CHAPTER 4

SQL QUERY DISSEMBLER- A SELF OPTIMIZING AUTONOMIC SYSTEM

Current database workloads often consist of a combination of short OnLine Transaction Processing (OLTP) queries and large complex queries such as those typical of 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 [73]. These resource intensive queries can dominate the database system resources and negatively impact the performance of smaller, possibly more important, queries.

In this chapter, 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 is presented. Also, the impact of interference of large queries during the execution of small queries is discussed in the further sections.

4.1 Introduction

Past few decades, have experienced an information explosion. Now, to manage effectively such large volumes of information databases have been widely used Database applications have become a core component in most organizations' computing systems. One solution to this growing complexity problem is IBM's Autonomic Computing initiative [3].

In recent years, this complexity has approached a point where even DBAs and other highly skilled IT professionals are unable to comprehend all aspects of a database's day-to-day performance [68] and manual management has become virtually impossible [73]. OLAP queries usually consist of arithmetic operations, multiple joins, nested sub-queries, and other system or user-defined functions and they typically operate on large data sets. In this case study, 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 is presented. Here, a SQL disassembler that actually controls the impact of the execution of large queries, which has the impact on the other workload classes in a Database Management System, is implemented [73].

One of the efforts towards autonomic database involves workload control, that is, controlling the type of queries and the intensity of different workloads presented to the database to ensure the most efficient use of the system resources[3]. 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 [73]. 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 on time.

One challenge involved in the implementation of workload control, is handling of very large queries. 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 database, 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 databases [73].

A common approach to managing large queries within a database is to classify queries as they enter the system [73]. This approach has two disadvantages. First, the large query will 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 database resources of a running query, which allows a query to progress at a reduced rate, is preferable, especially in a differentiated service environment [73].
Controlling the consumption of database resources by a query (particularly a big query) is, however, not an insignificant 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[73].

In practice, these approaches are problematic for two reasons. First, running a query against a database involves many different and interrelated database 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. When a complex query like this consisting of nested queries is submitted to the CPU, it tends to consume many of the physical database resources such as CPU, buffer pool or disk I/O and/or the logical resources such as system catalogs, locks, etc. The query may consume the resources for long periods of time, thus, impacting other, possibly more important, queries which may require these resources to complete their work on time.

4.2 Related concepts

Related concepts to this case study discussed further are Decomposition Algorithm, and system architecture.

4.2.1 Decomposition Algorithm

The goal of this part of the research work was to understand the impact that the execution of large queries has on the performance. This method of 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 chain of smaller queries means that all resources are released between queries in the series and so are available to other parts of the workload [73].

In this approach, a method similar to query decomposition techniques commonly used in distributed database management systems was adopted. Currently this
decomposition algorithm supports select-only queries, which are representative in an OnLine Analytical Processing (OLAP) system.

4.2.2 Assumptions and Dependencies

The various assumptions that are made before design of the applications and the databases are:

During the decomposition procedure, all other workloads that could be accessing the same tables used by the large query are read only queries. This means that the data processed by both the large query (before the decomposition) and its equivalent segment schedule (after the decomposition) are the same. Without this assumption, the result equivalency of this approach cannot be guaranteed.

The major assumption relating to the hardware is that the system has enough physical memory. The system is assumed to have sufficient memory to store a large database. The system's response time is assumed to be low enough to avoid long delays while retrieval of data from select queries [73].

4.3 System Architecture of query disassembler

Figure 4.3 shows the Query Disassembler [73]. Each large query is submitted to Query Disassembler before it is executed by the DBMS (step 1). Query Disassembler calls SQL Server’s Explain utility to obtain a Query Execution Plan (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) [73].

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 Department of CSE, RVCE
Management Module to handle this situation (step 7)[73]. The Exception Management Module is not currently implemented in this prototype.

The main contribution of this work is, identifying the execution time of the selected benchmark queries when run individually. Then, execution time of the same queries when run in parallel with other queries is identified. This is to identify the impact of one query over the other with respect to its execution time. Also, Query disassembler is used to break the queries, if necessary and manages the execution of the queries submitted to a DBMS. In this approach, a method similar to query decomposition techniques commonly used in distributed database management systems is adopted. But, unlike distributed database systems where queries are re-written to access data from multiple sources, this approach focuses on breaking up a large query into an equivalent set of smaller queries in a centralized database environment. Currently this algorithm supports select-only queries, which are typical in an OLAP system.
Figure 4.1: Query dissembler framework [73]
The 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. One important step of this approach involves translating a decomposed segment into an equivalent SQL statement. The approach of controlling the execution of a large query is to decompose the large query based on its QEP.

This approach is static and cannot handle all types of large queries. The current approach relies exclusively on the SQL compiler to provide the necessary performance-related information, especially cost, to do the decomposition job[73]. From this point of view, the approach is relatively independent from the configuration of the underlying computer system (assuming that the DBMS configuration parameters remain the same).

4.4 Experimental setup

The system used was a Pentium 3 processor with 500 MHz speed. The language used to create the application was Java. The backend, to store the details of the database, was created using Microsoft SQL server. TPC-H benchmark queries were used for experimentation. Scaling factor of the database was chosen as 1, indicating the database size as approximately 1GB. Total number of tables in the database is 8. TPC-H query number 10, 11, 16 and 18 were the sample queries considered for experimentation. These queries need to interact with maximum number of tables in the database. TPC-H benchmark, however, models the analysis end of the business environment where trends are computed and refined data are produced to support the making of sound business decisions. In OLTP benchmarks the raw data flow into the OLTP database from various sources where it is maintained for some period of time [73]. In TPC-H, periodic refresh functions are performed against a DSS database whose content is queried on behalf of or by various decision makers. Complete schema and table details along with queries used for experimentation are documented in Appendix-B at the end of the thesis.
4.5 Result and Interpretation

Table 4.1: Query number and its execution time.

<table>
<thead>
<tr>
<th>Query</th>
<th>Execution Time (milliseconds)</th>
<th>Execution Time (in milliseconds for second query in parallel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only Q10</td>
<td>1.0000E+03</td>
<td>--</td>
</tr>
<tr>
<td>Only Q11</td>
<td>9.8400E+02</td>
<td>--</td>
</tr>
<tr>
<td>Only Q16</td>
<td>2.0630E+03</td>
<td>--</td>
</tr>
<tr>
<td>Only Q18</td>
<td>9.3700E+02</td>
<td>--</td>
</tr>
<tr>
<td>Q10 and Q11</td>
<td>1.38676283E+12</td>
<td>1.38676831E+11</td>
</tr>
<tr>
<td>Q10 and Q16</td>
<td>1.38676330E+13</td>
<td>1.38676330E+13</td>
</tr>
<tr>
<td>Q10 and Q18</td>
<td>1.38676315E+12</td>
<td>1.38676315E+12</td>
</tr>
<tr>
<td>Q11 and Q10</td>
<td>1.38676330E+12</td>
<td>1.38676330E+12</td>
</tr>
<tr>
<td>Q11 and Q16</td>
<td>1.38676336E+12</td>
<td>1.38676336E+12</td>
</tr>
<tr>
<td>Q11 and Q18</td>
<td>1.38676340E+12</td>
<td>1.38676340E+12</td>
</tr>
<tr>
<td>Q16 and Q10</td>
<td>1.38676352E+12</td>
<td>1.38676352E+12</td>
</tr>
<tr>
<td>Q16 and Q11</td>
<td>1.38676358E+12</td>
<td>1.38676358E+12</td>
</tr>
<tr>
<td>Q16 and Q18</td>
<td>1.38676362E+12</td>
<td>1.38676362E+12</td>
</tr>
<tr>
<td>Q18 and Q10</td>
<td>1.38676373E+12</td>
<td>1.38676373E+12</td>
</tr>
<tr>
<td>Q18 and Q11</td>
<td>1.38676377E+12</td>
<td>1.38676377E+12</td>
</tr>
<tr>
<td>Q18 and Q16</td>
<td>1.38676381E+12</td>
<td>1.38676381E+12</td>
</tr>
</tbody>
</table>

Figure 4.2: Graph representing queries versus execution time
TPC-H benchmark queries were used for experimentation. The queries chosen were Q10, Q11, Q16 and Q18. The execution time were recorded in milliseconds. It was found that the execution time of the queries take longer execution time when run in parallel rather than individually. Though the difference in execution time in parallel and individual is minimal, it can be interpreted that, as the workload on the CPU increases, the response time also increases considerably. This is due to the fact that the resources need to be shared among the queries or input.

4.6 Snapshots

![Figure 4.3: Screen shot showing execution time during parallel execution of queries.](image)

Figure 4.3 is an indicative screen shot taken during experimentation. Figure shows main and child thread execution in parallel with its execution start time and end time. The difference between start and end time of the execution of the query is the total time taken for execution.
Figure 4.4: GUI showing the execution time of queries in parallel

Figure 4.4 shows the Graphical User Interface between the user at the application layer and database at the backend. In this manner, selected standard TPC-H queries were executed in isolation and in parallel with other queries. As discussed before Q10, Q11, Q16 and Q18 benchmark queries were chosen for experimentation. These queries need to interact with various tables in the database for execution.

4.7 Summary

In this experiment, 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 are presented. Experiments shows that concurrent execution of large resource-intensive queries can have significant impact on the performance of other smaller workloads. Therefore one can 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 the approach is feasible, 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.
As discussed in chapter 3, query execution optimization in autonomic DBMS is one of the major components of ADBMS. Current optimizers already available uses best execution plan against any SQL query or uses existing materialized views automatically. Also, current optimizers use cost based and heuristic based techniques for optimization. There is a facility for the administrator to set the optimizer goals, which may be either throughput or response time. This experiment investigates the impact of running the large queries independently and in parallel with others. It was found that the execution time of the query increases when run in parallel than independently. It was found that Q10 and Q16 when run parallelly took comparatively longer time than other queries.