SQL Query Disassembler

An Approach to Managing the Execution of Large SQL Queries

by

Yabin Meng

A thesis submitted to the

School of Computing

in conformity with the requirements for the

degree of Master of Science

Queen's University

Kingston, Ontario, Canada

September, 2007

Copyright © Yabin Meng, 2007

Abstract

Current database workloads often consist of a mixture 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. These resource intensive queries can monopolize the database system resources and negatively impact the performance of smaller, possibly more important, queries.

In this thesis, 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. We describe a prototype implementation of our approach for IBM DB2™ and present a set of experiments to evaluate the effectiveness of the approach.

Acknowledgments

I would like to extend my sincerest gratitude to my supervisor, Professor Patrick Martin, for his great guidance and help over the years that I have pursued my education and research at Queen's University.

I would also like to thank Wendy Powley for her support. She has always been a wonderful source of advice and suggestions. Without her great work, I could not finish my thesis so smoothly.

I would like to give my gratitude to the School of Computing at Queen's University for their support. I would also like to acknowledge IBM Canada Ltd., NSERC, and CITO for the gracious financial support they have provided.

I would like to thank my lab mates and fellow students for their encouragement, support, and kindness.

Finally I would like to thank my family for their love and understanding. Their support is always the key in my journey.

Table of Contents

List of Tables

List of Figures

List of Equations

Chapter 1

Introduction

1.1 Motivation

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 of management to ensure acceptable performance for all applications is very difficult. 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 [29] and manual management has become virtually impossible.

One solution to the growing complexity problem is IBM's Autonomic Computing initiative [29] [31]. 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-tuning, self-protecting and self-healing. 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. One challenge involved in the implementation of workload control is the handling of very large queries that are common in data warehousing and online analytical processing (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.

1.2 Problem

In the past several decades, we have experienced an information explosion. According to a study conducted by Lyman and Varian [2], there were 5 exabytes (10^{18} bytes) of "static" information (in the form of paper, film, magnetic and optical storage medias) and another 18 exabytes of "dynamic" information flowing through electronic channels (TV, radio, internet, etc) in the year 2002, with the growth factor estimated to be about 30% per year. Ninety-two percent of the static information is stored on magnetic media, mostly on hard disks. In order to effectively manage such large volumes of information, DBMSs have been widely used, thus leading to an astonishing boost in the volume of data that a single database must manage. According to the 2005 report of the "TopTen Program" by the Winter Corporation [3], the world's largest data warehouse in 2005 contained 100,386 GB of data, and the largest scientific database was 222,835 GB.

Due to the high degree of competition within the business environment, more and more companies are employing data warehousing and OLAP technologies to help the *knowledge worker"* (executive, manager, analyst, etc.) [4] make better and faster

decisions. Decision-support queries usually boast very complex forms, including multiple joins, nested sub-queries, multi-dimension aggregations, arithmetic operations, and system- or user- defined functions. Moreover, they also operate over huge amounts of data.

```
1   select  s_name,
2           count(*) as numwait
3   from    supplier,
4           lineitem l1,
5           orders,
6           nation
7   where   s_suppkey = l1.l_suppkey
8           and o_orderkey = l1.l_orderkey
9           and o_orderstatus = 'F'
10          and l1.l_receiptdate > l1.l_commitdate
11          and exists (
12                  select  *
13                  from    lineitem l2
14                  where   l2.l_orderkey = l1.l_orderkey
15                          and l2.l_suppkey <> l1.l_suppkey
16          )
17          and not exists (
18                  select  *
20                  from    lineitem l3
21                  where   l3.l_orderkey = l1.l_orderkey
22                          and l3.l_suppkey <> l1.l_suppkey
23                          and l3.l_receiptdate > l3.l_commitdate
24          )
25          and s_nationkey = n_nationkey
26          and n_name = '[NATION]'
27  group by
28          s_name
29  order by
30          numwait desc,
31          s_name;
```

Figure 1: TPC-H Q21, an example of decision-support queries

Figure 1 shows one query, Query 21, of the TPC-H benchmark [5] which is a decision support benchmark developed by Transaction Processing Performance Council. Query 21 is one of the suite of business oriented ad-hoc queries specified in the benchmark and is used to "identify suppliers, for a given nation, whose product was part of a multi-supplier order (with current status of 'F') where they were the only supplier

who failed to meet the committed delivery date" [5]. As we can see from Figure 1, this query has a complex SQL statement including multiple joins among four different tables, three of which are relatively large (lineitem, orders, and suppliers). It also includes nested sub-queries and aggregation. Query 21 is typical of decision support queries.

When a query like TPC-H Q21 is submitted to a high volume database for execution, 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 in a timely fashion.

The situation is made worse by the emerging trend of server consolidation and service-oriented architecture (SOA). Business entities use server consolidation as an important means of cutting unnecessary costs and maximizing return on investment by shifting the functionalities of several, under-utilized servers onto one powerful server. This trend on database servers means that one single database server must support very different workloads simultaneously that were traditionally handled by different database servers. One direct consequence of this trend is that the DBMS must now be able to handle multiple workloads with diverse characteristics, dynamic resource demands, and competing performance objectives.

Service-oriented management (SOM) is the operational management of service delivery within a SOA [6]. The main purpose of SOM is to guarantee a differentiated service delivery based on Service Level Objectives (SLOs) and Service Level Agreements (SLAs). Within SOM, a system's behavior is driven by business objectives.

In order to realize this goal, a DBMS must have the ability to dynamically control its resource allocations for the queries that are submitted to it.

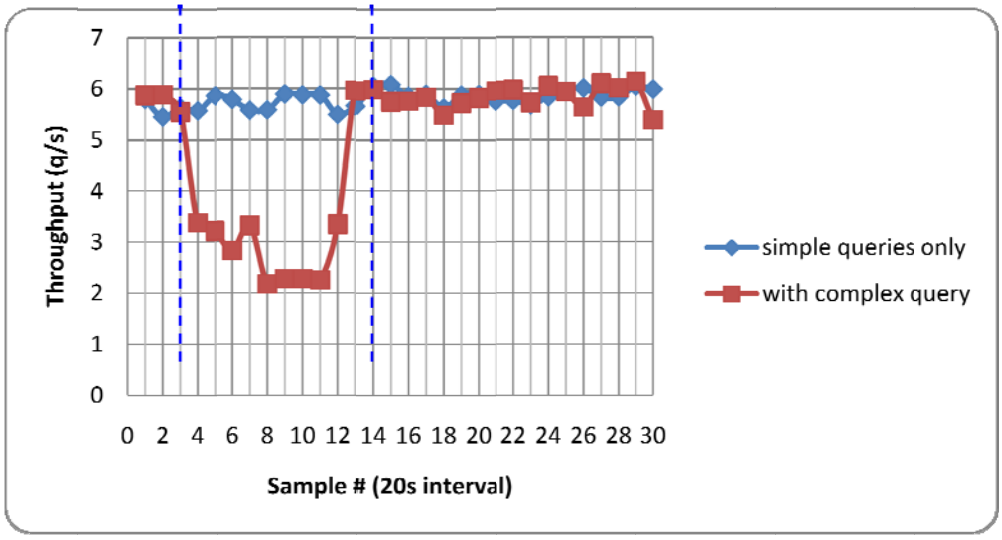

Figure 2: Performance degradation due to running a complex query

Figure 2 shows an example of the performance degradation experienced by a series of small read-only OLTP-like queries due to the interference of a large query (TPC-H Q21) in a DBMS. The throughput of the small queries in the system are monitored for 600 seconds and sampled every 20 seconds, with and without the large query running simultaneously. Chapter 5 provides an explanation of the experimental configuration. It can be seen from Figure 2 that when the large query is running (starting at the 3rd sampling point and ending at the 13th sampling point), the average throughput of the small queries drops dramatically to less than 50% of the original throughput.

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 delayed 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.

1.3 Research Statement

The goal of this research is to investigate a high-level approach to controlling the impact that the execution of large queries has on the performance of other workload classes in a DBMS. Our approach divides a large query into an equivalent set of smaller queries and then schedules the execution of these smaller queries.

Our work makes two main contributions. The first contribution 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 contribution is a prototype implementation called Query Disassembler. Query Disassembler uses the proposed

algorithm to break up queries, if necessary, and manages the execution of the queries submitted to a DBMS.

The remainder of the thesis is organized as follows. Chapter 2 describes research background and related work. The core part of our work, the decomposition algorithm, is discussed in detail in Chapter 3. In Chapter 4 we introduce Query Disassembler, which is a prototype implementation of our approach for IBM DB2™. We present a set of experiments to evaluate our approach in Chapter 5. We conclude the thesis in Chapter 6.

Chapter 2

Background and Related Work

Very complex queries have gained plenty of research attention in online analytical processing (OLAP) and data warehousing systems due to the emphasis on increasing query throughput and decreasing response time in these systems [4]. On one hand, much of the research focuses on minimizing the query completion time, or providing feedback more quickly for the large query itself. In Section 2.1 we present some research efforts in this area. On the other hand, how to reallocate DBMS resources to meet different quality of service (QoS) requirements for a complex workload, ideally in an autonomic way based on some pre-defined business objectives and policies, is attracting more and more attention. Section 2.2 describes research efforts in this area. In both sections, we outline how our work relates to these previous research efforts. In Section 2.3, we will briefly present the general query decomposition technique that is commonly used in the distributed database systems and show how our decomposition algorithm is different from that technique.

2.1 Improving the Performance of Large Queries

In recent years, the control of running large queries such as those typical of OLAP and data warehousing in a DBMS has become more interactive. The traditional optimization techniques which are common in current database systems often fail to meet this new requirement because of their inherent "batch mode" characteristics. This means that once a large query is submitted to a DBMS, users have no control over its execution and they

often wait for a long period of time without any feedback until a precise answer is returned.

In order to overcome this problem, various techniques of providing more timely feedback are proposed. Luo et al. [7] and Chaudhuri et al. [8] investigate the possibility of providing an online progress indicator (percentage of the task that has completed) for long-running large queries. In both approaches, the progress estimator works on the query execution plan (QEP) that is chosen by the query optimizer for a given query. They differ in their choice of the basic unit of the query execution work. Luo et al. use one page of bytes that has been processed along the QEP as one basic unit. Chaudhuri et al. choose one "GetNext ()" call by the operators in the QEP as one basic unit. These techniques do not shorten the execution time of the large queries themselves, but they can provide users continuous feedback on how much of the work has completed.

Haas et al. [9] propose a join algorithm, called Ripple Joins, for online multi-table aggregation queries and Hellerstein et al. [10] investigate how to apply this new algorithm in a DBMS to generate results more quickly. The underlying reasoning of their work comes from the cognition that since large aggregation queries tend to give a general picture of the data set, it is more appealing to provide users estimated online aggregation results with a proximity confidence interval to the final result. Their algorithm adopts the statistical method of sampling from base relations in order to generate answers more quickly. A major advantage to this approach is that it allows users the ability to make a tradeoff between the estimation precision and the updating rate. Their approaches do not necessarily speed up the query execution itself. There may be some improvement, however, by the replacement of a blocking join algorithm like hash join with the non-blocking ripple join algorithm. Nevertheless, this is not the main objective of their work.

If appropriately used, materialized views (MVs) can provide performance improvement in query processing time since a (large) portion of the final result is pre-computed. The difficulty of using this approach, however, lies in how and when to exploit the MVs. Goldstein et al. [11] present a fast and scalable view-matching algorithm for determining whether part or all of a query can be computed from materialized views. They also demonstrate an index structure, called a filter tree, to help speed up the search for an appropriate view among the views maintained by a DBMS. This approach is very attractive in a situation where system workloads are stable because in these systems we are able to create useful MVs, that is MVs with repeatable usage among different queries in advance based on the understanding of the workload characteristics. In contrast, when the system's workloads are diverse and ad hoc, it is impossible to do so, and therefore this approach is not effective.

Kabra et al. [12] examine the possibility of dynamic memory reallocation for physical operators within a QEP based on improved estimates of statistics. Most modern algorithms for basic relational operators use DBMS statistics to estimate their memory requirement which, in turn, determines the algorithms' performance. In their work, Kabar et al. propose a run-time statistics collection technique which can be used to help improve the estimation of the database statistics. Their work involves the modification of a QEP by inserting "Statistics Collector" operators at several points in the QEP. The collected statistics can be used to obtain more accurate estimates for the remainder of the query or, if necessary, to create a better QEP for the query.

All the research efforts presented above mainly focus on increasing the performance (or perceived performance) of a large query itself. They do not directly address the problem of controlling the execution of large queries. However, the ideas presented

provide useful insights into our own research. First, our approach involves the decomposition of a large query into an equivalent set of smaller queries. The decomposition algorithm works on the QEP of a query and tries to identify pipelined parts within a QEP, just as the techniques used by Luo et al. [7] and Chaudhuri et al. [8]. Second, the ultimate goal of our work is not only to improve the performance of other queries in the presence of large queries, but we would like to minimize the impact of our approach on the large queries as well. The techniques of providing answers more quickly or speeding up the large query's execution as presented by Haas et al. [9], Hellerstein et al. [10], Goldstein et al. [11], and Kabra et al. [12] could therefore be helpful in satisfying this purpose.

2.2 DBMS Resource Allocation

The problem of resource allocation within a DBMS is very complicated. The reason is rooted in the inherent heterogeneity and multiplicity of the DBMS resources. A DBMS contains not only the common physical resources, like CPU, memory, and disk I/O, but it also contains many logical resources such as system catalogs, locks, etc. These resources, either physical or logical, are often inter-related and interact with each other, thus further complicating the resource allocation problem.

Traditionally, much of the work that has been done with regards to DBMS resource allocation has been implemented through static tuning of database parameters in order to optimize system wide performance. In recent years, with the emerging trend of server consolidation, the increased complexity of a DBMS and the ongoing emphasis on service-

oriented management, a more dynamic and goal-oriented approach is attracting more research interest.

Carey et al. [13] investigate the architectural consequences of adding priority to a DBMS. They develop a specific priority-based algorithm for managing the key physical DBMS resources, especially the disk(s) and the buffer pool(s). Their simulation results showed that "the objective of priority scheduling cannot be met by a single priority-based scheduler", which means that no matter whether the bottle neck of a DBMS is the CPU or the disk, it is always essential to also use a priority-based replacement algorithm on the buffer pool.

Brown et al. [14] investigate goal-oriented resource management in a DBMS. In their work, they propose a feed-back based algorithm, called M&M, which adjusts DBMS multi-programming levels (MPL) and memory allocations simultaneously, in an automatic way, to achieve a set of per-class response time goals for a multi-class complex workload while leaving the largest possible left-over resources for the non-goal, or best-effort classes. In their work, they adopt a per-class solution strategy, which means that, in a given timeframe, the algorithm is only activated for one class and takes action for that specific class in isolation. They use additional heuristics to compensate for the insensitivity of their approach to class inter-dependence.

Niu et al. [15] aim to optimize overall database resource usage by controlling the workloads presented to it. In their work, a workload detection process is used to monitor the characterization of the current workloads and to predict the future trends of the workloads. Based on the classification of the workloads made by the workload detection process, the workload control process is invoked to automatically adjust the MPLs assigned for each class such that the SLO for each class is satisfied. Unlike the average

response time goal used by Brown et al. [14], Niu et al. use Query Velocity, as the goal, or SLO, for each workload class.

Most of the goal-oriented, multiclass-workload research work expresses workload goals and the tuning policies in IT friendly ways such as response time or throughput. Although this allows the computer system to understand and control the workloads' behavior easily, it makes it more difficult for the decision makers. Boughton et al. [16] investigate the possibility of automatically translating high-level business polices into low-level system tuning polices using an economic model. The effectiveness of their economic model is tested in the context of the buffer pool sizing problem in a DBMS.

Currently, commercial database systems also provide a certain level support for dynamic DBMS resource management. IBM DB2 Query Patroller [17] is a query management system that aims to boost overall database system resource utilization. Using Query Patroller, queries submitted to a DB2 database are grouped into different categories based on their size and the submitters' identities. Each query class can have its own class-level policy (e.g. maximum number of queries allowed for each class). The system in general can have a high-level system policy affecting all query classes (e.g. maximum workload cost value for the system).

Teradata's Priority Scheduler (Teradata PS) [18] introduces the concepts of "user group", "performance group" and "allocation group". A Teradata DBMS uses "user groups" to classify the queries that are submitted by database users. It then establishes a user-to-priority connection by setting a valid "performance group" name in the user's record. The performance group is a priority scheduler component that associates users to "allocation groups" which, as well as their predefined relative weights, determine the real

physical database resource usage such as the frequency of accessing CPU and the relative position in the I/O queue.

Although goal-oriented, multi-class resource allocation is becoming a trend for DBMS resource management, most research work and current commercial products treat extremely large queries in a static and somewhat "crude" way. A popular approach is to adopt some kind of admission control mechanism to preclude large queries out of the system in advance and delay their execution until a system off-peak time. Our research investigates an approach such that not only do other queries in the system have more reasonable resource allocation, but the large queries themselves can be controlled in a more flexible and manageable way. The "utility throttling" technique used by Parekh et al. [19] for controlling the performance impact that a database administration utility has on the system has similar goals to our work but adopts a different approach. Unlike our approach which is implemented outside of the database engine and achieves the dynamic control over a large query by breaking it into pieces, their approach is implemented within the database engine and dynamically forces a resource-demanding utility to go to sleep for a while if the predefined workload objectives are not satisfied.

2.3 Query Decomposition in Distributed Database System

In a typical distributed database system, the data that needs to be accessed by a SQL query usually resides on several inter-connected remote sites. In order to process these remotely distributed data effectively and efficiently, new query processing techniques are required. D. Kossmann [31] presents a high level overview of the state of the art of query processing techniques for distributed database and information systems. Architecturally,

most of the distributed DBMSs support a basic processing model of “moving query to data”, which means that an “administrative” site (the site that receives the query) has to break down the query somehow such that each sub-query, after being sent to a remote site, only accesses the data that resides on that site. The purpose of decomposition here is mainly to reduce the communication cost that is usually the dominant factor of the query processing in a distributed environment. The way of the query decomposition in a distributed DBMS environment highly depends on how the underlying data is partitioned across the different sites. Our current approach of query decomposition, on the other hand, only focuses on a centralized environment right now and the decomposition method depends solely on the structure and the operator cost distribution of a query’s execution plan as suggested by the query compiler. The purpose of our method is also different from that used in a distributed system. The intention of our approach is to control the resource consumption by a large query so other queries, possibly more important, can get more DBMS resources for their own execution. The goal of decomposition in a distributed database, however, is to reduce the query processing cost and/or response time for the query in consideration.

Chapter 3

Decomposition Algorithm

The goal of our work is 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. In our approach, we adopt a method similar to query decomposition techniques commonly used in distributed database management systems. 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.

3.1 Query Execution Plan

The output of a query optimizer for a declarative query statement is called a Query Execution Plan (QEP). The structure of a QEP determines the order of operations for query execution. The QEP is typically represented using a tree structure where each node represents a physical database operator (e.g. nested loop join, table scan etc). Multiple plans may exist for the same query and it is a query optimizer's top priority to choose an optimal plan. To supplement the QEP, most query optimizers produce performance-

related information such as cost information, predicates, selectivity estimates for each predicate and statistics for all objects referenced in the query statement.

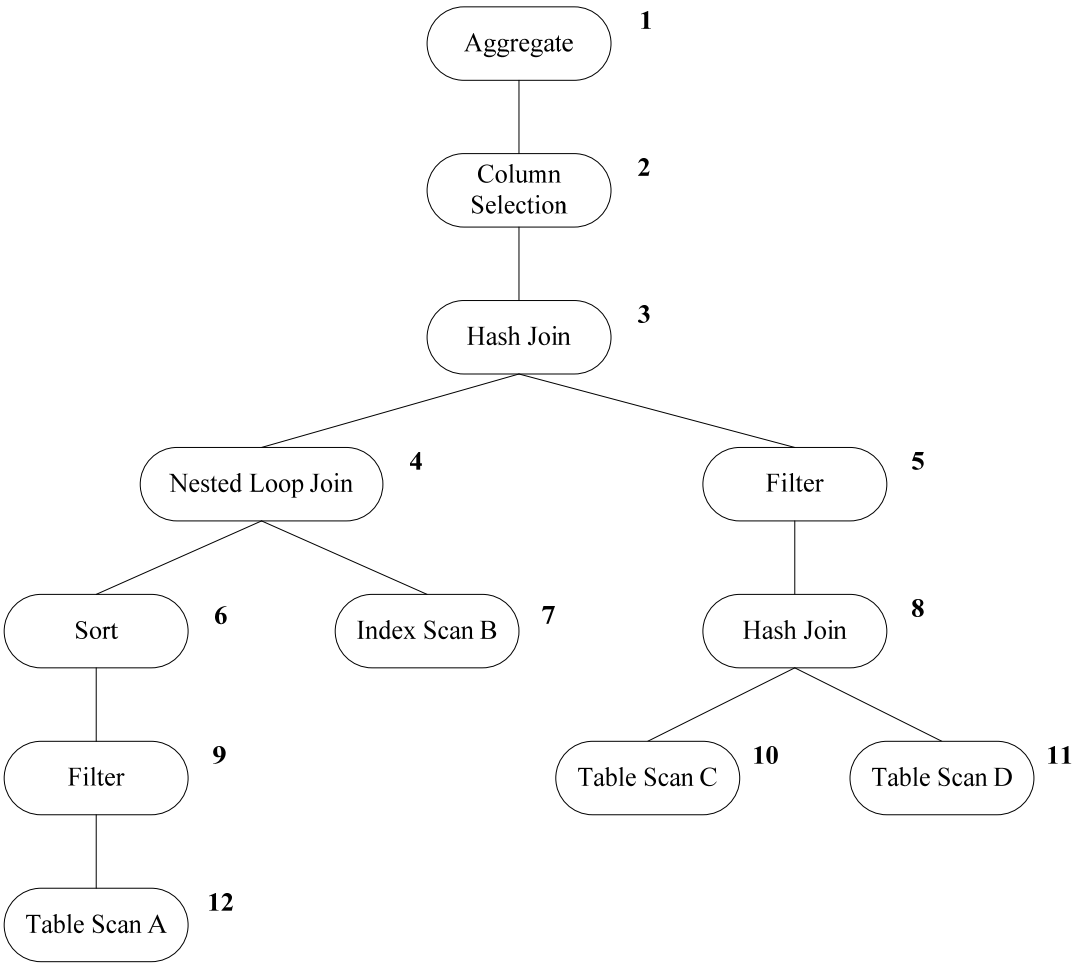

Figure 3: A sample QEP

Figure 3 shows an example QEP that we use for illustrative purposes throughout this chapter. In this QEP, data from four different database tables (Tables A, B, C, and D) are retrieved, filtered, joined, and then aggregated to create the desired final results (See Appendix B for further explanation of the common physical operators in a QEP). We note that the plan structure shown in Figure 3 is only a conceptual structure and not an actual plan from a query optimizer. It is used for illustrative purposes only.

3.2 Virtual Node, Segment, and CB-Segment

The operators in a QEP can be classified as either *blocking* operators or *pipelining* operators. An operator is *blocking* if it does not produce any output until it has consumed at least one of its inputs completely. *Pipelining* operators produce outputs immediately and continuously until all inputs have been processed. The hash join operator (Node 8) in Figure 3, for example, is a blocking operator and the filter operators (Nodes 5 and 9) are pipelining operators.

We classify the common physical operators in a QEP as follows:

- Table Scan, Index Scan, Filter, Column Selection and Nested Loop Join are pipelining operators
- Distinct (or Unique) is a blocking operator
- Sort, Hash Join and Merge-Sort Join are blocking operators
- Union, Intersect and Except are blocking operators
- Aggregation operators are treated as blocking operators although in reality they may be pipelining operators depending on the type of aggregate function or whether the group-by operator is used.

A **Virtual Node** is a conceptual (non-physical) node used as a connector between two segments (defined below) in our algorithm. It establishes a dependency relationship (see Section 3.3) between two segments and is implemented as a Table Scan node that provides access to a temporary database table. A virtual node is a pipelining operator.

A **Segment** is a sub-tree of a QEP such that: (1) the root node of a segment must be a blocking node or the return node of the original QEP, (2) a segment can have at most

one blocking node, and (3) all non-root nodes within a segment are pipelining nodes, including virtual nodes. The definition of segment guarantees that any identified segment is a maximum unit that can be executed in a pipelined fashion.

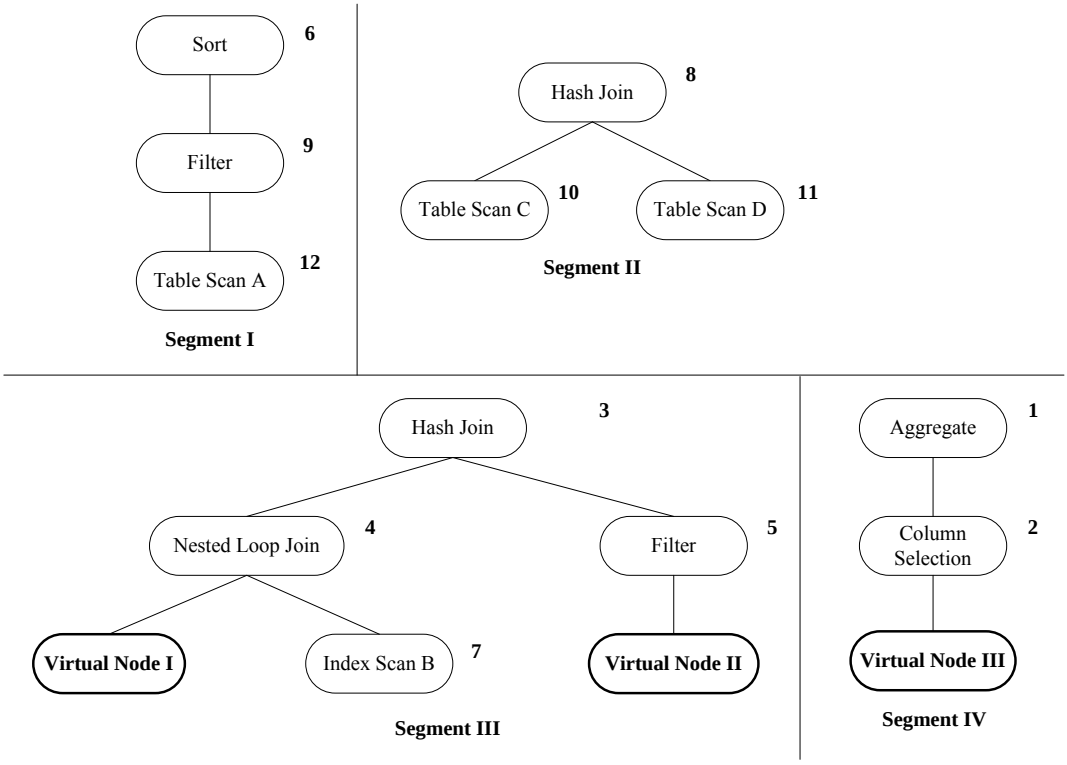

Figure 4: Segments and virtual nodes for QEP in Figure 3

Figure 4 shows the segments and virtual nodes for the QEP in Figure 3. In this Figure, Virtual Nodes I, II, and III represent Segments I, II, and III respectively. Virtual Node I creates a dependency relationship between Segment I and Segment III. Similarly, dependency relationships are also established between Segments II and III by Virtual Node II and between Segments III and IV by Virtual Node III (see Section 3.3 and Section 3.4 for more details on segment dependency as well as the detailed segment identification process).

A **Cost-Based Segment (CB-Segment)** is any valid sub-tree of a QEP. Unlike a Segment, there are no constraints placed on a CB-Segment. A CB-Segment is augmented with cost information such as the total cost of the CB-Segment, or the cost percentage of the CB-Segment over the total QEP cost. Cost is expressed in units adopted by a particular DBMS. In DB2, for example, a unit called *timeron* is used (Appendix C). In our work, a CB-Segment is created through merging or decomposing Segments (and/or CB-Segments). In the rest of the thesis, unless explicitly stated, we use the term ***segment*** to refer to both Segment and CB-Segment.

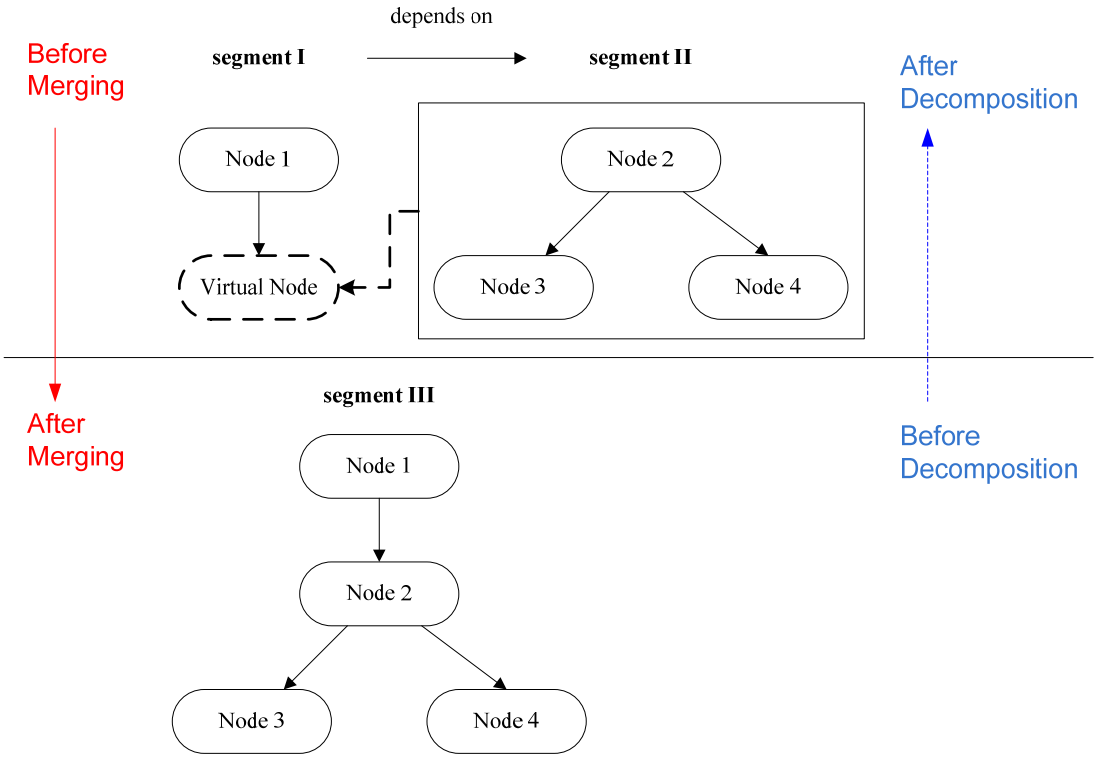

Figure 5: An example of merging/decomposing segment(s)

Figure 5 shows an example of how the merging/decomposing procedure works. In this Figure, segment I and segment II can be merged together to create segment III or

segment III can be decomposed to create segment I and segment II. The merging process requires that segment I depends on segment II, meaning that segment I has a virtual leaf node that represents segment II (see Section 3.3 for segment dependency). When segment I and segment II are merged together, the virtual node in segment I is removed and segment II is added as a child sub-tree of the virtual node's parent node (Node 1 in this case) at the virtual node's original position in segment I. The newly created tree becomes the merged segment III. If multiple virtual nodes exist in a segment that needs to be merged, then each virtual node is replaced by the segment represented by the virtual node. This process is explained in Figure 5 in the direction from top to bottom (marked by the solid line on the left). Similarly but for the reverse direction (from bottom to top and marked by the dotted line on the right), when segment III needs to be decomposed, the whole sub-tree in segment III that is represented by segment II needs to be replaced by a virtual node and thus creating segment I. During this procedure, a segment dependency relationship between segment I and segment II is established (see Section 3.3 for detail).

3.3 Segment Dependency and Schedule

According to the definitions in Section 3.2, a segment may contain virtual nodes as well as other regular operation nodes. In our work, each virtual node within a segment is used to represent another segment, which means that the outside segment depends on the segment represented by the virtual node. If a segment does not include a virtual node, then it is independent. In Figure 4, for example, segments I and II are independent segments because they contain no virtual nodes. Segment III includes two virtual nodes, virtual nodes I and II, which represent segments I and II, respectively. Therefore, segment

III depends on both segments I and II. Similarly, segment IV depends on segment III because it contains virtual node III, which represents Segment III. The dependency relationships among the segments in Figure 4 are shown in Figure 6.

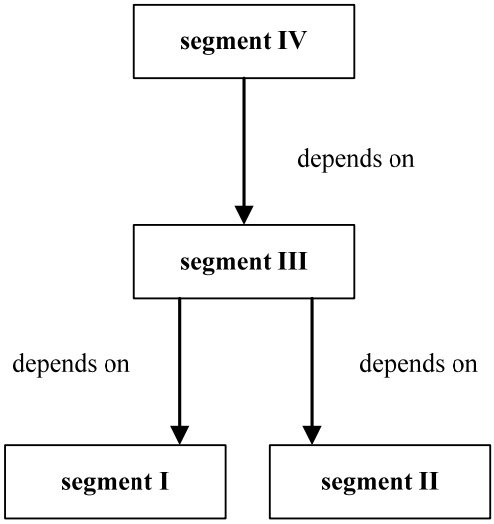

Figure 6: Dependency relationships for segments in Figure 4

In a QEP, if segment A depends on segment B, then the subpart of the QEP that is represented by segment B has to be executed before the subpart represented by segment A because it needs the output of segment B in order to produce its own results. If segments are independent of each other in a QEP, then they can be executed in any order or in parallel.

In our work, we define the execution order of all the segments in a QEP as the ***Segment Schedule*** for the QEP. Based on the segment dependency relationships in Figure 6, the Segment Schedule for the QEP in Figure 3 can be one of the three cases shown in Figure 7 (assuming parallelism is possible).

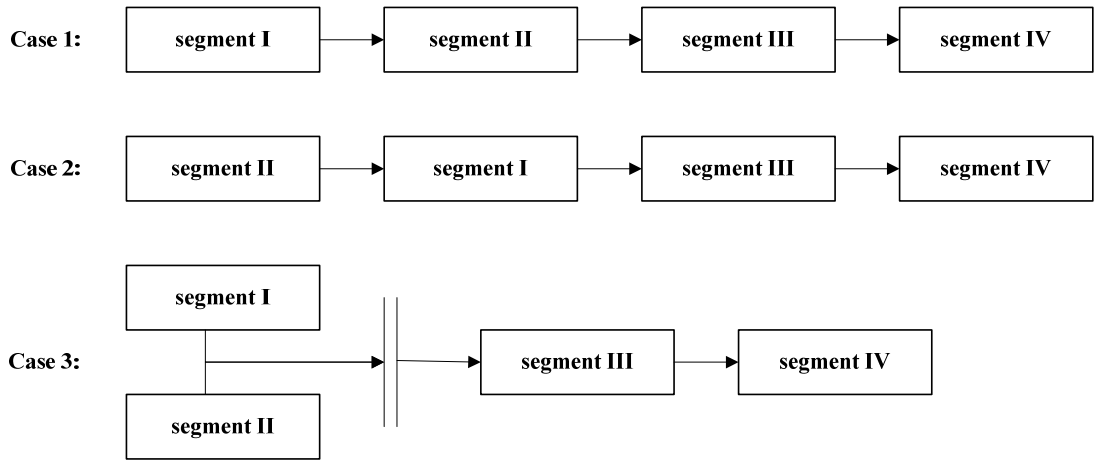

Figure 7: Segment schedule for QEP in Figure 3

3.4 Decomposition Algorithm

Our decomposition algorithm aims to find a cost-efficient strategy to decompose a QEP. The algorithm takes two passes. The first pass identifies all possible segments in a QEP by exploring its tree structure. During this pass, a bottom-up scan of the QEP is used to search for blocking nodes. Each blocking node forms the root of a segment and its lower-level descendents form the subtree. Once a segment is discovered, the sub-tree that it represents in the original QEP is replaced by a virtual node, thus creating a new tree with a virtual node as one of its leaves. This search-and-replace procedure continues until all nodes in the original QEP are processed and all segments are identified. At the same time, by means of the virtual nodes that are created during this pass, the dependency relationships among the segments are also defined. A segment having a virtual node as a leaf node depends on the segment represented by the virtual node.

A pseudo code description of the first pass of the decomposition algorithm is shown in Figure 8. It includes an iterative procedure called “FindSegments” which is applied to a

query's QEP. When the iterative process of this procedure is done, all segments in the QEP as well as their dependency relationships are identified and stored in two global sets, ***GSegSet*** and ***GSegRelSet***.

```
1   FindSegments (QEP) {
2       subTreeSet ← create a empty tree set;
3       FOR each leaf node (curLeaf) of QEP
4           curPath ← identifies the tree path that curLeaf belongs to in QEP;
5           cbaNode ← find its closet blocking ancestor node or the return node along curPath;
6           subTree ← create a sub-tree that is rooted at cbaNode and includes all nodes in the
                curPath from cbaNode to curLeaf;
7           add subTree into subTreeSet;
8       ENDFOR;
9       REPEAT
10          curSubTree ← get next sub-tree from subTreeSet;
11          newSeg ← NULL;
12          IF the root node of curSubTree has only one input in the original QEP THEN
13              newSeg ← create a new segment that has the same structure as curSubTree;
14          ELSE
15              matchedSubTrees ←find other sub-trees that have the same root as curSubTree;
16              IF matchedSubTrees is not empty THEN
17                  newSeg ← create a new segment by merging matchedSubTrees with curSubTree
                        such that shared nodes only appear once in the segment;
18                  remove matchedSubTrees from subTreeSet;
19              ENDIF;
20          ENDIF;
21          IF newSeg is not NULL THEN
22              add newSeg into GSegSet;
23              update QEP such that the whole newSeg sub-tree in QEP is replaced by a newly
                    created virtual node;
24          ENDIF:
25          remove curSubTree from subTreeSet
26      UNTIL subTreeSet is empty;
27      REPEAT
28          call procedure "FindSegments(newQEP)";
29          segRels ← create segment dependency relationships between any two segments that are
                found in two consecutive iterations if they are connected by a virtual node;
30          add segRels into GSegRelSet;
31      UNTIL all nodes in the original QEP are processed
32  }
```

Figure 8: Decomposition algorithm – the first pass

The first pass of the algorithm creates a set of smaller queries such that pipelined operations are never interrupted., It does not, however, take cost information into consideration. Therefore, it may generate a "skewed" solution, meaning that some resulting segments (sub-queries) may be more costly than others (see Section 3.5 for a description of the skew of a solution).

A "skewed" solution has two major drawbacks which makes it impractical. The first drawback is that some of the generated segments may themselves be large, costly queries. Case A in Figure 9 shows this situation. In Figure 9, the percentage number beside each segment represents the segment's cost as a percentage of the total QEP. As we can see in Case A, segment III covers 97% of the total cost while the other three segments together cover the remaining 3%. In this situation, breaking a large query this way will not solve our original problem. It is more reasonable to decompose segment III further, if possible.

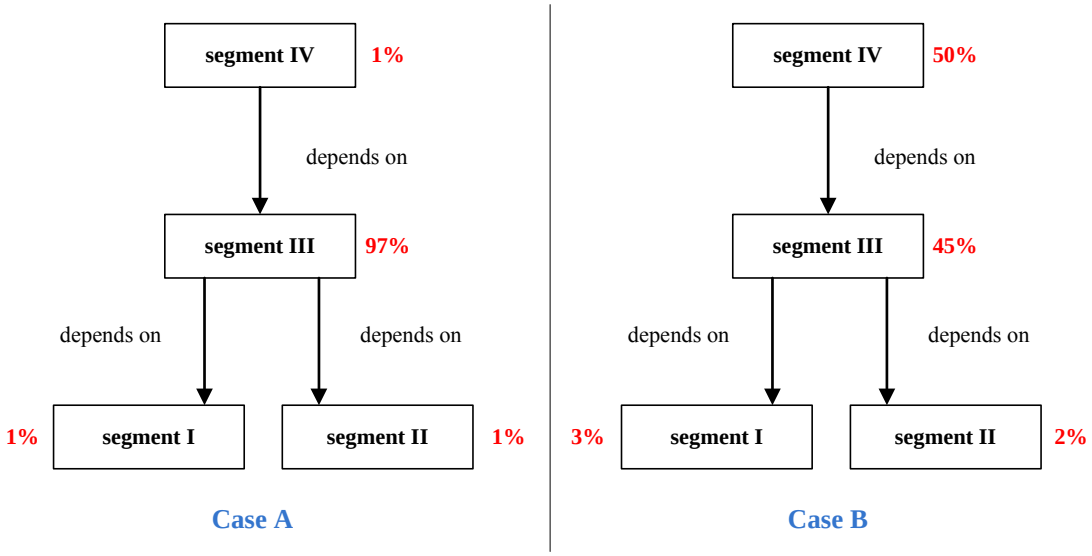

Figure 9: Examples of skewed solutions

The second drawback of a skewed solution lies in the possibility of unnecessary execution overhead such as that shown in Case B in Figure 9 . As will be seen in Section 3.6, our approach of decomposing a large query into multiple smaller queries incurs overhead related to the storage of intermediate results. In Case B of Figure 9, segment I and segment II each cover a very small portion of the total cost. If we implement this solution, the intermediate results from these two segments are stored in temporary tables, thus introducing additional overhead. In this situation, it is better to merge both segments I and II into segment III.

In order to overcome the drawbacks, the algorithm needs to be extended such that a more cost balanced solution is reached. The second pass of the algorithm aims to implement this goal. Figure 10 shows the pseudo code for this pass.

```
1   ReOrganizeSegments (QEP) {
2       call procedure "FindSegments(QEP)" to generate the global segment set "GSegSet" and the
            global segment relationship set "GSegRelSet";

3       FOR each segment(curSeg) in GSegSet
4           calculate the cost for curSeg based on QEP compiler information;
5       ENDFOR;

6       curSKF ← calculate the "skew factor" for GSegSet;
7       validSKFRange ← a pre-defined acceptable SKF range;

8       REPEAT
9           minSeg ← find the smallest segment (having minimum cost) in GSegSet.;
10          conSeg ← find the smallest segment that is connected to minSeg in GSegRelSet (either
                depends on or is depended on minSeg);
11          newCBSeg ← merge minSeg and conSeg to create a new larger CB-segment;
12          update GSegSet such that minSeg and conSeg are removed and newCBSeg is added;
13          update GSegRelSet such that all segment dependency relationships that involve minSeg
                and conSeg are modified correctly to involve newCBSeg instead;
14          curSKF ← calculate the "skew factor" for GSegSet;
15      UNTIL curSKF is within validSKFRange OR there is only one segment left in GSegSet;

16      IF there is only one segment left in GSegSet THEN
17          notify "Exception Management" module that a cost balanced solution is impossible
18      ENDIF;
19  }
```

Figure 10: Decomposition algorithm – the second pass

The pseudo code in Figure 10 defines how a cost-balanced solution is reached by merging the segments that are found by the first pass of the algorithm. It does not consider decomposing segments. Using our algorithm, decomposing a large segment always interrupts a pipelined operation. This is usually much less efficient in practice and can incur excessive overhead. For this reason, our algorithm always tries to reach a cost-balanced solution through merging first. If it is impossible to do so, a message is generated to bring a DBA's attention (or someone else who is running the query decomposition). When received the message, this person may ignore it and just think of the large query as un-decomposable, or he/she may choose to manually inspect the nodes as well as their cost information within each segment to determine whether or not to break a segment further to reach a more cost balanced solution by interrupting a pipelined operation. The rule of thumb for this manual process is that the smaller segments identified should equally share the cost of the original large segment. In our algorithm, an administrative parameter is used to control whether human intervention is allowed to break segments manually.

We note that it is not always possible to decompose a large query into a cost-balanced solution. One common example of this situation is when the cost of a single node (not a segment) covers the majority of the total cost because our algorithm does not handle the decomposition of a single operator thus making it impossible to decompose such a query. Such a case is shown in Figure 11. When applying the decomposition algorithm to this sample QEP, only one segment is generated because all the operators in the QEP are pipelining operators. Among all these operators, the Table Scan A node (node 6) alone covers almost all the total QEP cost (95%). In this situation, even with

human intervention we cannot find a cost-balanced solution. Moreover, applying the decomposition in this case incurs unnecessary execution overhead. Our decomposition algorithm detects the existence of such a case and provides appropriate feedback to a DBA and/or the query submitter to indicate that the query cannot be decomposed.

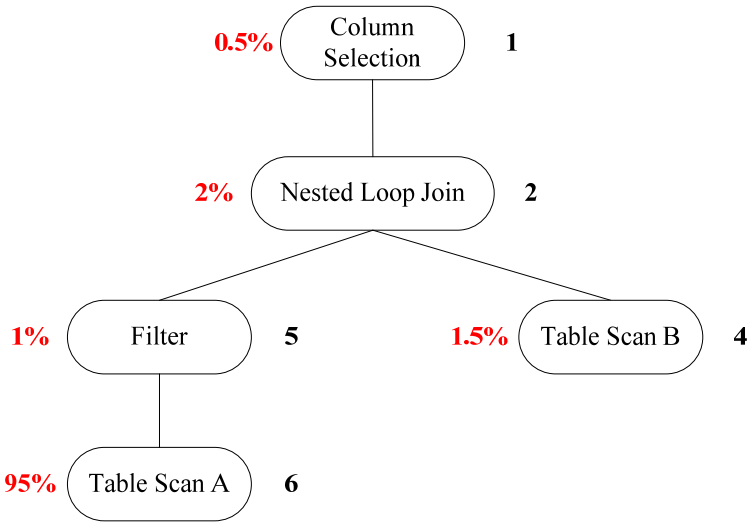

Figure 11: An example of a query that cannot be decomposed

3.5 Skew Factor

The second pass of the decomposition procedure uses the "skew factor" (SKF) of a solution extensively to determine a cost-balanced solution. In this section, we will explain in detail what a SKF is and how it is calculated.

Suppose that for a set of segments **SEGS = {seg_1, seg_2… seg_n}**, its related cost set is **COSTS = {$cost_1$, $cost_2$… $cost_n$}**, in which $cost_1$ is the cost value for seg_1, $cost_2$ is the cost value for seg_2, and so on. The SKF for **SEGS** measures how skewed **SEGS** is in terms of **COSTS**. To put in another way, the SKF value for **SEGS** measures the variance of **COSTS**. The higher the variance is, the higher the SKF value should be.

Equation 1 defines how the SKF value for **SEGS** can be calculated. In this equation, **VAR** represents sample variance and ***cost'*** is the average value of **COSTS** that is calculated by equation 2.

$$\begin{aligned} \textbf{\textit{SKF}}\ \textbf{(SEGS)} &= \textbf{VAR (COSTS)} \\ &= \textbf{VAR (cost}_1\textbf{, cost}_2\textbf{... cost}_n\textbf{)} \\ &= \sum_{i=1}^{n}(costi - cost')^2 \Big/ (n-1) \end{aligned}$$

Equation 1: Skew factor

$$\begin{aligned} \textbf{\textit{cost'}} &= \textbf{MEAN(COSTS)} \\ &= \textbf{MEAN (cost}_1\textbf{, cost}_2\textbf{... cost}_n\textbf{)} \\ &= \sum_{i=1}^{n} costi \Big/ n \end{aligned}$$

Equation 2: Average segment cost

In the equations above, the segment costs can be specified as either absolute values (in whatever appropriate unit) or relative values. The relative cost value of a segment in **SEGS** is defined as the percentage of the segment's absolute cost value over the total absolute cost value of all the segments in **SEGS**. The advantage of using relative cost values in Equations 1 and 2 is that it normalizes the calculated SKF value to the range of [0, 1]. Without the normalization, there is no way to easily specify a general threshold SKF value that can be employed by the second pass of the decomposition algorithm to find a cost-balanced solution. The SKF value calculated using Equations 3.1 and 3.2 would fluctuate widely depending on the query and how the query is decomposed.

In our approach, the relative costs of segments are employed in calculating the SKF value. The default administrative threshold value is set as 0.07, which corresponds to a "30% vs. 70%" cost distribution in a 2-segment solution, meaning that a large query can be decomposed into 2 smaller segments with one covering 30% of the total cost and another covering 70% of the total cost. Any solution whose SKF value is greater than the threshold value is considered as a skewed solution by our algorithm and therefore needs to be merged (or decomposed) further.

3.6 Executing Segments

The decomposition algorithm breaks a large QEP into a set of inter-dependent smaller segments and can form a segment schedule for the QEP. Following the schedule, the execution of the set of generated segments will generate the same result as the original large query does (see Section 3.7 for the proof).

There are two main problems that need to be solved. The first problem is how to store the intermediate results of a segment so that dependent segments can make use of the results. In our approach, we solve this problem by creating temporary database tables to hold the intermediate results. The overhead resulting from this solution includes: 1) the cost of creating empty temporary tables, 2) the cost of inserting intermediate results into the temporary tables, and 3) the cost of retrieving the stored intermediate results from the temporary tables. The overhead could be large, especially in cases where segments are created by breaking a pipelined operation. In our approach, this type of overhead is unavoidable due to the fact that our approach is implemented outside the database engine.

However, techniques that are able to exploit advanced database optimizer information could reduce the overhead greatly.

The second problem relates to how to execute a segment in practice. So far in this chapter, a segment is expressed as an operator tree, which can be thought of as a QEP if virtual nodes are consider as other regular physical nodes that could appear in a real QEP. A segment expressed in this way cannot be executed directly by a database compiler. A transformation of the segment from a QEP form to an executable form is necessary. In our work, we use an approach similar to the one used by Venkataraman et al. [27] to translate a QEP into a declarative SQL statement.

The basic step of the transformation is to traverse a QEP and translate each operator encountered into a part (or several parts) of the resulting SQL statement. For example, a filter node in a QEP can be translated into a "where" condition in a SQL statement. When all the operators in the QEP are translated, we then assemble all the translated parts together in a proper way such that a syntax-correct SQL statement is generated. During this process, we need to acquire additional information from the optimizer to complete the transformation, e.g. the type of condition used in a filter. In our approach, virtual nodes are treated as table scan nodes. The input table for the scan is an intermediate temporary table which is used to hold the data produced by the segment (or sub-query) referred to by the virtual node.

The translation procedure is highly vendor-specific because the crucial compiler information is different among different database compilers. We therefore postpone the detailed discussion of the segment-to-statement procedure until Section 4.5 where the translation procedure for DB2 is examined.

3.7 Decomposition Argument

Proposition: Given a query Q, the decomposition algorithm produces a set of queries {Q_1, Q_2, …, Q_n} and a dependency graph G such that if the queries Q_1, Q_2, …, Q_n are executed in an order determined by G, then they produce the same result relation as Q.

Assumption:

We assume that 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 our approach can not be guaranteed. Our proof below is based on this assumption.

Argument:

Given a QEP for a query Q, we know each edge of the QEP corresponds to a relation that is the result of execution of the source nodes of the edge. The decomposition algorithm identifies segments that can be executed as the sub-queries Q_is and replaces each segment with a virtual node by placing the result of its Q_i on the edge leaving that node. During this process, the algorithm maintains the same operator sequence within each segment as that in the original QEP for the query Q. The result of the set of replacements is a dependency graph G.

The original QEP is a tree, so G is a tree with each node of G representing a sub-query Q_i. The execution of the Q_is is determined by moving up G from the leaves such that a node in G for a query Q_j is only executed after its children, if any, have executed. G maintains all dependencies in the original QEP so: (1) each Q_i will receive the same input

as its corresponding segment in the QEP; (2) when all Q_is are executed according to G, the ordering of the operators encountered is the same as that in Q except some virtual nodes along the execution paths which, however, do not change the result because they just simply store the intermediate results for previous segments. $\{Q_1, Q_2, \ldots, Q_n\}$ will therefore produce the same result as Q.

Chapter 4

Query Disassembler

Query Disassembler is a prototype implementation of our approach to query decomposition using IBM DB2. It implements the decomposition algorithm and provides a framework for managing the decomposition process and scheduling the execution of the resulting set of smaller queries.

4.1 The Framework

Figure 12 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***).

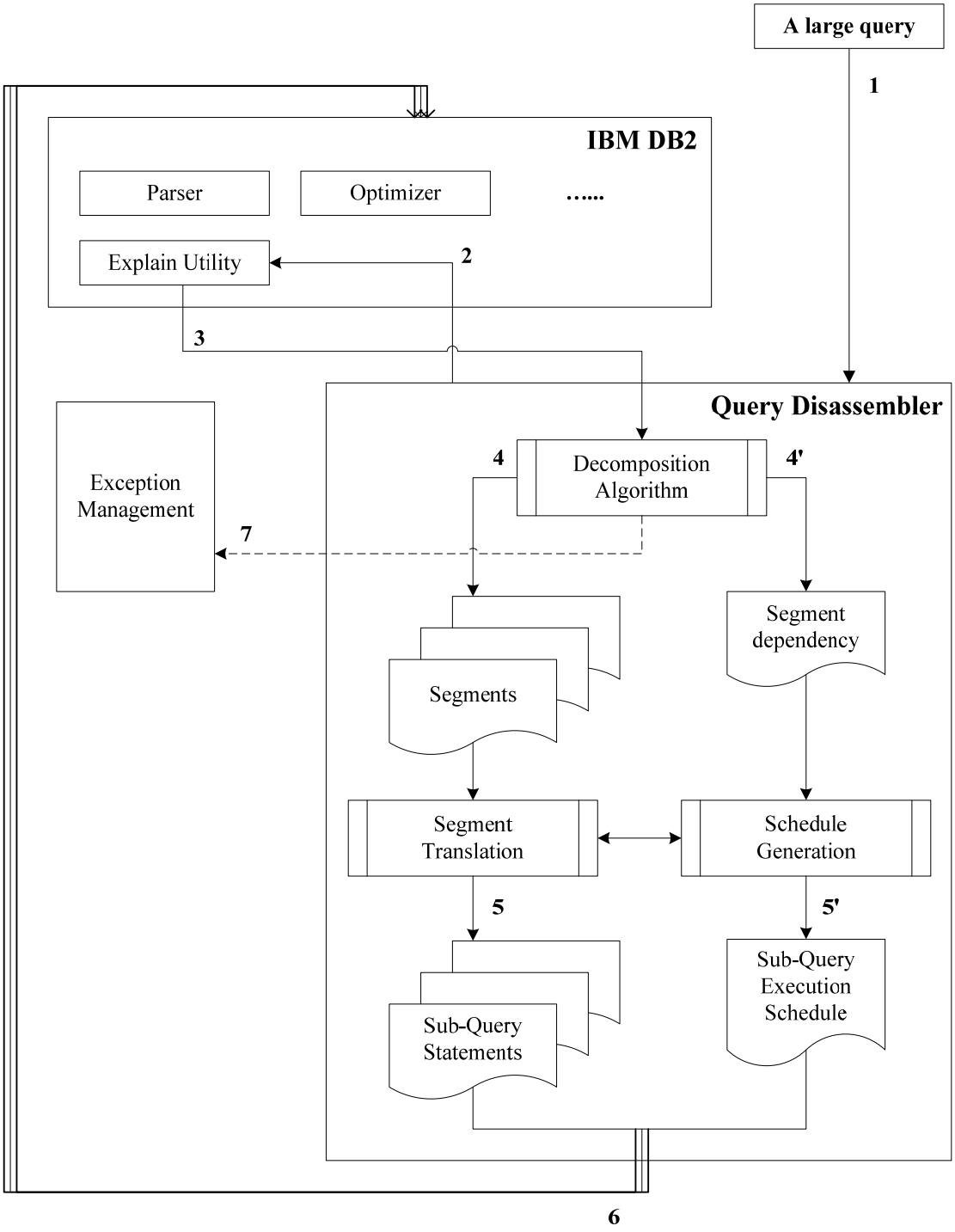

Figure 12: Query Disassembler framework

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.

4.2 Graphical User Interface

Figure 13 shows the main Graphical User Interface (GUI) for Query Disassembler. The left part of the GUI (Part I) lists the explained query instances and query statements that are returned by the DB2 Explain utility [22]. Details of DB2 Explain are found in Appendix C. Two SQL statements are shown for each explained SQL query. One is the original SQL statement that is submitted by the user, and the other is the optimized SQL statement that is suggested by the DB2 compiler as a result of applying the compiler's internal rewriting rules on the original statement. The optimized SQL statement is executed more efficiently than the original query. In our work, the optimized SQL statement is mainly used for translating segments into their equivalent SQL statement counterparts, which will be explained in detail in Section 4.5.

The right part of the GUI (Part II) shows the QEP for the explained query. This QEP shows the estimated operational tree structure and also includes other useful performance-related data, such as cost, node predicates, and so on. The QEP and its

related data are either directly provided by the DB2 Explain utility or calculated from that information.

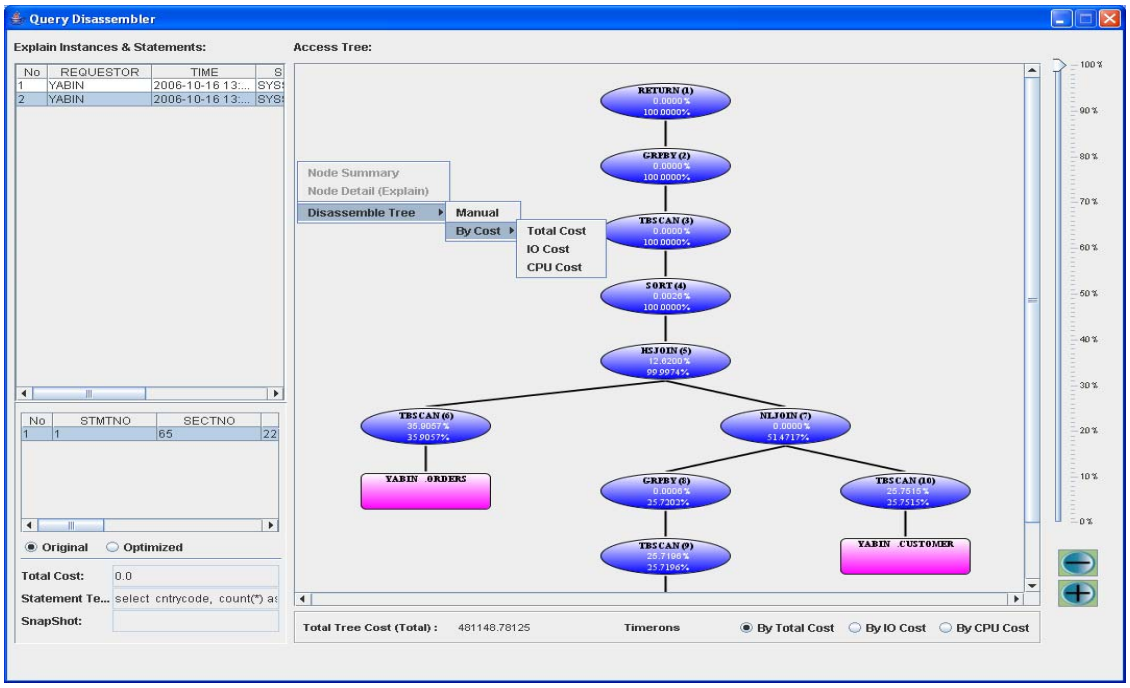

Figure 13: GUI of Query Disassembler

One of the most important pieces of information is the compiler-estimated execution-cost value for the entire QEP and each of its internal nodes. The cost is measured in a DB2-specific unit, called ***timeron,*** [24] and is further divided into sub-costs that are directly linked with IO and CPU. The total cost for the entire QEP is shown at the bottom of Part II.

The cost for each internal node of the QEP is expressed in the following ways. The latter three costs are useful in the second pass of our decomposition algorithm to determine a more cost-balanced solution.

- The absolute estimated cost.

- The node's cost as a percentage of the cost of the entire QEP.
- The accumulated cost value up to this node in the QEP.
- The accumulated cost percentage up to this node in the QEP.

The accumulated cost value up to a node, say Node A, in a QEP refers to the total cost of all the nodes in a sub-tree of the QEP rooted by Node A. The accumulated cost percentage up to a node in a QEP is the accumulated cost value up to the node in the QEP expressed as a percentage of the total cost of the QEP. The accumulated cost value as well as the percentage up to a node is shown directly in the GUI on each node beneath the node name and the node ID. Each node within the QEP as shown in Figure 13 is given a unique integer number for easy reference.

When we choose to disassemble a QEP from the popup menu, the user is given a choice of using the decomposition algorithm to break up the tree automatically (the "By Cost" option) or to disassemble the tree manually (the "Manual" option). The "cost" used to decompose the QEP can be the IO-related cost, the CPU-related cost, or the combined total cost depending on whether the large query to be decomposed is IO-intensive, CPU-intensive, or mixed.

The "By Cost" option utilizes the decomposition algorithm as we discussed in Section 3.4 to break up the query automatically by analyzing its QEP structure as well as the related cost information. If a cost-balanced solution can be reached, then the GUI pops up a window to illustrate how the QEP is decomposed by displaying the breakpoints, that is the node numbers above which the QEP is decomposed. A segment schedule object (a Java object) is also created. The segment schedule object contains information about what segments are decomposed from the large query and their execution order

(schedule). Section 4.3 gives a detailed explanation of this object. If the tree cannot be decomposed, a message to this effect is displayed.

The "Manual" option of the Query Disassembler allows a DBA to specify a list of breaking points that he/she thinks is appropriate for decomposition. The manual disassembly procedure does not employ the decomposition algorithm as explained in Chapter 3, but it directly utilizes the specified node numbers to form segments and the corresponding execution schedule. Similar to the "By Cost" option, a segment schedule object is created after the manual decomposition procedure is done.

Figure 14 illustrates how the manual disassembly procedure works. The left part of Figure 14 shows an example QEP and we suppose that the specified breakpoints are 3 and 4 (as shown by "X" marks in Figure 14). As shown in the right part of Figure 14, the first step of the procedure creates two segments (segment I and segment II) such that each segment is equivalent to a sub-tree of the QEP and has one of the breakpoints as its root node. In the second step, two virtual nodes are created to replace the two segments in the original QEP and form the third segment (segment III) which depends on both segment I and segment II.

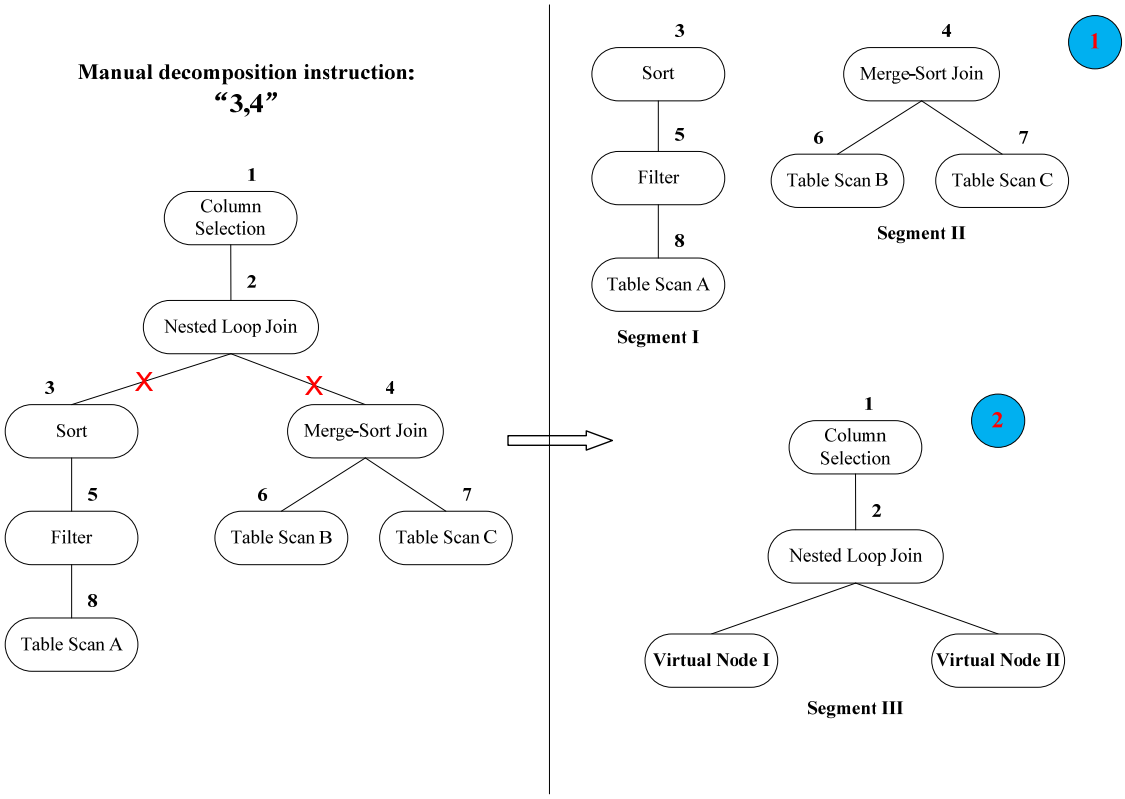

Figure 14: An example of manual disassembly procedure

4.3 Segment Schedule Object

Figure 15 shows the class diagram for the segment schedule object provided by our program. In this diagram, *Schedule* is the core component for segment scheduling. It includes one *Query* object that represents the original large query and a set of *ScheduleUnit* objects, each of which stands for a single scheduling unit (for example, a single segment) that is managed by the *Schedule* object. Each *ScheduleUnit* object has its own *Query* object which represents one small query that is decomposed from the original large query. A *ScheduleUnit* object is also used to create and populate the temporary tables that are needed to hold the intermediate results.

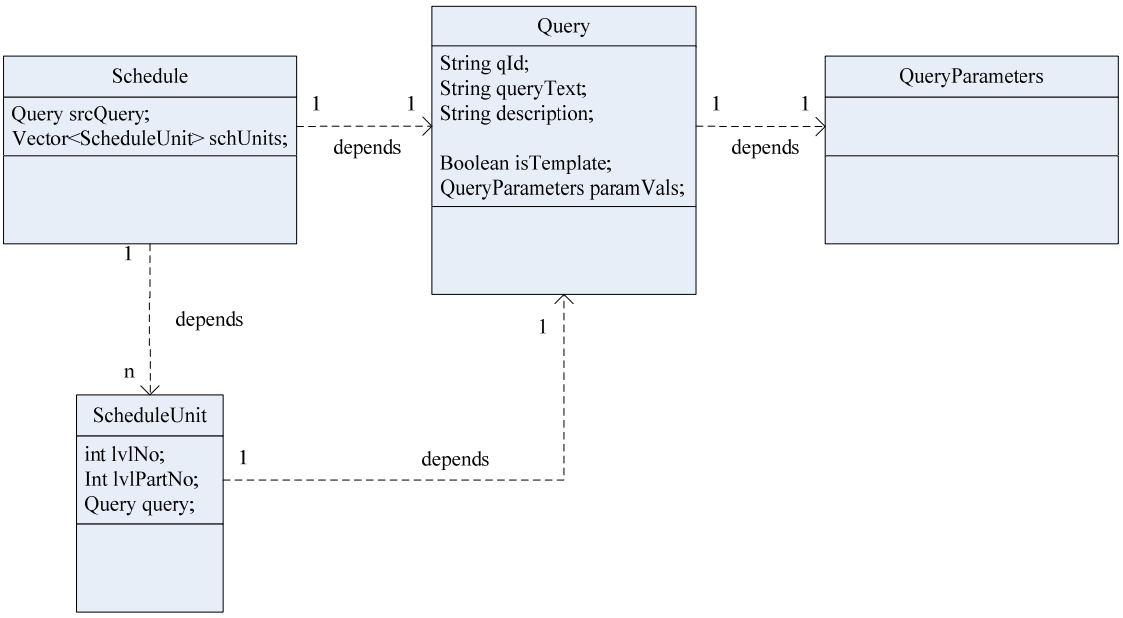

Figure 15: Segment schedule class diagram

4.4 DB2 specific operators

Other than the common operators listed in Section 3.2, there are some IBM DB2 specific operators, like CMPEXP and EISCAN, that can appear in an IBM DB2 QEP. Our algorithm supports some of these operators. The supported DB2 specific operators are treated in the same way as the common operators.

The following heuristics define how the DB2 specific operators are handled by our algorithm. This is supplementary to the rules defined in Section 3.2. If the QEP for a large query contains unsupported DB2 operators, the query is not considered by our algorithm and is simply passed to the Exception Management Module.

- TQUEUE (for parallelism support), RQUERY/SHIP (for federated system), CMPEXP and PIPE (for debug usage) are not supported by our algorithm.

- TEMP (storing data in a temporary table), GENROW (generates a table of rows, using no input from tables, indexes, or operators) are supported and are treated as blocking operators.
- IXAND (index and), RIDSCN (row ID scan), EISCAN (scans a user defined index to produce a reduced stream of rows) are supported and are treated as pipelining operators.

4.5 Translating segments in DB2

Although the DB2 explain utility provides a significant amount of performance-related information to aid the segment-translation processes, in many situations a segment decomposed from a QEP cannot be directly translated into an SQL statement by following the general translation procedure described in Section 3.6. A common situation is how to handle the DB2 specific ROWID predicates. A ROWID predicate is a predicate that includes ROWID as an operand (ROWID is used by the DB2 compiler to directly pinpoint a row rather than to go through the regular search procedure and therefore is much more efficient). DB2, however, does not provide facilities to get ROWID in a query's SQL statement.

In our approach, when we encounter such a situation, we utilize the optimized SQL statement that is provided by DB2 Explain as the source to the translation process. This optimized statement is equivalent to the original query statement and consists of multi-level nested sub-queries. After studying this version of a query we found that it is most often amenable to our translation process. Figure 16 shows the optimized form for TPC-H Q21.

```
1   select    q10.$c0 as "s_name",
2             q10.$c1 as "numwait"
3   from (
4             select    q9.$c0,
5                       count(*)
6             from (
7                       select distinct q8.$c0
8                       from (
9                                 select    q7.$c6
10                                from      lineitem as q1 right outer join (
11                                                  select distinct     q3.l_orderkey,
12                                                                      q3.l_suppkey,
13                                                                      q2.s_name
14                                                  from      supplier as q2,     lineitem as q3,
15                                                            orders as q4,       nation as q5,
16                                                            lineitem as q6
17                                                  where     (q2.s_suppkey = q3.l_suppkey)
18                                                            and (q4.o_orderkey = q3.l_orderkey)
20                                                            and (q4.o_orderstatus = 'F')
21                                                            and (q3.l_commitdate < q3.l_receiptdate)
22                                                            and (q2.s_nationkey = q5.n_nationkey)
23                                                            and (q5.n_name = 'SAUDI ARABIA')
24                                                            and (q6.l_suppkey <> q3.l_suppkey)
25                                                            and (q6.l_orderkey = q4.o_orderkey)
26                                                ) as q7 on
27                                                  (q1.l_orderkey = q7.$c3)
28                                                  and (q1.l_suppkey <> q7.$c4)
29                                                  and (q1.l_commitdate < q1.l_receiptdate)
30                                ) as q8
31                      ) as q9
32                      group by q9.$c0
33            ) as q10
34  order by
35            q10.$c1 desc,
36            q10.$c0;
```

Figure 16: DB2 optimized SQL statement for TPC-H Q21

In Section 3.4 we point out that any sub-tree within a query's QEP can be viewed as another QEP that corresponds to a smaller query contained in the original large query. Therefore, from the QEP point of view, a query's structure is also nested in multi-levels. The similarity between a query's QEP and its optimized SQL statement makes the

optimized SQL statement an excellent resource for the task of translating segments into their corresponding SQL statements.

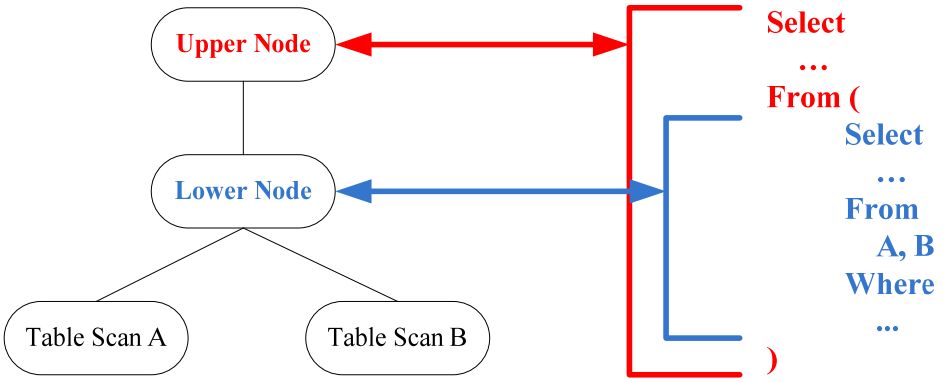

Figure 17: Example of matching a query's QEP with its Optimized SQL Statement

Figure 17 shows a simple example of how the matching process works. A rule of thumb for this process is to match level by level. We start by matching the highest node in a QEP to the outermost sub-query in the optimized SQL statement and continue until the lowest possible node in QEP is matched to the innermost sub-query in the optimized SQL statement. Although such a matching process works in DB2, we can not guarantee that it will work for other DBMSs.

Chapter 5

Experiments

In this section we describe a set of experiments to evaluate the effectiveness of our approach for controlling the execution of a large query. The computer system used is an IBM xSeries® 240 machine with dual 1 GHZ CPUs, four PCI/ISA controllers, and 17 Seagate ST 318436LC SCSI disks. We use IBM DB2 Version 8.2 as the database server.

5.1 Workload

The workload consists of a set of small read-only queries and one large query, which is either the TPC-H Q21 or the TPC-H Q22 query. The small query set consists of eight parameterized OLTP-like read-only queries (see Appendix E for detail). Each client submits a random stream of these queries. The average response time for these queries is typically less than half second. We control the intensity of the workload by varying the number of concurrent clients.

Q21 is an IO-intensive query that accesses five different tables, four of which are relatively large in size. Its SQL statement is complex and includes aggregation and sub-queries. Q22 is a CPU-intensive query that accesses two different tables, including one large table. Its SQL statement is less complicated than that of Q21, but, in addition to aggregation and sub-queries, it also contains some mathematical operations.

We examined the QEPs of TPC-H queries 1 through 20 (Q1 – Q20) and found that all are highly skewed under the current experimental database configuration so there is no way to find a cost-balanced solution. Within each of the QEPs for this set of queries,

there is always a single ***Table Scan*** node that covers most of the total QEP cost (at least 90%). Q21 and Q22, however, are two queries that can be decomposed by the decomposition algorithm such that a cost-balanced solution can be reached. When running alone in our test-bed environment (no interference from any other query), Q21 takes about 60 seconds to run and Q22 takes about 30 seconds to run.

Using our algorithm, Q21 is broken into two smaller queries. The first query accounts for approximately 70% of the total cost and the second covers the remaining 30%. Similarly, Q22 is also decomposed into two smaller queries that account for 60% and 40% of the total cost, respectively. Unlike Q21, Q22 is decomposed such that a pipelined operation is interrupted. Figure 18 shows the QEP for Q22 and illustrates how it is decomposed. We do not show the QEP for Q21 here because it is too large to see clearly (Appendix F shows the QEP anyway), but the process of decomposing it and how the cost is distributed is similar to that of Q22.

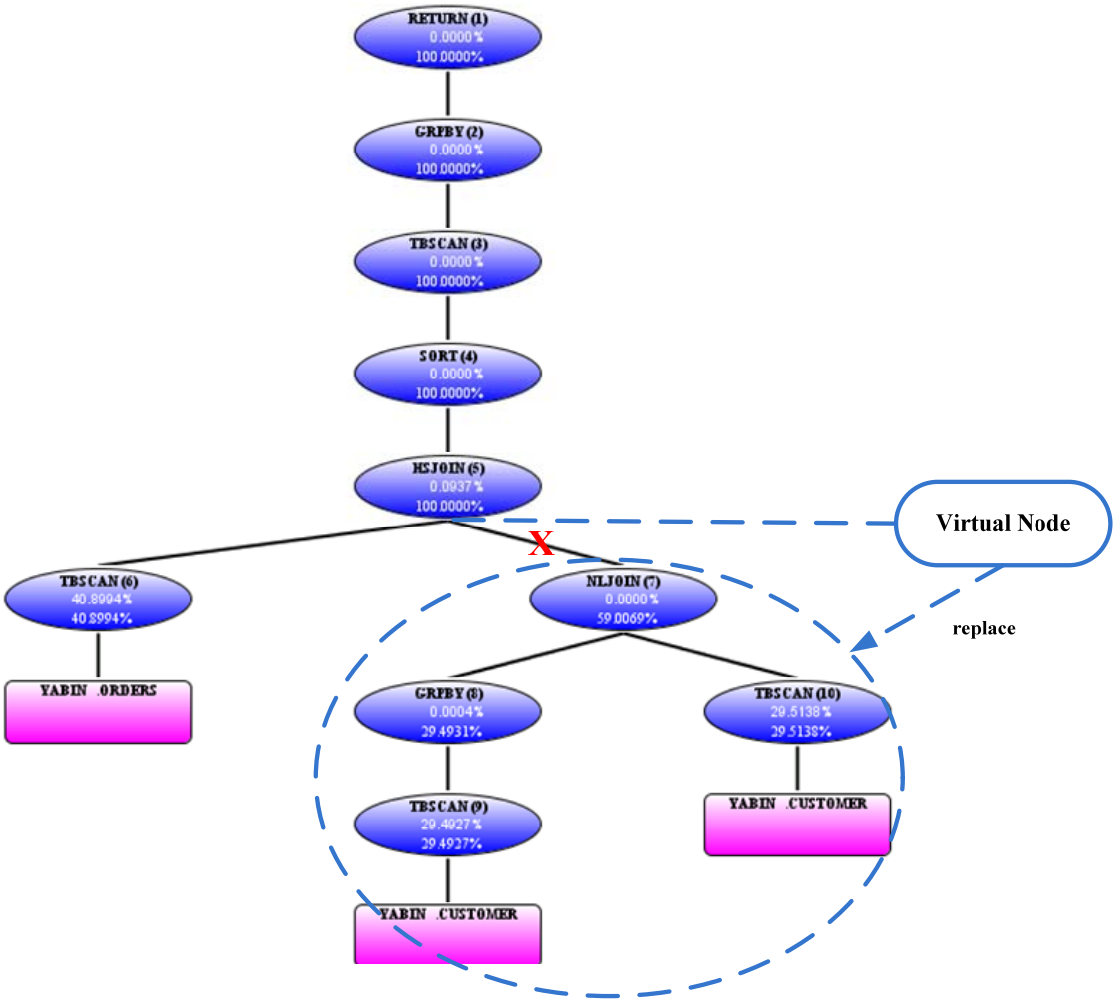

Figure 18: QEP of Q22 and its decomposition

In Figure 18, the QEP for Q22 is divided just above node 7 (NLJOIN) marked by an X, thus creating two segments – one is the sub-tree rooted at node 7 (segment I) and the other is the QEP for Q22 with a virtual node (segment II) replacing the sub-tree rooted at node 7. The cost estimates in Figure 18 show that segment I covers almost 60% of the total QEP cost for Q22 and segment II takes the remaining 40%.

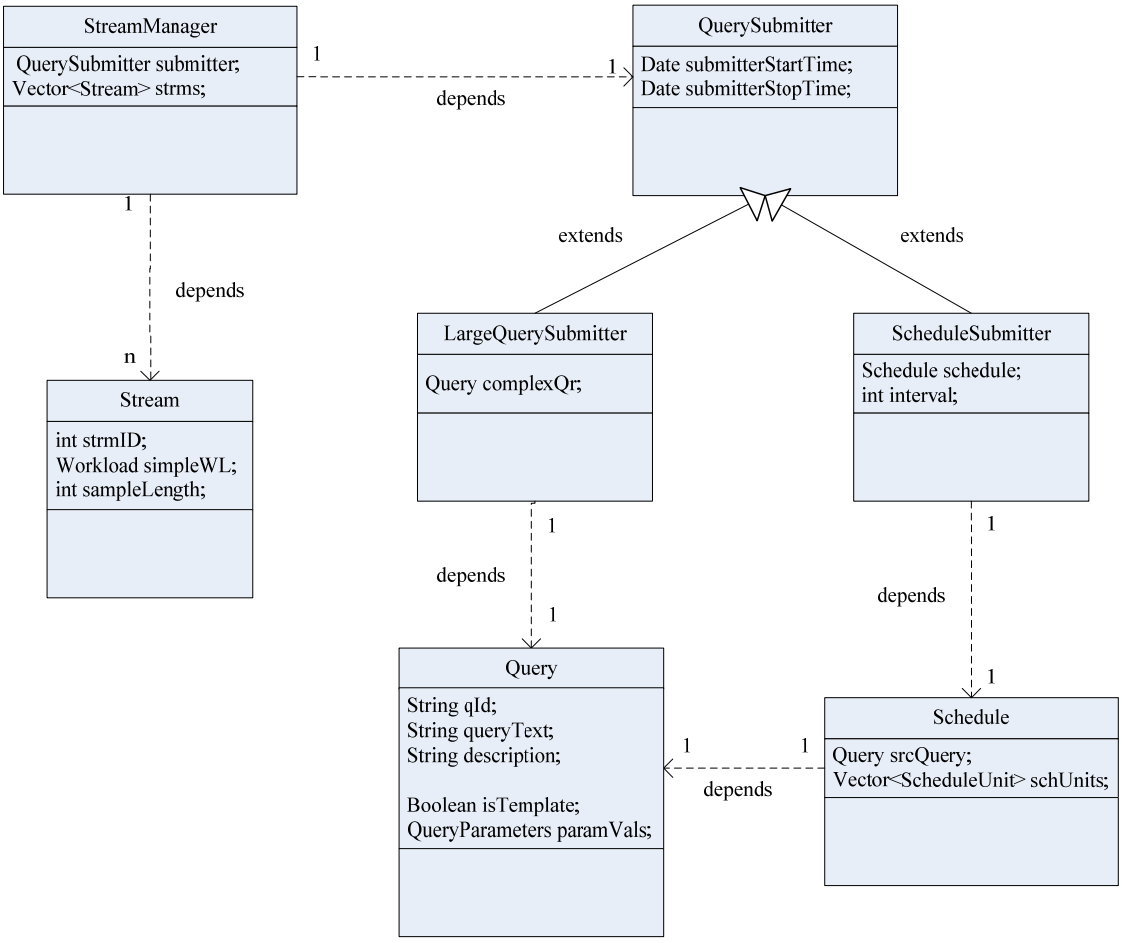

Figure 19: Workload generation class diagram

Figure 19 shows the class diagram of the workload generation in our experiments. In our work, the submission of the workload is controlled by a *StreamManager* object which, in turn, contains a set of *Stream* objects and a *QuerySubmitter* object. Each *Stream* object has a unique stream ID and represents a single client that submits the OLTP-like small queries. The number of *Stream* objects managed by the *StreamManager* object controls the workload intensity of the small queries. The *LargeQuerySubmitter* object, a sub-object of the *QuerySubmitter* object, is used to submit a large OLAP query to the database without decomposing it. The *ScheduleSubmitter* object, another sub-object of the

QuerySubmitter object, is used to submit a Segment Schedule consisting of the sub-queries that have been decomposed from the large query using the decomposition algorithm. The integer *interval* parameter of this object controls the length of the pause period (in seconds) between submitting consecutive queries in the schedule.

5.2 Experimental Scenarios and Database Configuration

We conducted experiments to test the effectiveness of our approach under four scenarios in which the large OLAP query causes different degrees of contention for resources.

- In scenario 1, the workloads run in separate database instances on the same system and just compete for system resources like CPU and IO.
- In scenario 2, the workloads run in the same database instance but use separate buffer pools, which add contention for general DBMS resources such as system catalogs and queues.
- In scenario 3, the workloads run in the same database instance and use the same buffer pool, which adds contention for memory resources.
- In scenario 4, the workloads access the same tables. This adds contention for the tables and indexes.

Within each case, four types of throughput data for the small query set are collected:

- Type1 – the small query set runs in the database alone.
- Type2 – the small query set and a large query (before decomposition) run simultaneously.

- Type3 – the small query set and a segment schedule (composed of the small queries that were decomposed from the large query) run simultaneously.
- Type4 – the small query set and a segment schedule (composed of the small queries that were decomposed from the large query) run simultaneously with a one minute pause between executing the small queries contained in the schedule. This data is used to confirm our observation that running a large query as a series of smaller queries will release all resources between queries in the series and so they are available to other parts of the workload.

In each of the 16 experimental cases (4 scenarios, 4 types of data), the workload is run 11 times in order to obtain a statistically-sufficient result. Appendix D shows method used to calculate the confidence intervals for all experimental cases. Each run lasts for 600 seconds and is sampled every 20 seconds. Within each run, the small query starts its run 60 seconds earlier than that of the large query (or its corresponding query schedule) and uses the time as a warm-up period. It runs continuously within each run and its throughput is monitored. On the other hand, due to the interference of the small query set, the execution time for the large query (or its corresponding query schedule) is substantially prolonged and therefore it runs only once per run. The first run is considered as a general database warm-up period, especially for the large query, and the results collected during this run are therefore excluded for the final analysis. Figures 20 to 27 show the throughput data for the small query set for the four cases. The analysis of the results is discussed in Section 5.3.

To accommodate the different purposes of the four scenarios, two databases, Db_1 and Db_2, are used for our experiments. Db_1 has one user table space (Db1Ts_1) and Db_2 has two user table spaces (Db2Ts_1 and Db2Ts_2). Within these table spaces, there

are four different sets of the standard TPC-H tables (TblSet1 to TblSet4) used in the experiments. Table 1 shows the size and the location of these table sets.

Table Set	Size	Database	Table Space
TblSet1	100MB	Db_1	Db1Ts_1
TblSet2	100MB	Db_2	Db2Ts_1
TblSet3	100MB	Db_2	Db2Ts_2
TblSet4	2GB	Db_2	Db2Ts_2

Table 1: Table sets configuration

The buffer pool size for each table space is scaled to 2% of the table space size. A more detailed description of the buffer pool configuration is provided in Section 5.3. Other key database parameters are configured as database default. No indices, other than the primary key index, are created on each of the database tables.

In our experiments, the large query (Q21 or Q22) always accesses the 2GB table set (TblSet4) and the small query set may access any table set (TblSet1 – TblSet4) depending on the experimental case. Table 2 shows which table sets are used in the various experimental scenarios.

Experimental Scenarios	Table Sets Used
Scenario 1	TblSet1, TblSet4
Scenario 2	TblSet2, TblSet4
Scenario 3	TblSet3, TblSet4
Scenario 4	TblSet4

Table 2: Table sets for experimental scenarios

5.2.1 Scenario 1: Separate Databases

This scenario tests the effectiveness of our approach in a situation where a large query competes with other queries for system resources such as CPU and disk I/O but does not share database-specific resources such as locks and buffer pool memory. The small query set and the large query run in two separate databases (Db_1 and Db_2 respectively). The large query accesses large tables in TblSet4 whereas the small query set accesses small tables in TblSet1. The size of the corresponding buffer pools is configured such that it is proportional to the table space size, which means that the buffer pool size for the table space Db2Ts_2 is 20 times as big as that of table space Db1Ts_1. The results of this scenario are shown in Figures 20 and 21 when the large query is Q21 and Q22, respectively.

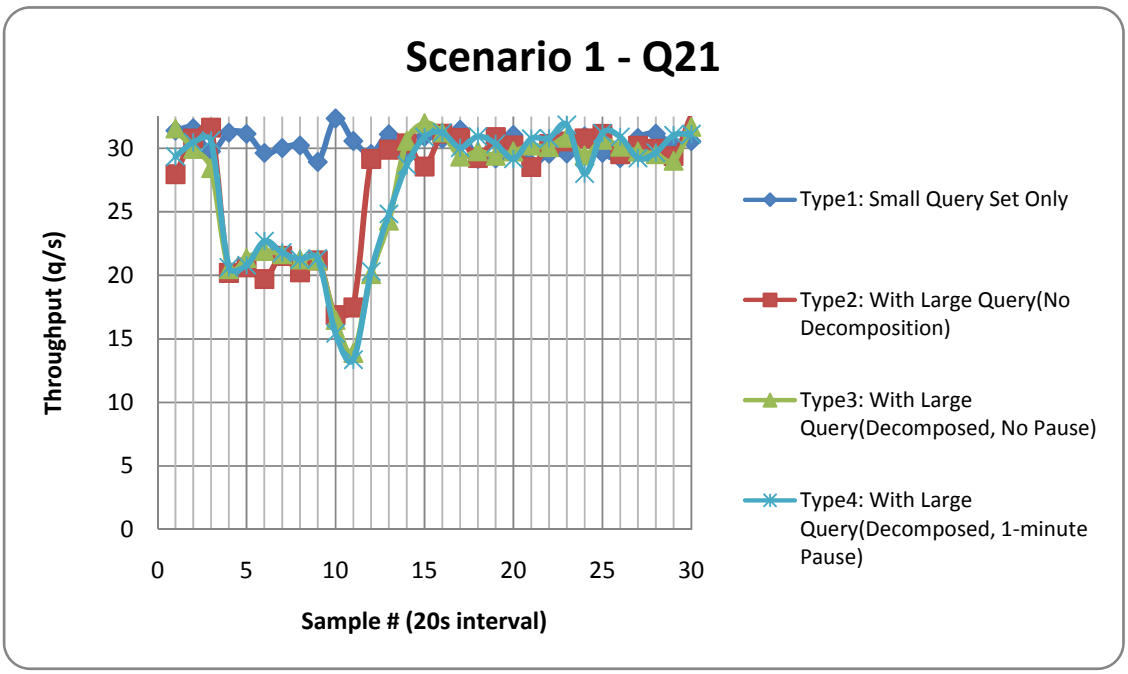

Figure 20: Scenario 1 – separate databases (Q21)

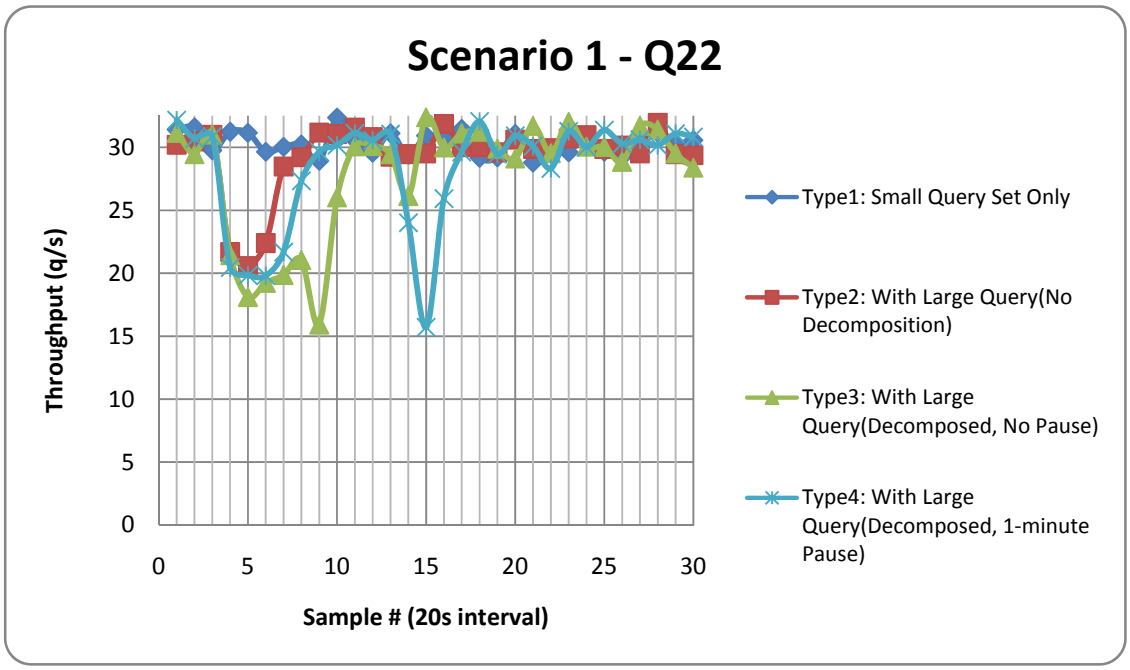

Figure 21: Scenario 1 – separate databases (Q22)

Figures 20 and 21, as well as Figures 22 to 27 that we will see in later sections, all confirm that the running a large query in a database has a significant impact on the performance of other workloads in the database. It also can be seen that in this scenario, our decomposition approach is unsuccessful. The throughput of the small query set is even worse when the large is decomposed, whether or not a 1-minute pause is applied. This is understandable, however, because in this scenario, the large query and the small query set compete only for operating system managed resources like CPU and disk I/O. The decomposition of a large query brings extra overhead of CPU and disk I/O usage and there is little that can be done by the DBMS to alleviate the performance degradation caused by the overhead.

5.2.2 Scenario 2: One Database, Separate Buffer Pools

Scenario 2 tests the effectiveness of our approach in a situation where a large query competes with other queries for both CPU and I/O resources and general DBMS resources such as catalogs and queues, but not for buffer pool memory. In this case, both the large query and the small query set run in Db_2. The large query accesses large tables in TblSet4 whereas the small query set accesses small tables in TblSet2. The buffer pool size for table space Db2Ts_2 is the same as in Case 1 and the buffer pool size for table space Db2Ts_1 is the same as that for table space Db1Ts_1 in Case 1. The results of this case using Q21 as the large query are shown in Figure 22 and the results using Q22 are shown in Figure 23.

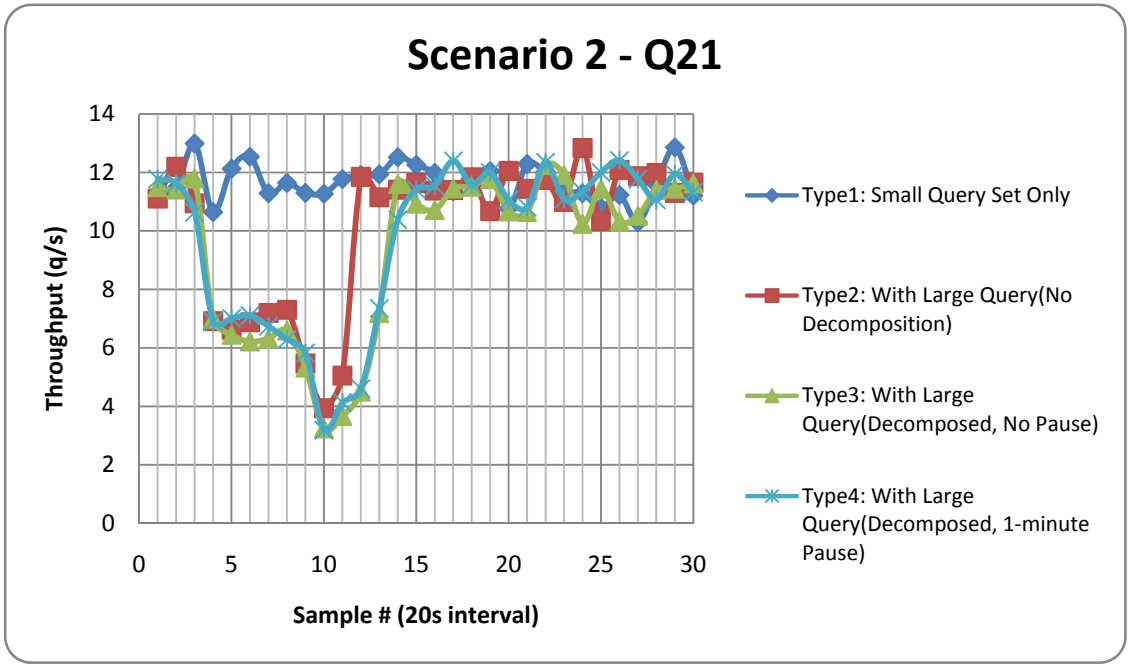

Figure 22: Scenario 2 – one database, separate buffer pools (Q21)

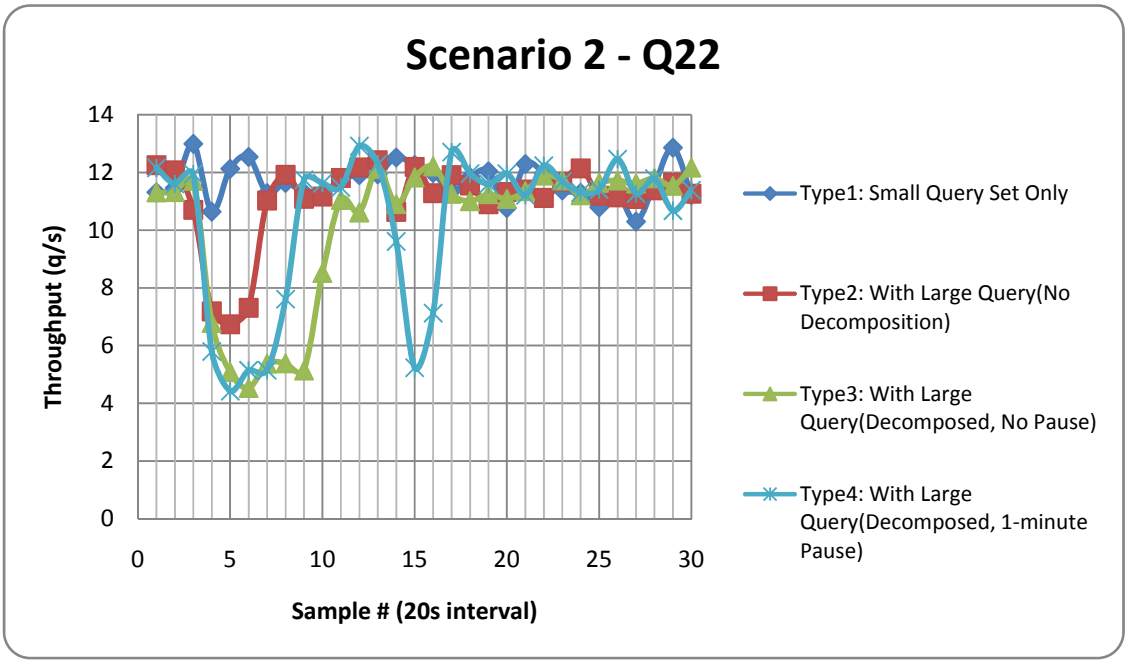

Figure 23: Scenario 2 – one database, separate buffer pools (Q22)

Compared with scenario 1, the four types of throughput data for the small query set in this scenario are all worse due to the added competition. Other than this point, however, the trends of how the throughput data change before and after decomposing the large query are similar to those in scenario 1. The decomposition approach still does not work well in this scenario.

5.2.3 Scenario 3: One Database, Shared Buffer Pool, Different Table Sets

Scenario 3 reflects the situation where a large query competes with other queries for all physical database resources, namely CPU, disk I/O and memory. In this case, however, there is no lock contention as the queries are accessing different sets of tables. Both the large query and the small query set run in Db_2. The large query accesses large tables in

TblSet4 whereas the small query set accesses small tables in TblSet3. There is a single, shared buffer pool and its size is configured to be the total of the buffer pool sizes in Case 2. The results of this case using Q21 are shown in Figure 24 and Figure 25 shows the results for Q22.

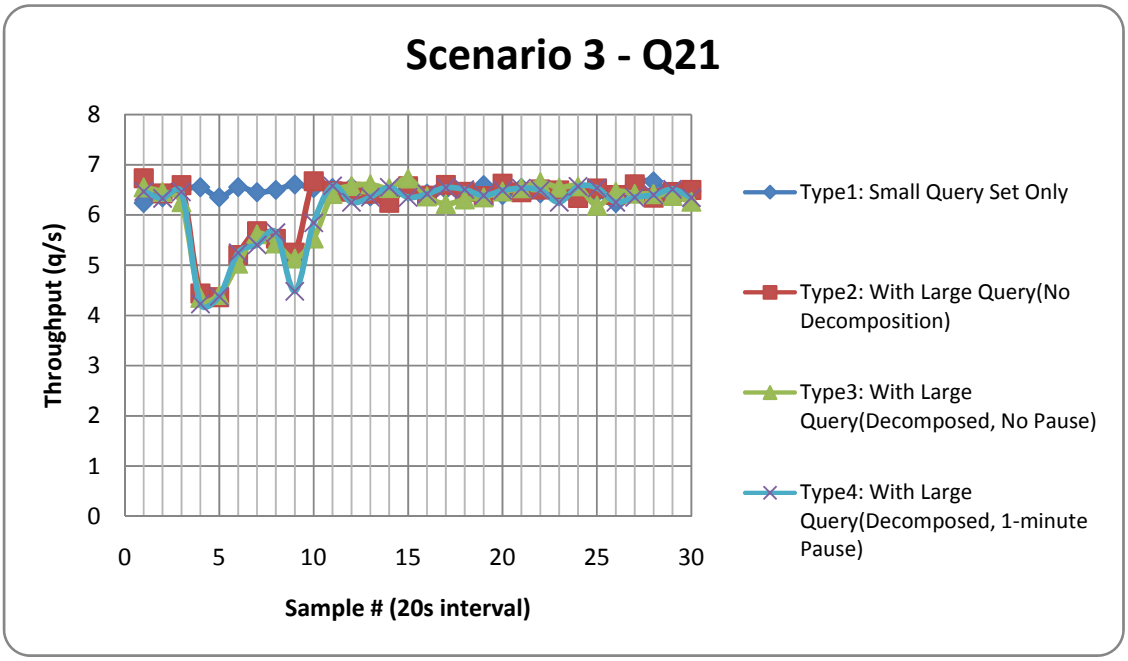

Figure 24: Scenario 3 – one database, shared buffer pools, different table sets (Q21)

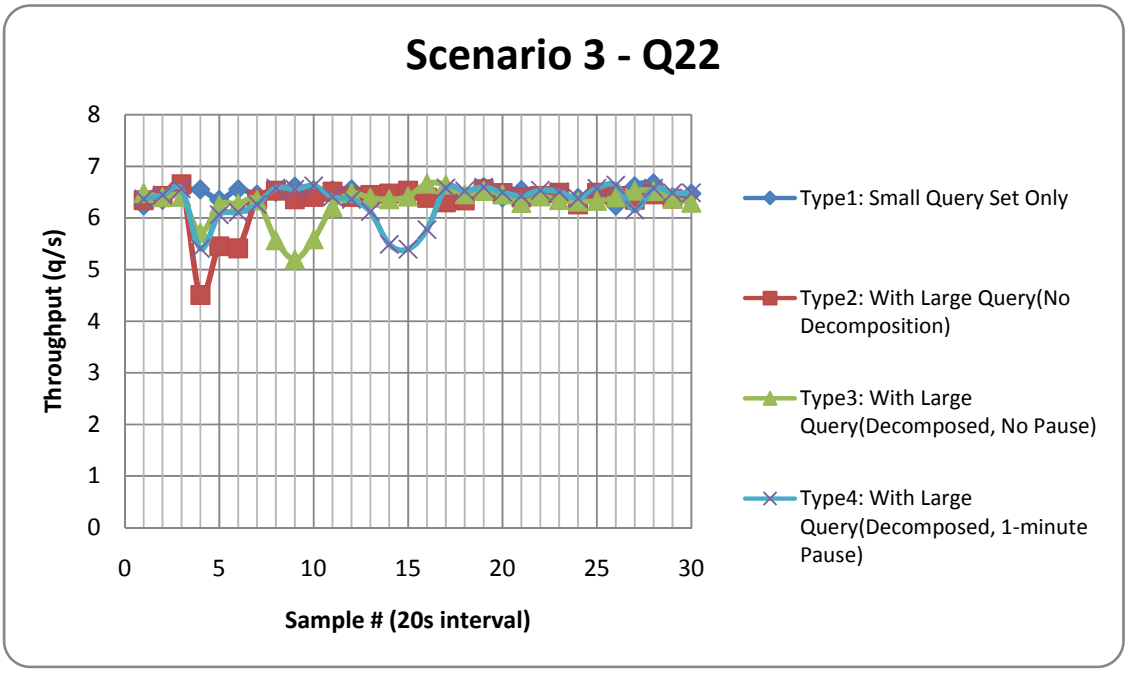

Figure 25: Scenario 3 – one database, shared buffer pools, different table sets (Q22)

Compared with scenarios 5.2.2 and 5.2.3, this scenario sees some improvement as a result of the decomposition of the large query, especially for the period between the 3rd sampling point to the 7th sampling point with TPC-H 22 as the large query. In Figure 25, both Type3 and Type4 data have two obvious performance drops along the curves. This can be explained by the overhead of writing intermediate temporary tables that is introduced by the decomposition approach. The similar trends also exist in other scenarios for both Q21 and Q22, although those for Q21 are not obvious.

5.2.4 Scenario 4: One Database, Shared Buffer Pool, Same Table Set

Scenario 4 reflects the situation where a large query competes with other queries for all physical database resources including locks. Both the large query and the small query set

run in Db_2 and access the same large tables in TblSet4. There is only one buffer pool involved in this case and its size is configured to be the same as that for Db2Ts_2 in Case 2. The results of this case using Q21 are shown in Figure 26 and in Figure 27 using Q22. Compared with scenario 3, this scenario sees more improvement that is brought by the decomposition of the large query.

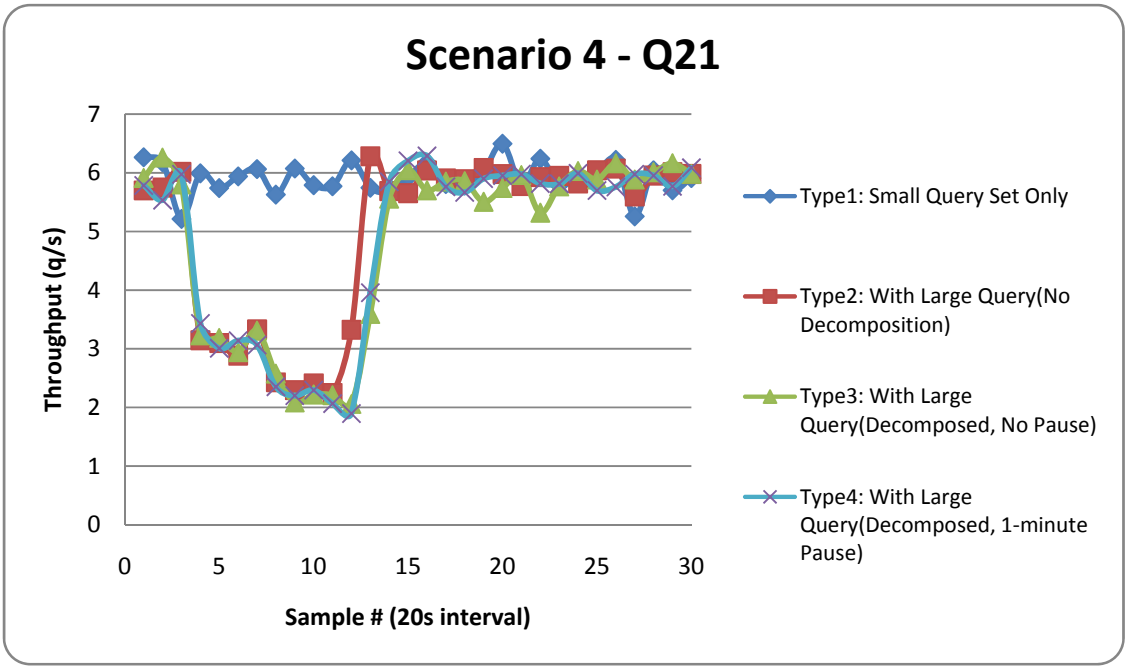

Figure 26: Scenario 4 – one database, shared buffer pools, same table set (Q21)

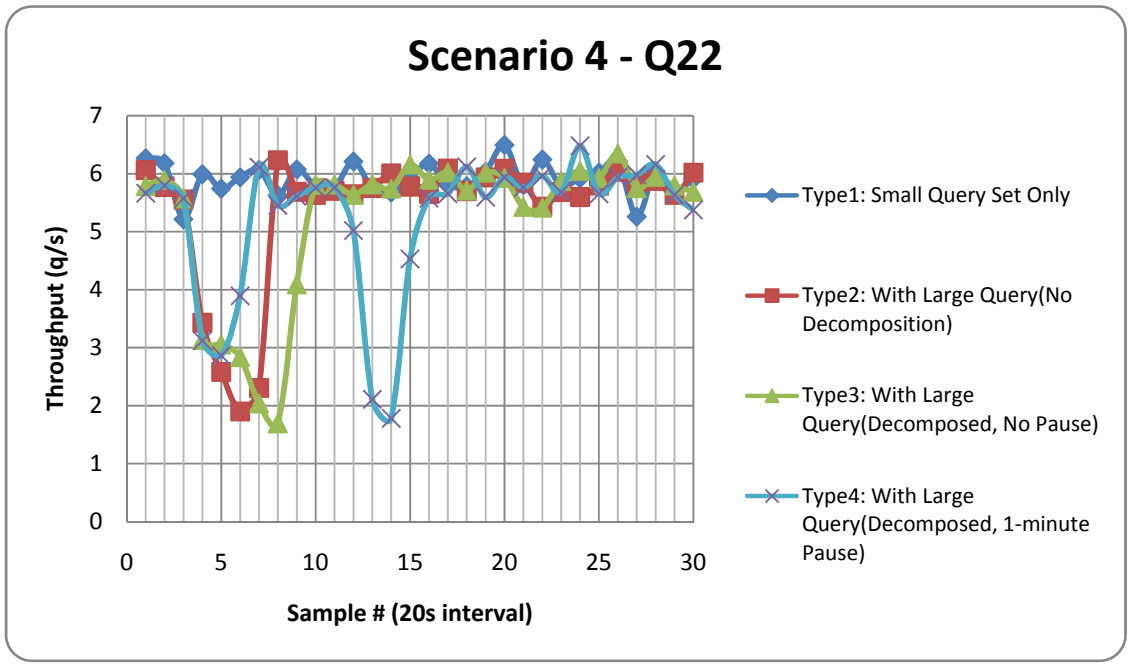

Figure 27: Scenario 4 – one database, shared buffer pools, same table set (Q22)

5.3 Analysis of the Results

We first observe that, as expected, decomposing the large queries causes significant increases in their response time. Table 3 shows how the response time changes for Q21 and Q22 respectively before and after they are decomposed.

Large Query	Experimental Scenario	Normal (s)	Decomposed (s)	Increase (%)
Q21	Scenario 1	162	197	21.6%
Q21	Scenario 2	157	197	25.5%
Q21	Scenario 3	155	191	23.2%
Q21	Scenario 4	172	195	13.4%
Q22	Scenario 1	67	126	88.1%
Q22	Scenario 2	64	134	109.4%
Q22	Scenario 3	57	133	133.3%
Q22	Scenario 4	78	108	38.5%

Table 3: The change of large query response time

As we can see from Table 3, the response time for query Q21, which is an IO-intensive query, increases an average of 20% over the four cases (162s normal execution versus 195s for the decomposed query). The response time for query Q22, which is a CPU-intensive query, increases an average of 87% over the four cases (67s normal execution, 125s for the decomposed parts). The increased response time of the decomposed queries is mainly due to the IO associated with the introduction of temporary tables. This additional IO is more significant for the CPU-intensive queries (Q22) than for the IO-intensive queries (Q21).

We also observe that increased contention for resources has a negative impact on the throughput of the OLTP-like workload. Tables 4 and 5 show the average throughput of the small query set over the "busy" period (different among experiment cases, but all between sampling points 3 and 18) within each experimental scenario when the large query is Q21 and Q22, respectively. Looking at the Type 1 column (small queries running alone) in both tables, we see that throughput of the small query set decreases 61% when the workloads are placed in the same database instance; decreases another 45% when the workloads are placed in the same buffer pool, and decreases another 7% when the workloads access the same tables.

Experimental Scenario	Type1 (q/s)	Type2 (q/s)	Type3 (q/s)	Type4 (q/s)
Scenario 1	30.54	24.92	23.99	24.05
Scenario 2	11.81	8.80	7.77	7.93
Scenario 3	6.48	5.96	5.85	5.83
Scenario 4	6.00	4.15	3.88	3.93

Table 4: Average throughput (Q21, "busy" period)

Experimental Scenario	Type1 (q/s)	Type2 (q/s)	Type3 (q/s)	Type4 (q/s)
Scenario 1	30.54	28.62	25.80	26.23
Scenario 2	11.81	10.70	8.95	9.16
Scenario 3	6.48	6.20	6.19	6.18
Scenario 4	6.00	4.97	4.66	4.67

Table 5: Average throughput (Q22, "busy" period)

Tables 4 and 5 also show that the throughput for the small query set is worse when the large query is decomposed (Type3 and Type4) than when the large query is not decomposed (Type2). This degradation in performance can be explained by the additional I/O overhead incurred by our approach to write temporary intermediate database tables. The overhead itself depends on how many intermediate results are written and it is unavoidable due to the fact that Query Disassembler is implemented outside the DBMS engine.

We see that, in the cases where the large query and the other queries are in one database and also share the same buffer pool, our approach works fine. From tables 4 and 5, we can see that the overall average throughput for the OLTP-like workload is only slightly lower when the large query is decomposed than when the large query is not decomposed. However, such a minor decease is implemented under a situation that our current approach brings an unavoidable big overhead (more on the overhead and the possible ways to reduce it are discussed in Section 6.1). Besides, if we only focus on the sampling interval between the 4^{th} and the 7^{th} sampling points in Figures 25 and 27 (assuming that the overhead brought by our approach could be controlled to a minimum level), it actually shows that the Type 3 and Type 4 throughputs are higher than the Type

2 throughput. This observation verifies that at any given time, a smaller query will likely hold fewer resources than a large query.

We also note that the throughput for Type 4, which includes the 1 minute delay between the executions of the decomposed query parts, is better than the case where the delay is not introduced. This delay means that by running a large query as a series of smaller queries, all resources that are occupied by the large query can be released between queries in the series and so are available for other queries and can be used to boost their performance for a period of time.

Our approach does not help in situations like scenario 1 (separate databases) or scenario 2 (one database, separate buffer pools). This, however, is expected. Carey et al. [13] point out that "whether the system bottleneck is the CPU or the disk, it is essential that priority scheduling on the critical resource be used in conjunction with a priority-based buffer management algorithm". In scenario 1 and scenario 2, the large query does not compete for memory (buffer pool) with other queries. Therefore, our approach will not make much difference in these two cases, even when there is no overhead involved.

The most surprising observation in our experiments comes from decomposing Q21 (Figures 20, 22, 24, and 26). In these cases, since by our decomposition algorithm, both Q21 and Q22 are broken into two smaller units, we expect that when a one-minute pause is applied between executing the two smaller units for Q21, the Type 4 curve shape in Figures 22, 22, 24, and 26 would be similar to that in Figure 21, 23, 25, and 27 correspondingly. To put in another way, we expect a throughput increase during this one-minute period before it decreases again. In Figures 22, 22, 24, and 26, however, the trend is not obvious.

The reason for this is subtle. In Section 5.1, we mentioned that our decomposition algorithm breaks Q21 into two "70% and 30%" smaller parts, and breaks Q22 into two "60% and 40%" smaller parts. In our experiments the actual execution for Q22 reflects this 60-40 division, but the execution of Q21 does not. In reality, the first smaller part of Q21 consumes most of the total execution time. How to take advantage of extra database compiler information to detect this type of circumstance in advance is slated for future work.

Chapter 6

Conclusion and Future Work

In this thesis 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. We also describe Query Disassembler, which is a prototype implementation of our approach with IBM DB2.

6.1 Conclusions

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 our 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. If we had the ability to save the 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. In our current approach, this type of overhead is inevitable.

The experiments 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 our approach. We could, however, shorten the overall delay by utilizing more advanced techniques of saving intermediate tables as discussed in previous paragraphs.

6.2 Future Work

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. Some of them are the following:

- Currently in our work, the small query set in our experiment workload contains just read-only queries. It is desirable to consider update-queries (e.g. INSERT, UPDATE, and DELETE) in the workload. These queries tend to create more resource contentions on a database system and may cause data inconsistency problems. We would like to examine the feasibility and/or the effectiveness of our approach under this situation.
- 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. In future work, we would like to investigate a better way to execute the decomposed segments, preferably within a database engine so internal query models, such as Query Graph Model in DB2, can be directly utilized.

- 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 can not 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, our 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.

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-201692.pdf.*

[2] P. Lyman, H. R. Varian. How Much Information 2003? *http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/.*

[3] Winter Corporation 2005 TopTen Award Winners, 2005, *http://www.wintercorp.com/VLDB/2005_TopTen_Survey/TopTenWinners_2005.asp.*

[4] S. Chaudhuri, U. Dayal. An Overview of Data Warehousing and OLAP Technology, *ACM SIGMOD Record 26(1)*, March 1997, pp. 65- 74.

[5] Transaction Processing Performance Council. *http://www.tpc.org.*

[6] J. Bloomberg. Why service-oriented management? *http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci929186,00.html.*

[7] 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.

[8] 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.

[9] P. J. Haas, J. M. Hellerstein. Ripple Joins for Online Aggregation, *Proc. of the 1999 ACM SIGMOD Int. Conf. on Management of Data,* Philadelphia, U.S.A, June 1999, pp. 287 – 298.

[10] J. M. Hellerstein, P. J. Hass, H. J. Wang. Online Aggregation, *Proc. of the 1997 ACM SIGMOD Int. Conf. on Management of Data,* Tucson, U.S.A, June 1997, pp. 171 – 182.

[11] J. Goldstein, P. Larson, Optimizing Queries Using Materialized Views: A Practical, Scalable Solution, *Proc. of the 2001 ACM SIGMOD Int. Conf. on Management of Data,* Santa Barbara, USA, June 2000, pp. 331 -342.

[12] 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.

[13] 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.

[14] K. P. Brown, M. Mehta, M. J. Carey, M. Livny. Towards Automated Performance Tuning for Complex Workloads, *Proc. of the 20th Int. Conf. on Very Large Data Bases*, Santiago, Chile, September 1994, pp. 72 – 84.

[15] 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.

[16] 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.

[17] IBM DB2 Query Patroller Guide: Installation, Administration, and Usage, IBM online documentation,

ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/en_US/db2dwe81.pdf

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

[19] S. Parekh, K. Rose, J. Hellerstein, S. Lightstone, M. Hurras, V. Chang, Managing the Performance Impact of Administrative Utilities, *IBM Research Report RC22864,* August 2003.

[20] 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.

[21] 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.

[22] IBM DB2 Visual Explain Tutorial, IBM online documentation, *ftp://ftp.software.ibm.com/ps/products/db2/info/vr8/pdf/letter/nlv/db2tvb80.pdf*

[23] IBM DB2 SQL Reference Volume 1, IBM online documentation, *ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/ko_KR/db2s1k81.pdf.*

[24] IBM DB2 Administration Guide: Performance, IBM online documentation, *ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/ja_JP/db2d3j81.pdf*.

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

[26] 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.

[27] H. Pirahesh, J. Hellerstein, W. Hasan. Extensible Rule-based Query Rewrite Optimization in Starburst, *Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data*, San Diego, USA, June 1992, pp. 39 – 48.

[28] L. Haas, J. Freytag, G. Lohman, H. Pirahesh. Extensible Query Processing in Starburst, *Proc. of the 1989 ACM SIGMOD Int. Conf. on Management of Data*, Portland, USA, June 1989, pp. 377 – 388.

[29] J.O. Kephart, D.M. Chess. The Vision of Autonomic Computing, *IEEE Computers, 36(1)*, 2003, pp. 41 – 52.

[30] "The Problem" – Autonomic Computing Overview (2005), *http://www.research.ibm.com/autonomic/overview/problem.html.*

[31] "Autonomic computing and IBM" (2002), *http://www-03.ibm.com/autonomic/pdfs/AC_BrochureFinal.pdf*.

[32] D. Kossamann. The State of the Art in Distributed Query Processing, *ACM Computing Surveys (CSUR) 32(4)*, 2000, pp 422 – 469.

[33] 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.

Glossary of Acronyms

CB-Segment	Cost Based Segment
DBA	Database Administrator
DBMS	Database Management System
GUI	Graphical User Interface
MPL	Multi Programming Level
MV	Materialized View
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
QEP	Query Execution Plan
QoS	Quality of Service
SKF	Skew Factor
SLA	Service Level Agreement
SLO	Service Level Objective
SOA	Service Oriented Architecture
SOM	Service Oriented Management
SQL	Structured Query Language
TPC-H	Transaction Processing Performance Council Benchmark H

Appendix A

TPC-H Benchmark

The TPC-H benchmark is a decision support benchmark developed by the Transaction Performance Council (TPC). It is used to evaluate the performance of decision support systems by virtue of executing a set of complex queries against a standard database (containing large volume of data) under controlled conditions in order to give answers to real-world business questions. The TPC-H benchmark contains a suite of business oriented ad-hoc queries. It is designed such that both the queries and the data reflect broad industry-wide relevance and a sufficient degree of ease of implementation.

A TPC-H database contains eight base tables. The relationships between these tables are illustrated in Figure 28. In Figure 28, the arrows point in the direction of one-to-many relationships between tables. The parentheses following each table name defines the prefix of the column names for that table. For example, the real column name for the name of a nation should be "N_NAME". The number below each table name represents the cardinality (number of rows) of the table. The SF in front of the number represents the scale factor used to obtain a chosen database size. Take SUPPLIERS table as an example, a SF value of 5 means that the actual SUPPLIERS table has 50,000 (5 * 10,000) rows inside. A TPC-H database with a SF value 1 (the TPC-H tables in the database all have a SF value 1) is approximately 1 GB large in size.

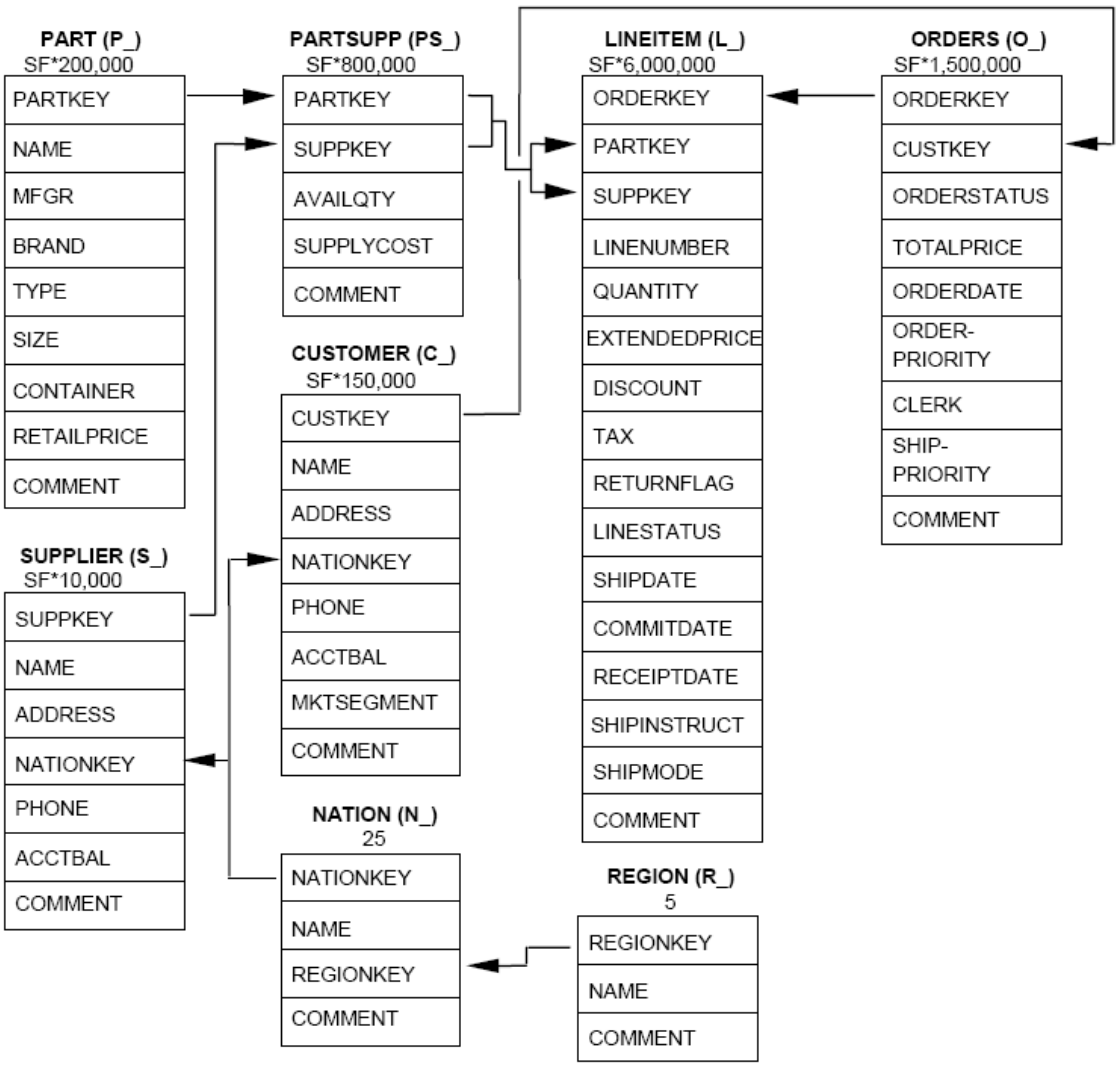

Figure 28: TPC-H schema [5]

TPC-H defines twenty-two decision support queries (Q1 to Q22). Tables 6 list the business questions for which Q21 and Q22 provide answers. For the business questions that Q1 to Q20 aim for, please see the official TPC-H specification [5].

Query #	Business Question Description
Q21	**Suppliers Who Kept Orders Waiting Query:** This query identifies certain suppliers who were not able to ship required parts in a timely manner.
Q22	**Global Sales Opportunity Query:** The Global Sales Opportunity Query identifies geographies where there are customers who may be likely to make a purchase.

Table 6: TPC-H queries

In TPC-H specification, Q1 to Q22 are all parameterized query templates. In order to generate executable decision support queries from these templates, parameter substitution by some randomly selected real value is required. The TPC-H specification defines the value range for each of the parameters involved and suggests a default value for it for the purpose of query validation. Figure 29 shows the SQL statement template for TPC-H Q22. The template for Q21 can be found in Figure 1 in Section 1.2:

```
1   select    cntrycode,
2             count(*) as numcust,
3             sum(c_acctbal) as totacctbal
4   from (
5             select    substring(c_phone from 1 for 2) as cntrycode,
6                       c_acctbal
7             from      customer
8             where     substring(c_phone from 1 for 2) in ('[I1]','[I2]',’[I3]','[I4]','[I5]','[I6]','[I7]')
9                       and c_acctbal > (
10                                  select    avg(c_acctbal)
11                                  from      customer
12                                  where     c_acctbal > 0.00
13                                            and substr (c_phone from 1 for 2) in
14  ('[I1]','[I2]','[I3]','[I4]','[I5]','[I6]','[I7]')          )
15                      and not exists (
16                                  select    *
17                                  from      orders
18                                  where     o_custkey = c_custkey       )
19            ) as custsale
20  group by            cntrycode
21  order by            cntrycode;
```

Figure 29: TPC-H Q22 statement (template)

Appendix B

Common QEP Operators

Table 7 gives the list of the common physical operators that can appear in a QEP. Simple descriptions of what these operators do are also provided.

Operator	Description
Table Scan (**TBSCAN**)	A TBSCAN operator retrieves the data of a database table by reading all the required data directly from the data pages.
Index Scan (**IXSCAN**)	An IXSCAN operator scans an index to produce a reduced stream of data.
Filter (**FILTER**)	A FILTER operator filters a stream of data based on the criteria supplied by the filter predicates.
Column Selection (**COLSEL**)	A COLSEL operator selects the data for designated columns from a stream of data.
Nested Loop Join (**NLJOIN**)	A NLJOIN operator joins two streams of data using the standard nested loop join algorithm.
Distinct/Unique (**UNIQUE**)	A UNIQUE operator eliminates duplicates from a stream of data.
Sort (**SORT**)	A SORT operator sorts a data stream in the order of one or more of its columns, optionally eliminating duplicate entries.
Hash Join (**HSJOIN**)	A HSJOIN operator joins two streams of data using the standard hash join algorithm.
Merge-Sort Join (**MSJOIN**)	A MSJOIN operator joins two streams of data using the standard merge-sort join algorithm. A merge-sort join is also called a *merge scan join* or a *sorted merge join*.
Union (**UNION**)	A UNION operator concatenates two data streams (having same data structure) and retrieves all data from both streams.
Intersect (**INTERSECT**)	A INTERSECT operator concatenates two data streams (having same data structure) and retrieves the data that are shared by both streams.
Except (**EXCEPT**)	A EXCEPT operator concatenates two data streams (having same data structure) and retrieves the data from the first data stream that is not contained in the second stream.
Aggregation/Group By (**GRPBY**)	A GRPBY operator groups data by common values of designated columns or functions. It is required to produce a group of values, or to evaluate set functions.

Table 7: Common QEP Operators

Appendix C

DB2 Explain Facility

IBM DB2 provides a facility called SQL Explain to allow a DBA to capture information about the access plan that is chosen by the DB2 optimizer [24]. The information captured includes: 1) operation sequence to process a query; 2) cost information; 3) predicates and selectivity estimates for each predicate; 4) statistics for all objects referenced in the SQL statement; and 5) values for the host variables, parameter markers, or special registers. The information can help a DBA understand how database tables and indexes are accessed for a submitted query and to evaluate the performance tuning strategies.

DB2 uses a suite of explain tables to store the captured explain data that can be accessed using the following methods:

- Use Visual Explain Tool [22] to view explain-snapshot information.
- Use the *db2exfmt* tool to display explain information in preformatted output.
- Use the *db2expln* and *dynexpln* tools to see the access plan information for static SQL statements or dynamic SQL statements that contains no parameter markers, respectively.
- Write one's own queries to access the explain tables.

Table 8 lists the relational tables that are provided by DB2 to store the explain information, which is used by our program to build up the QEP as well as its related cost information for a large query.

Table Name	Description
EXPLAIN_ARGUMENT	Contains information about the unique characteristics of each individual operator, if any.
EXPLAIN_INSTANCE	The main control table for all Explain information. Each row of data in the Explain tables is explicitly linked to one unique row in this table. Basic information about the source of the SQL statements being explained and environment information is kept in this table.
EXPLAIN_OBJECT	Identifies those data objects required by the access plan generated to satisfy the SQL statement.
EXPLAIN_OPERATOR	Contains all the operators needed by the SQL compiler to satisfy the SQL statement.
EXPLAIN_PREDICATE	Identifies the predicates that are applied by a specific operator.
EXPLAIN_STATEMENT	Contains the text of the SQL statement as it exists for the different levels of explain information. The original SQL statement as entered by the user is stored in this table with the version used by the optimizer to choose an access plan. When an explain snapshot is requested, additional explain information is recorded to describe the access plan selected by the SQL optimizer. This information is stored in the SNAPSHOT column of the EXPLAIN_STATEMENT table in the format required by Visual Explain. This format is not usable by other applications.
EXPLAIN_STREAM	Represents the input and output data streams between individual operators and data objects. The data objects themselves are represented in the EXPLAIN_OBJECT table. The operators involved in a data stream are represented in the EXPLAIN_OPERATOR table.

Table 8: Relational tables that store explain data [24]

All explain information as stored in the explain tables is organized around the concept of an ***explain instance***, which represents one invocation of the explain facility. Each *explain instance* can contain the explain information for multiple SQL statements, either static or dynamic. The information stored in the explain tables reflects the relationships between operators and objects in the access plan.

Other than the operation sequence in an access plan, the explain facility also captures cost information for each operator. The cost captured for an operator is an estimated cumulative cost, from the start of access plan execution up to and including the operator that includes:

- The total cost (in timerons).
- The number of page I/Os.
- The number of CPU instructions.
- The cost (in timerons) of fetching the first row, including any initial overhead required.
- The communication cost (in frames).

The unit of cost is ***timeron*** which is a DB2-specific relative cost unit. It does not directly link to any actual unit of measure, like response time or throughput, but gives a relative estimate of the resources required by the database manager. It is determined by the optimizer based on internal values such as statistics.

Appendix D

Maximum Error of Estimation in Experimental Results

In Section 5.3, we describe the experiment cases and the type of data to collect. There are 4 different experiment scenarios and within each case, there are 4 different data types. In this appendix, for simplicity we use S1T1, S1T2, S1T3, S1T4, S2T1, S2T2, S2T3, S2T4, S3T1, S3T2, S3T3, S3T4, S4T1, S4T2, S4T3, and S4T4 to name the 16 different types of throughput data to be collected, in which S means the experiment scenario and T means the data type. The number following S and T means the experiment scenario number and the data type number, respectively.

Equation 3 gives the formula of calculating the maximum error of estimation, meaning the maximum possible error between the sample mean and the population mean. In this equation, n is the sample number, σ is the standard deviation of the sample and z (α/2) is the z-value for a confidence level (1-α) 100%. In our experiment, n is equal to 10 and we uses 95% as the confidence level, corresponding to a z-value of 1.96.

$$E = z(\alpha/2) * \sigma/\sqrt{n}$$

Equation 3: Maximum error of estimate

Tables 9 and 10 list the calculated E-values for the 16 different types of data when the large query is TPC-H Q21, Q22 respectively. It can be seen from these tables that, with a confidence level of 95%, the maximum errors for the 16 types of throughput data that is collected by our experimental method are all less than 0.4 queries per second, corresponding to about less than 2% of the true values in most cases.

Data	Sample MEAN (q/s)	Sample STDEV (q/s)	E-Value (95% Conf.) (q/s)	Conf. Interval MEAN (95% Conf.) (q/s)
S1T1	30.54	0.30	0.18	(30.36, 30.72)
S1T2	27.58	0.56	0.34	(27.24, 27.92)
S1T3	27.05	0.47	0.29	(26.76, 27.34)
S1T4	27.14	0.40	0.25	(26.89, 27.39)
S2T1	11.81	0.26	0.16	(11.65, 11.97)
S2T2	10.27	0.30	0.19	(10.08, 10.46)
S2T3	9.55	0.28	0.17	(9.38, 9.72)
S2T4	9.82	0.29	0.18	(9.64, 10)
S3T1	6.48	0.41	0.25	(6.23, 6.73)
S3T2	6.22	0.36	0.22	(6, 6.44)
S3T3	6.14	0.33	0.21	(5.93, 6.35)
S3T4	6.12	0.38	0.24	(5.88, 6.36)
S4T1	6.00	0.18	0.11	(5.89, 6.11)
S4T2	5.06	0.13	0.08	(4.98, 5.14)
S4T3	4.90	0.13	0.08	(4.82, 4.98)
S4T4	4.92	0.15	0.09	(4.83, 5.01)

Table 9: E-Value for collected throughput data (Q21)

Data	Sample MEAN (q/s)	Sample STDEV (q/s)	E-Value (95% Conf.) (q/s)	Conf. Interval MEAN (95% Conf.) (q/s)
S1T1	30.54	0.30	0.18	(30.36, 30.72)
S1T2	29.55	0.47	0.29	(29.26, 29.84)
S1T3	28.03	0.60	0.37	(27.66, 28.4)
S1T4	28.44	0.54	0.34	(28.1, 28.78)
S2T1	11.81	0.26	0.16	(11.65, 11.97)
S2T2	11.20	0.18	0.11	(11.09, 11.31)
S2T3	10.32	0.20	0.12	(10.2, 10.44)
S2T4	10.48	0.17	0.11	(10.37, 10.59)
S3T1	6.48	0.41	0.25	(6.23, 6.73)
S3T2	6.31	0.37	0.23	(6.08, 6.54)
S3T3	6.30	0.38	0.23	(6.07, 6.53)
S3T4	6.33	0.37	0.23	(6.1, 6.56)
S4T1	6.00	0.18	0.11	(5.89, 6.11)
S4T2	5.45	0.12	0.07	(5.38, 5.52)
S4T3	5.30	0.12	0.07	(5.23, 5.37)
S4T4	5.29	0.10	0.06	(5.23, 5.35)

Table 10: E-Value for collected throughput data (Q22)

Appendix E

Small Query Set

As part of our experimental workload, we run a small query set consisting of eight queries which access the TPC-H tables. The templates for these queries are shown below. Within each template, a question mark (?) represents a parameter that is substituted by a randomly selected real value as defined below.

Query 1:

Template	select count(*) from region
Param. Values	None

Query 2:

Template	select r.r_name, count(n.n_nationkey) from region as r, nation as n where n.n_regionkey = r.r_regionkey and r.r_name = [?] group by r.r_name
Param. Values	{“ AFRICA”, “AMERICA”, “ASIA”, “EUROPE”, “MIDDLEEAST”}

Query 3:

Template	select n.n_name, count(s.s_suppkey) from supplier as s, nation as n where n.n_nationkey = s.s_nationkey and n.n_name = [?] group by n.n_name
Param. Values	{“ ALGERIA”, “BRAZIL”, “EGYPT”, “IRAN”, “MOROCCO”, “UNITED KINGDOM”}

Query 4:

Template	select c.c_mktsegment, count(c.c_custkey) from customer as c, nation as n where n.n_nationkey = c.c_nationkey and n_name = [?] and 100000 < c.c_custkey and c.c_custkey < 200000 group by c.c_mktsegment
Param. Values	{“ ALGERIA”, “BRAZIL”, “EGYPT”, “IRAN”, “MOROCCO”, “UNITED KINGDOM”}

Query 5:

Template	select p.p_container, count(*) from part as p where p.p_brand = [?] and 100000 < p.p_partkey and p.p_partkey < 200000 group by p.p_container
Param. Values	{“ Brand#11”, “Brand#22”, “Brand#33”, “Brand#44”, “Brand#55}

Query 6:

Template	select sum(ps.ps_availqty), sum(ps.ps_supplycost) from partsupp as ps where ps.ps_partkey = [?]
Param. Values	{1000, 2000, 3000, 4000, 5000, 6000}

Query 7:

Template	select o.o_orderstatus, count(*) from orders as o where [?] < o.o_orderkey and o.o_orderkey < [?] + 100000 and o.o_orderpriority = '1-URGENT' group by o.o_orderstatus
Param. Values	{100000, 200000, 300000, 400000, 500000, 600000}

Query 8:

Template	select l.l_orderkey, count(l.l_linenumber) from lineitem as l, orders as o where l.l_orderkey = o.o_orderkey and [?] < o.o_orderkey and o.o_orderkey < [?] + 100000 and o.o_orderpriority = '1-URGENT' group by l.l_orderkey
Param. Values	{100000, 200000, 300000, 400000, 500000, 600000}

Appendix F

QEPs of TPC-H Q21 and Q22 from DB2's Explain Utility

Figure 30 shows the QEP structure of TPC-H Q22 that is returned by DB2's Explain Utility. This structure is very similar to that as shown in Figure 18 in Section 5.1, which is the QEP structure of Q22 that is returned by Query Disaasembler, except that the cost presentation in the Query Disassembler's structure is more versatile (e.g. relative cost, accumulative cost, and etc.), which makes it a better candidate for the decomposition algorithm.

Similarly, Figure 31 shows the QEP structure of TPC-H Q21 from DB2's point of view. But unlike the case of Q22, we do not show its Query Disassembler structure in Section 5.1 because of its overly big size.

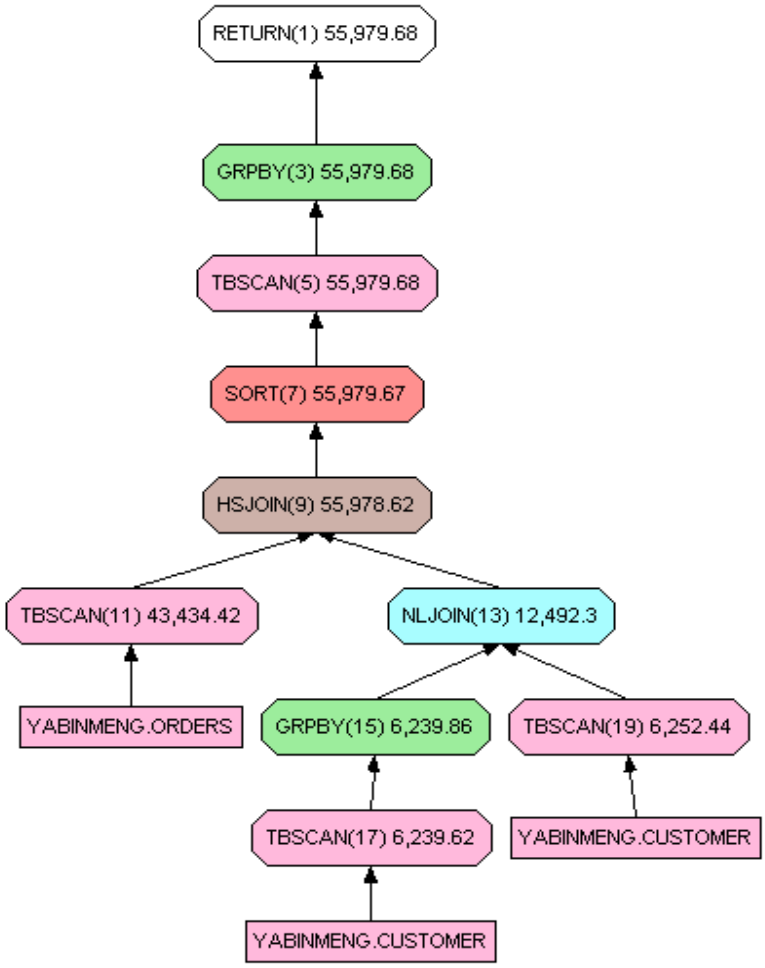

Figure 30 : QEP of TPC-H Q22 by DB2 Explain Utility

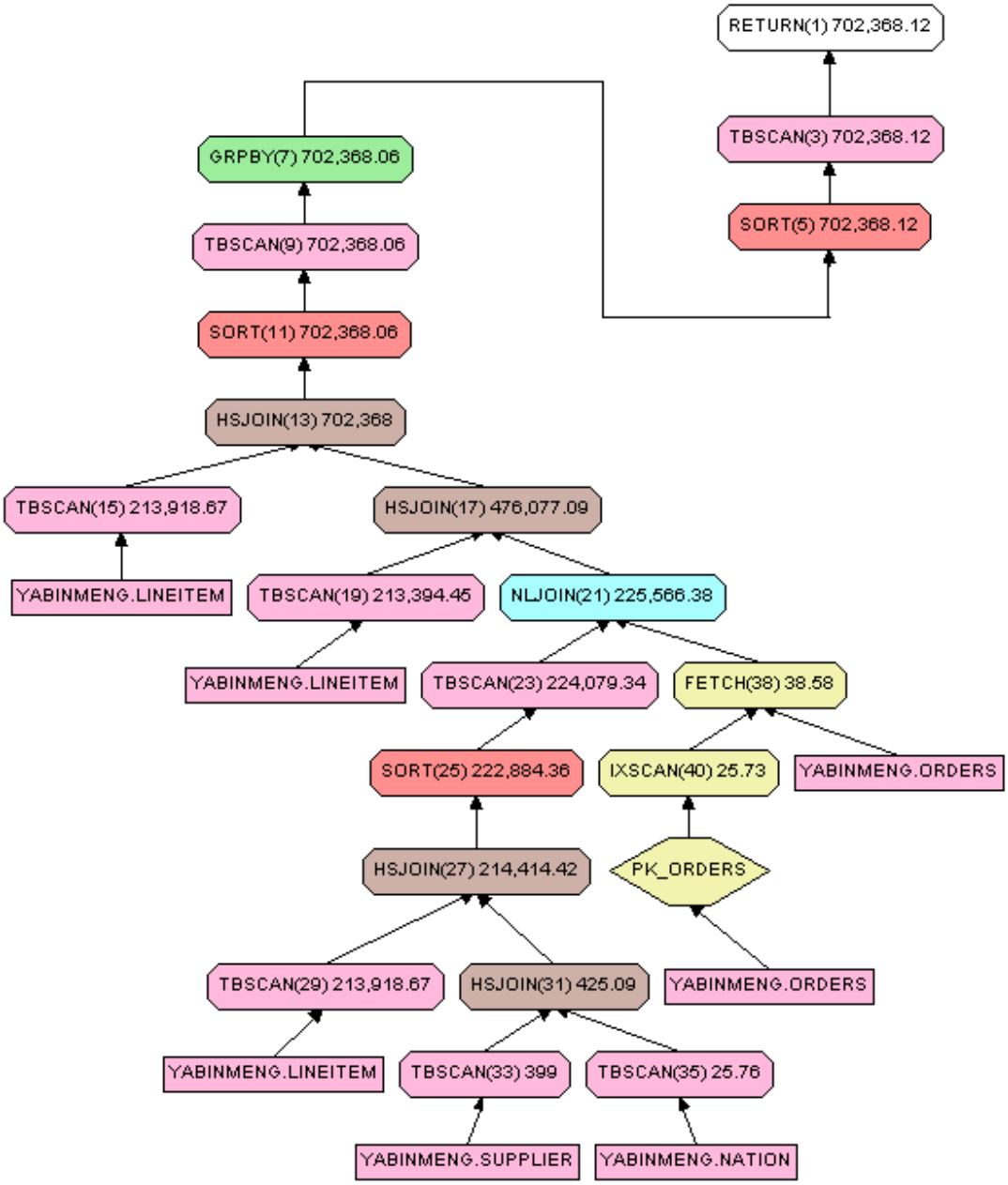

Figure 31 : QEP of TPC-H Q21 by DB2 Explain Utility