to database application developers, assessing the usability SQOPI should include the evaluation of many factors like developer friendliness, scalability, usability and performance. Among all evaluation factors, performance is considered a key factor in judging the usability of this framework. Here we present the detailed evaluation of SQOPI performance based on the metrics discussed previously. We built a demo application (Figure 1) for this purpose. The application takes a batch of queries from input files, executes them one by one and reports back the results in output file. Instead of picking a random batches of queries, we preferred to report our results on a set of queries where a variety of query optimization techniques could be applied and tested. Ultimately we settled on the same queries (Table 2) used in work by [6] on the TPC benchmark [25].

For a 1GB database Scale Factor (SF) we got the following experimental results. Figure 3 presents the total CPU time profiling for all parse, execute, or fetch calls of the queries. We can see that, on average, the optimized query is 2x better than the original query. Figure 4 and Figure 5 show the ELAPSED cost and the DISK cost respectively. Noticeably, there isn't any major cost saving in both factors except for Queries Q3, Q6 since they are optimized using the empty answer set detection technique. The total cost savings are summarized in Figure 6.

To assess the confidence in our results we performed the same experiments on a larger database with as scale factor of 10GB. Figures Figure 7, Figure 8 and Figure 9 show the corresponding CPU, ELAPSED and DISK costs respectively, while Figure 10 shows the total cost saving. The results reveal a very similar pattern with the exception of cost savings associated with Q5. The cost savings for Q5 increased when we moved to 10G Scale. A potential cause of this is that a less percentage of the newly generated data conformed to the predicates of the generated optimized query.

Table 2: Queries

1	SELECT LINEITEM.L_RETURNFLAG,SUM(LINEITEM.L_QUANTITY) AS AMOUNT FROM lineitem,orders,PART,supplier WHERE L_ORDERKEY = O_ORDERKEY AND L_PARTKEY = P_PARTKEY AND L_SUPPKEY = S_SUPPKEY AND L_SHIPDATE >= TO_DATE('01/01/1993','dd/mm/yyyy') GROUP BY L_RETURNFLAG ORDER BY amount DESC;
2	SELECT o_orderstatus, SUM(o_totalprice) AS price FROM orders, customer WHERE o_custkey = c_custkey AND o_orderdate >= to_date('1/1/1993','dd/mm/yyyy') AND o_orderpriority = '2-HIGH' GROUP BY o_orderstatus ORDER BY price DESC;
3	SELECT C_CUSTKEY, C_NAME, C_ACCTBAL FROM SUPPLIER, CUSTOMER, ORDERS, LINEITEM WHERE C_CUSTKEY = O_CUSTKEY AND O_ORDERDATE >= TO_DATE('1/1/1993','dd/mm/yyyy') AND O_ORDERKEY = L_ORDERKEY AND L_RETURNFLAG = 'R' AND S_NAME = C_NAME;
4	SELECT SUM(L_EXTENDEDPRICE * (1-L_DISCOUNT)) AS revenue FROM LINEITEM WHERE L_EXTENDEDPRICE <=1000 AND L_DISCOUNT <= 0.05 AND L_RECEIPTDATE <= to_date('1/1/1997','dd/mm/yyyy');
5	SELECT LINEITEM.L_RETURNFLAG, INEITEM.L_LINESTATUS, SUM(LINEITEM.L_QUANTITY) AS SUM_L_QUANTITY FROM LINEITEM, ORDERS WHERE LINEITEM.L_ORDERKEY =ORDERS.O_ORDERKEY AND ORDERS.O_ORDERDATE >= to_date('1/10/1995','dd/mm/yyyy') AND LINEITEM.L_SHIPDATE <=to_date('7/10/1995','dd/mm/yyyy') GROUP BY LINEITEM.L_RETURNFLAG, LINEITEM.L_LINESTATUS ORDER BY LINEITEM.L_RETURNFLAG, INEITEM.L_LINESTATUS;
6	SELECT ORDERS.o_orderkey, ORDERS.O_ORDERDATE, ORDERS.o_shippriority FROM ORDERS, LINEITEM WHERE ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY AND ORDERS.o_totalprice >= 65000 AND ORDERS.O_ORDERDATE > to_date('01/01/1994','dd/mm/yyyy') AND LINEITEM.L_COMMITDATE < to_date('01/07/1993','dd/mm/yyyy') ORDER BY ORDERS.O_ORDERDATE;
7	SELECT ORDERS.o_orderkey, ORDERS.O_ORDERDATE, ORDERS.o_shippriority FROM ORDERS, LINEITEM WHERE ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY AND ORDERS.o_totalprice >= 65000 AND ORDERS.O_ORDERDATE > to_date('01/01/1994','dd/mm/yyyy')

```
AND LINEITEM.L_COMMITDATE <
to_date('01/07/1993','dd/mm/yyyy')
ORDER BY
  ORDERS.O_ORDERDATE;
```

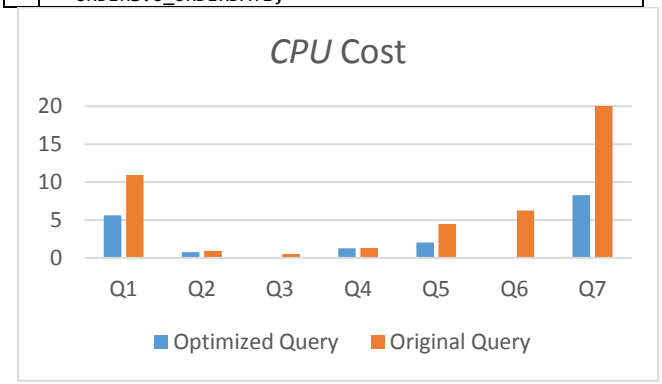

Figure 3: CPU Cost for queries 1G

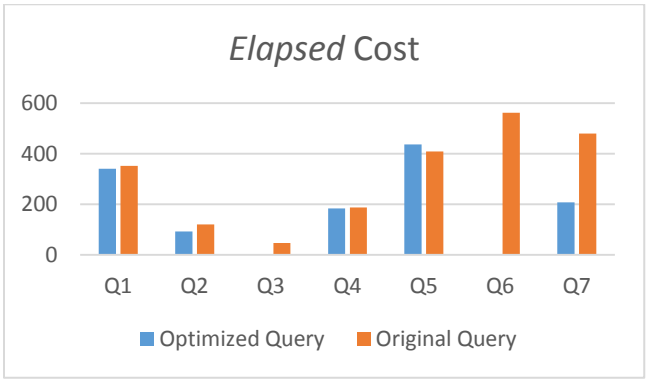

Figure 4:Elapsed Cost for queries 1G

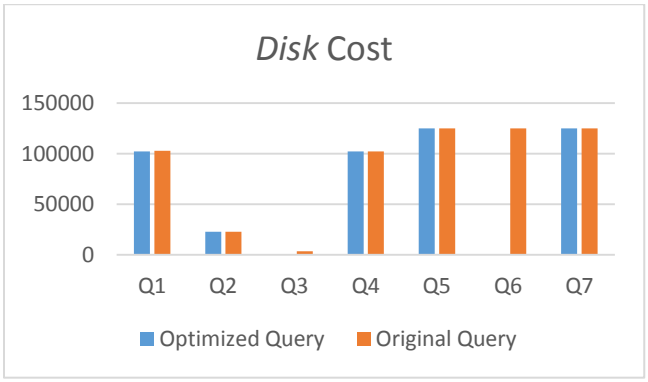

Figure 5:Disk Cost for queries 1G

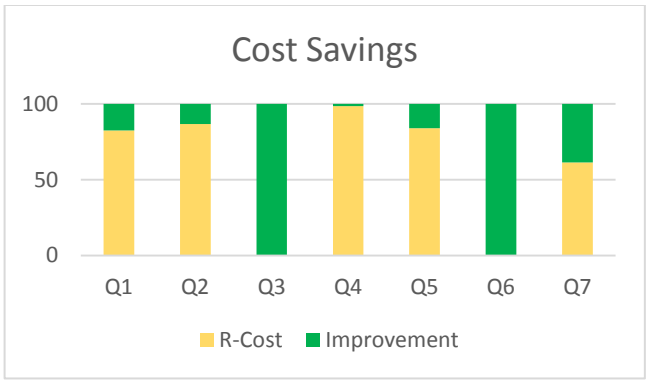

Figure 6 : Total Cost Saving 1G

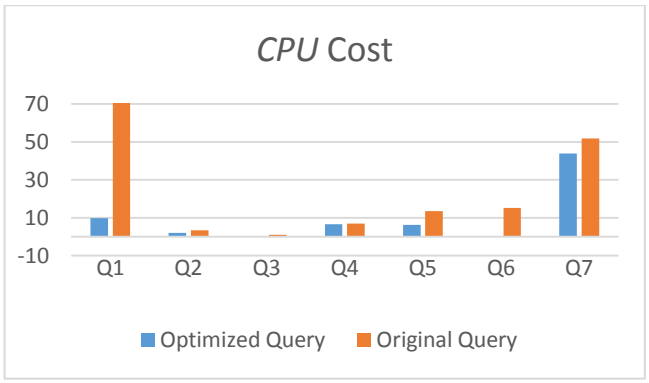

Figure 7:CPU Cost for queries 10G

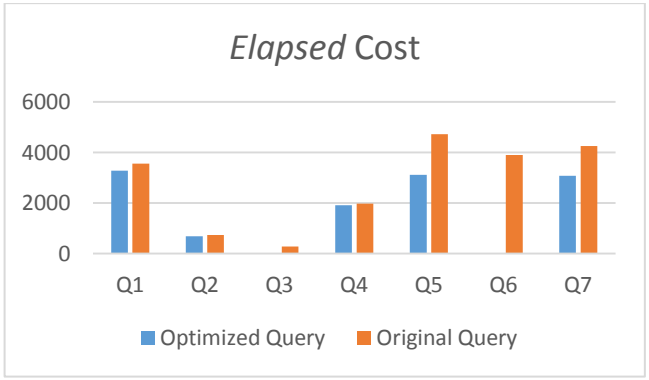

Figure 8: Elapsed Cost for queries 10G

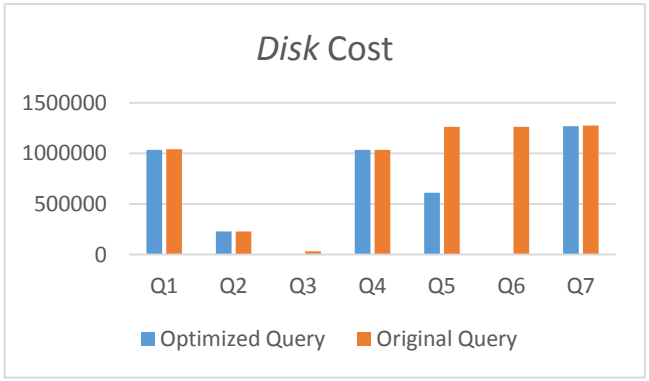

Figure 9: Disk Cost for queries 10G

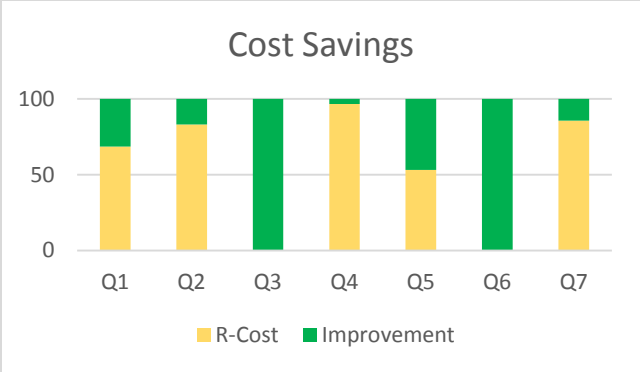

Figure 10: Total Cost Saving 10G

7. CONCLUSIONS AND FUTURE WORK

In this paper we discussed the design and implementation of the SQOPI Semantic Query Optimization framework. We showed the detailed architecture and discussed the rationale behind each subcomponent. SQOPI integrates easily with existing data manipulation Apps. Actually it takes nothing more than adding SQOPI as a library to the project code and changing data manipulation classes to refer to different (but similar and compliant) SQOPI classes. The experimental results reveals a promising saving in the query execution cost after using SQOPI. Our experimental considered the oracle DBMS. For future work, we seek investigating the usability and performance of our framework on other commercial DBMS like SQL Server.

8. REFERENCES

[1] U. S. Chakravarthy, J. Grant, and J. Minker, "Logic-based approach to semantic query optimization," *ACM Transactions on Database Systems (TODS)*, vol. 15, pp. 162-207, 1990.

[2] M. Hammer and S. B. Zdonik, "Knowledge-based query processing," in *Proceedings of the sixth international conference on Very Large Data Bases-Volume 6*, 1980, pp. 137-147.

[3] M. Jarke, J. Clifford, and Y. Vassiliou, "An optimizing prolog front-end to a relational query system," *ACM SIGMOD Record*, vol. 14, pp. 296-306, 1984.

[4] J. J. King, "Quist: A system for semantic query optimization in relational databases," in *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*, 1981, pp. 510-517.

[5] S. T. Shenoy and Z. M. Ozsoyoglu, "Design and implementation of a semantic query optimizer," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 1, pp. 344-361, 1989.

[6] Q. Cheng, J. Gryz, F. Koo, T. C. Leung, L. Liu, X. Qian, *et al.*, "Implementation of two semantic query optimization techniques in DB2 universal database," in *VLDB*, 1999, pp. 687-698.

[7] P. Godfrey, J. Gryz, and C. Zuzarte, "Exploiting constraint-like data characterizations in query optimization," in *ACM SIGMOD Record*, 2001, pp. 582-592.

[8] J. Chomicki, "Querying with intrinsic preferences," in *Advances in Database Technology—EDBT 2002*, ed: Springer, 2002, pp. 34-51.

[9] B. G. Lowden and J. Robinson, "Improved data retrieval using semantic transformation," in *Database and Expert Systems Applications*, 2004, pp. 391-400.

[10] B. G. Lowden and J. Robinson, "Constructing inter-relational rules for semantic query optimisation," in *Database and Expert Systems Applications*, 2002, pp. 587-596.

[11] J. Grant, J. Gryz, J. Minker, and L. Raschid, "Semantic query optimization for object databases," in *Data Engineering, 1997. Proceedings. 13th International Conference on*, 1997, pp. 444-453.

[12] H. H. Pang, H. J. Lu, and B. C. Ooi, "An efficient semantic query optimization algorithm," in *Data Engineering, 1991. Proceedings. Seventh International Conference on*, 1991, pp. 326-335.

[13] S.-C. Yoon, L. J. Henschen, E. Park, and S. Makki, "Using domain knowledge in knowledge discovery," in *Proceedings of the eighth international conference on Information and knowledge management*, 1999, pp. 243-250.

[14] B. Genet and G. Dobbie, "Is semantic optimisation worthwhile," in *Proceedings of the 21st Australasian Computer Science Conference*, pp. 245-256.

[15] X. Zhang and Z. M. Ozsoyoglu, "Implication and referential constraints: A new formal reasoning," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 9, pp. 894-910, 1997.

[16] C.-N. Hsu and C. A. Knoblock, "Discovering robust knowledge from databases that change," *Data Mining and Knowledge Discovery*, vol. 2, pp. 69-95, 1998.

[17] P. Godfrey, J. Gryz, and J. Minker, *Semantic query optimization for bottom-up evaluation*: Springer, 1996.

[18] A. Sayli and B. Lowden, "The use of statistics in semantic query optimization," *CYBERNETICS AND SYSTEMS RESEARCH*, pp. 991-996, 1996.

[19] S. T. Shenoy and Z. M. Ozsoyoglu, *A system for semantic query optimization* vol. 16: ACM, 1987.

[20] D. K. Burleson, *Practical application of object-oriented techniques to relational databases*: Wiley-QED Publishing, 1994.

[21] C. J. Date, *An Introduction To Database Systems, 8/E*: Pearson Education India, 2006.

[22] A. C. Bloesch and T. A. Halpin, "Conceptual queries using ConQuer-II," in *Conceptual Modeling—ER'97*, ed: Springer, 1997, pp. 113-126.

[23] "ADO.NET Framework," *http://msdn.microsoft.com/en-us/library/aa286484.aspx*.

[24] B. H. Genet and A. Hinze, "Open issues in semantic query optimization in relational DBMS," 2004.

[25] T. P. P. Council, "'TPC Benchmark B," *Standard Specification, Waterside Associates, Fremont, CA*, 1990.