Volume 2, No. 2, Mar-Apr 2011

ISSN No. 0976-5697

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

SQL Query Dissembler –A Self Optimizing Autonomic System

Ms. Shanta Rangaswamy*
Department of Computer Science and Engineering
R.V. College of Engineering,
Bangalore, India
shantharangaswamy@rvce.edu.in

Dr. Shobha G
Department of Computer Science and Engineering
R.V. College of Engineering,
Bangalore, India
deanpgcs@rvce.edu.in

Abstract: Current database workloads often consist of a mixture of short OnLine Transaction Processing (OLTP) queries and typical large complex queries such as OnLine Analytical Processing (OLAP). OLAP queries usually involve multiple joins, arithmetic operations, nested sub-queries, and other system or user-defined functions and they typically operate on large data sets. These resource intensive queries can monopolize the database system resources and negatively impact the performance of smaller, possibly more important, queries. In this paper, we present an approach to managing the execution of large queries that involves the decomposition of large queries into an equivalent set of smaller queries and then scheduling the smaller queries so that the work is accomplished with less impact on other queries. Here we implement a SQL disassembler that actually controls the impact of the execution of large queries that has the impact on the other workload classes in a Database Management System. The approach involved divides a large query into an equivalent set of smaller queries and later schedules the execution of these smaller queries.

Keywords: Query Optimization, DBMS, Dissembler

I. INTRODUCTION

In past decades, we have experienced an information explosion. Now, to manage effectively such large volumes of information DBMSs have been widely used. The database management system (DBMS) has been very successful over the last half-century history. According to an IDC report made by C. Olofson [1] in 2006, the worldwide market for DBMS software was about $15 billion in 2005 alone with an estimated 10% growth rate per year. DBMSs and database applications have become a core component in most organizations' computing systems. These systems are becoming increasingly complex and the task to ensure acceptable performance for all applications is a challenge. In recent years, this complexity has approached a point where even DataBase Administrators (DBAs) and other highly skilled IT professionals are unable to comprehend all aspects of a DBMS's day-to-day performance [14] and manual management has become virtually impossible.

One solution to this growing complexity problem is IBM's Autonomic Computing initiative [14] [15]. An autonomic computing system is one that is self-managed in a way reminiscent of the human autonomic nervous system. To be more specific, an autonomic DBMS should be self-Configuring, self- Healing, self-Optimizing and self-Protecting (Self-CHOP). One of the efforts towards autonomic DBMS involves workload control, that is, controlling the type of queries and the intensity of different workloads presented to the DBMS to ensure the most efficient use of the system resources.[14]One challenge involved in the implementation of workload control is the handling of very large queries that are common in data warehousing and OLAP systems. These queries are crucial in answering critical business questions. They usually boast very complicated SQL and access a huge amount of data in a database. When executed in a DBMS, they tend to consume a large portion of the database resources, often for long periods of time. The existence of these queries can dramatically affect overall database performance and restrict other workloads requiring access to the DBMS. Our goal is to design a mechanism to dynamically control the execution of a large query so as to lessen its impact on competing workloads.

Due to the high degree of competition within a business environment more and more companies employ data warehouses and OLAP technologies to help the knowledge worker make better and fast decisions. When a large complex query is submitted to a high volume database for execution, it tends to consume many of the physical database resources like CPU. Also it may consume resources for long periods of time, thus impacting other possibly more important queries which may need resources to complete their work in a timely fashion.

A common approach to managing large queries within a DBMS is to classify queries as they enter the system and then to delay the submission of the large queries. This approach has 2 disadvantages. First, the large query is simply delay and no progress on that work is achieved. Second, in businesses with 24/7 availability there may exist no time at which the large query will not interfere with other work. A more flexible approach such as dynamically adjusting the DBMS resources of a running query, which allows a query to progress at a reduced rate, is preferable, especially in a differentiated service environment. Controlling the consumption of DBMS resources by a query (particularly a big query) is, however, not a trivial task. Ideally, low-level approaches, such as directly assigning CPU cycles or disk I/O bandwidths to a query based on its complexity and/or importance, are desirable. In practice, however, these approaches are problematic for two reasons. First, running a query against a DBMS involves many different and interrelated DBMS components. It is impossible to ensure that a query is treated equally (from the viewpoint of resource allocation) across all these

components. Secondly, it is difficult to determine the appropriate settings for the resource allocations for all the components.

II. ARCHITECTURE

The main purpose of this project was to implement a SQL disassembler that actually controls the impact of the execution of large queries that has the impact on the other workload classes in a Database Management System. Our approach involved here divides a large query into an equivalent set of smaller queries and later schedules the execution of these queries.

The goal of the work was to control the impact that the execution of large queries has on the performance of other workload classes. Our approach to decomposing a large query into a set of smaller queries is based on two observations. First, at any given time, a smaller query will likely hold fewer resources than a large query and so, interferes less with other parts of the workload. Second, running a large query as a series of smaller queries means that all resources are released between queries in the series and so are available to other parts of the workload. Figure 1 shows the Query Disassembler. Each large query is submitted to Query Disassembler before it is executed by the DBMS (*step 1*). Query Disassembler calls DB2's Explain utility to obtain a (cost-augmented) QEP for the submitted query (*steps 2 and 3*). The decomposition algorithm then divides the QEP into multiple segments, if possible, while keeping track of dependency relationships among the segments (*steps 4 and 4'*). The Segment Translation procedure transforms the resulting segments into executable SQL statements (*step 5)*, which are then scheduled for execution by the Schedule Generation procedure (*step 5').* The generated SQL statements are submitted to the DBMS for execution as per the schedule that is obtained in step 5' (*step 6*).

If the decomposition algorithm determines that it is impossible to break up the submitted large query, for example a single operator within the QEP for the large query covers most of the total cost, Query Disassembler notifies an Exception Management Module to handle this situation (*step 7*). The Exception Management Module is not currently implemented in our prototype but we envision that it could be implemented using an appropriate mechanism such as delaying the execution of the large query to an off-peak time in the system.

The work makes two main contributions. The first is an original method of breaking up a large query into smaller queries based on its access plan structure and the estimated query cost information. The second is an implementation of the Query disassembler. Query disassembler is used to break the queries, if necessary and manages the execution of the queries submitted to a DBMS. In our approach, we adopt a method similar to query decomposition techniques commonly used in distributed database management systems.But,unlike distributed database systems where queries are re-written to access data from multiple sources, our approach focuses on breaking up a large query into an equivalent set of smaller queries in a centralized database environment. Currently our algorithm supports select-only queries, which are typical in an OLAP system.

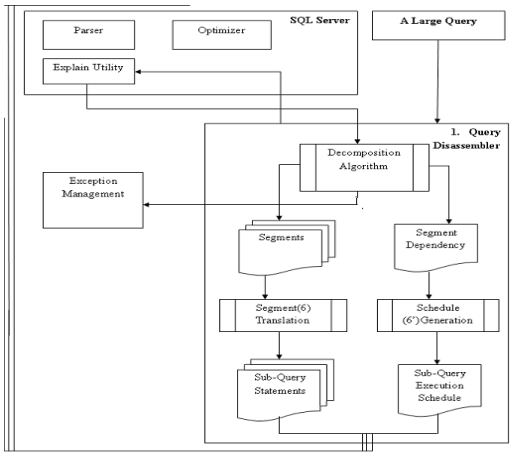

Figure 1: Query disassembler framework

Our work shows the feasibility and potential of the management of the execution of large queries in a database to increase workload performance. This suggests a number of interesting opportunities of future research. Currently, the small query set in our experiment workload contains just read-only queries. One important step of our approach involves translating a decomposed segment into an equivalent SQL statement. This step is highly vendor-specific and has some limitations that are inherent in our current approach due to the fact that our approach is implemented outside a database engine. The approach of controlling the execution of a large query in our work is to decompose the large query based on this QEP. This approach is static and cannot handle all types of large queries. It is very attractive to investigate a more flexible control mechanism, such as dynamically pausing or throttling query execution, so that the large queries that cannot be handled by our current algorithm, can be processed properly. Our current approach relies solely on the DB2 compiler to provide the necessary performance-related information, especially cost, to do the decomposition job. From this point of view, our approach is relatively independent from the configuration of the underlying computer system because the DBMS screens the system configuration change on the approach's behalf (assuming that the DBMS configuration parameters remain the same). However, it is very interesting to investigate how our approach can react to the change of the system configuration in a more active and reasonable way. For example, if more CPUs are added in the system, the decomposition algorithm could utilize that information to generate a more parallel segment schedule and therefore the performance of our approach could be enhanced by taking advantage of the parallelism introduced.

III. IMPLEMENTATION

The application developed has a main source code as program.cs where in, the entire process of disassembly is done. The code works as follows: First a user gives a nested query with proper parenthesis and related '=' operators. The application developed gets the query in the nested form and then it reads the queries in the normal form i.e. from outermost query to the innermost query. All these are stored

in an array in an individual way. Then at the final stage of this step we have the queries in an array.

During implementation, it was found that the query format must be maintained i.e. proper nesting of queries was needed like there must be some sort of separation between the queries. For an example if we have three nested queries we must make sure that segment dependency is maintained. So, we need to have a symbol that helps the application developed to identify the individual parts of the query. This is done with the help of the '=' operator.

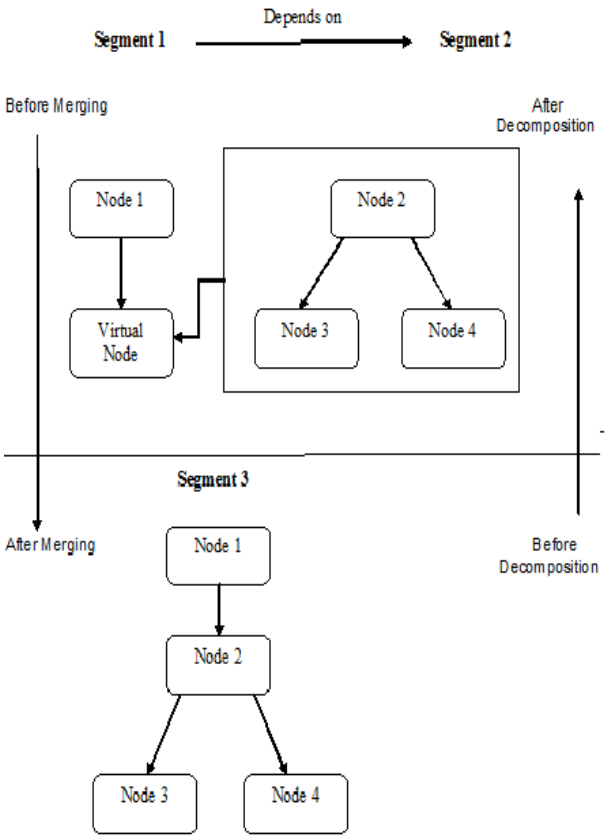

Figure 2: An example of merging/decomposing segment(s)

In the next step, we start reading the queries from the array and start creating a set of segments. Here we consider the dependency between the segments, i.e. if two segments are dependent then they must be scheduled properly, or else we get undesired results. So, creating a proper schedule is of high importance. After considering all these things we start reading the queries in the opposite direction from the array. Then we create a table that has two variants, that is, one variant has segments or individual queries that have no segment dependencies and the other set has segments that are dependent on each other and must be executed to get proper results. In the final step of the application, we take the individual queries as input and the queries are then submitted to the database that has been created.

IV. CONCLUSION

In this, we present an approach to managing the execution of large complex queries in a database and therefore controlling its impact on other smaller, possibly more important, queries. A decomposition algorithm that breaks up a large query into a set of equivalent smaller queries is discussed in detail. Our experiments show that concurrent execution of large resource-intensive queries can have significant impact on the performance of other workloads, especially as the points of contention between the workloads increase. We conclude that there is a need to be able to manage the execution of these large queries in order to control their impact.

The experiments show that our approach is viable, especially in cases when contention among the workloads is high, for example when a large query and other workloads run in the same database and share buffer pools. In other cases when the competition is low (by "low", we mean that the workloads do not share buffer pools, our approach does not work well. In these cases, the performance degradation that is caused by the overhead of our approach dominates and therefore makes our approach impracticable.

In this approach, the major overhead is primarily due to the costs involved in saving the intermediate results to connect the decomposed queries. Specifically, these costs include those related with creating, populating, accessing, and destroying the temporary tables that are necessary for accommodating the intermediate results. The overhead could be large in some cases, especially when a decomposition solution is reached by interrupting a pipelined operation. Currently, due to the fact that our approach is implemented outside of a database engine, we have no choice but to use an expensive way to store the intermediate results, which is to submit a "CREATE TABLE" SQL statement followed by an "INSERT" SQL statement and a "DROP TABLE" statement. For intermediate results from inside a database engine, we could probably design a cheaper and faster mechanism to save the intermediate results. A possible solution would be to save the ROWID and COLUMNID information of a table instead of storing its real record values. There are two main advantages of doing so. First, it can create a much smaller intermediate table because the ROWID and COLUMNID information of a table record is usually much smaller in size than the real record value. Second, it can also create a much faster intermediate table because the DBMS can utilize the ROWID and COLUMNID information to pinpoint the needed information directly rather than to go through an expensive and slow search process.

Another big improvement of saving the intermediate results from inside a database engine is that it would avoid the overhead that is caused by the DBMS following the standard parsing, compiling, and optimizing procedure to execute a submitted SQL statement.Here, this type of overhead is inevitable.

The experiment also show that our approach always causes performance degradation for the large query itself and sometimes the reduction can be significant, especially when the large query is decomposed in a way that a pipelined operation is interrupted. One reason for the degradation comes from the decomposition processes itself and another comes from creating, accessing, and deleting the intermediate tables. The first type of degradation is unavoidable in this approach. We could, however, shorten the overall delay by utilizing more advanced techniques of saving intermediate tables as discussed in previous paragraphs.

V. REFERENCES

[1] IDC Competitive Analysis: Worldwide RDBMS 2005 Vendor Shares: Preliminary Results for the Top 5 Vendors Show Continued Growth, http://www.oracle.com/corporate/analyst/reports/infrastructure/dbms/idc-1692.pdf.

[2] G. Luo, J. F. Naughton, C. J. Ellmann, M. W. Watzke. Toward a Progress Indicator for Database Queries, Proc. of the 2004 ACM SIGMOD Int. Conf. on Management of Data, Paris, France, June 2004, pp. 791 – 802.

[3] S. Chaudhuri, V. Narasayya, R. Ramamurthy. Estimating Progress of Execution for SQL Queries, Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data, Paris, France, June 2004, pp. 803 -814.

[4] N. Kabra, D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans, Proc. of the 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, USA, June 1998, pp. 106 -117.

[5] M. J. Carey, R. Jauhari, M. Linvy. Priority in DBMS resource scheduling, Proc. Of the 15th Int. Conf. on Very Large Data Bases, Amsterdam, The Netherlands, August 1989,pp. 397 – 410.

[6] B. Niu, P. Martin, W. Powley, R. Horman, P. Bird. Workload Adaptation in Autonomic DBMSs, Proc. of the 2006 Conf. of the Centre for Advanced Studies on Collaborative Research, Toronto, Canada, October 2006, Article No. 13.

[7] H. Boughton, P. Martin, W. Powley, and R. Horman. Workload Class Importance Policy in Autonomic Database Management Systems, Seventh IEEE Int. Workshop on Policies for Distributed Systems and Networks, London, Canada, June 2006, pp. 13-22.

[8] C. Ballinger, Introduction to Teradata Priority Scheduler, July 2006, http://www.teradata.com/library/pdf/eb3092.pdf.

[9] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, M. Urata, Answering Complex SQL Queries Using Automatic Summary Tables, Proc. of the 2000 ACM SIGMOD Int. Conf. on Management of Data, Dallas, USA, June 2000, pp. 105 -116.

[10] P. Martin, W. Powley, H. Y. Li, K. Romanufa, Managing Database Server Performance to Meet QoS Requirements in Electronic Systems, Int. Journal on Digital Libraries 3(4), pp. 316-324.

[11] R. Ramakrishnan, J. Gehrke, Database Management Systems (3rd Edition), McGraw-Hill Companies, Inc. 2003.

[12] S. Venkataraman, T. Zhang. Heterogeneous Database Query Optimization in DB2 Universal DataJoiner, Proc. of the 24th Int. Conf. on Very Large Data Bases, New York City, USA, August 1998, pp. 685 – 689.

[13] L. Liu, C. Pu, K. Richine, Distributed Query Scheduling Service: An Architecture and Its Implementation, International Journal of Cooperative Information Systems (IJCIS) 7(2&3), 1999, pp 123 – 166.

[14] Kephart J, Chess D. The vision of autonomic computing. IEEE Computing 2003;36:41–50.

[15] Shanta Rangaswamy, "Autonomic Computing-SCORE/EROCS" at International Association of Computer Science and Information Technology, ICSTE 2009, Chennai, Tamil Nadu.

Author Profile

Ms. Shanta Rangaswamy, Assistant Professor, Department of CSE, R.V. College of Engineering is pursuing her PhD from Kuvempu University. Her research areas of interest are Autonomic computing, Data mining, Performance Evaluation of systems, Cryptography and Steganography, System Modeling and Simulation.

Dr. Shobha G., Dean, PG Studies, (CSE & ISE) is associated with R.V.College of Engineering since 1995.She has received her Masters degree from BITS, Pilani and Ph.D (CSE) from Mangalore University. Her research areas of interest are Database Management Systems, Data mining, Data warehousing, and Information and Network Security.