Managing Long-Running Queries Using A Query Progress Indicator

By

Mastoureh Hassannezhad

A thesis submitted to the Graduate program in
School of Computing
in conformity with the requirements for the
Degree of Master of Science

Queen's University
Kingston, Ontario, Canada
December, 2014

Copyright © Mastoureh Hassannezhad, 2014

Abstract

Database management systems (DBMSs) are the essential tools to monitor and manage data and transactions in databases. Normally a database receives hundreds of requests, called workloads so dynamic workload management is an important component of an autonomic DBMS solution. Controlling concurrently running queries in DBMSs is one form of workload management which is called execution control and it aims to guarantee desired performance levels for queries with varying priorities. One of the challenges in making such control-decisions is determining how much more processing time the query requires prior to completion. Query progress indicators attempt to estimate the progress of running queries. This offers a means to identify the long-running queries at early stages, potentially before other important workloads are negatively impacted, and thereby, assists in making a better decision to select the appropriate execution control mechanism for handling long-running queries.

Our work illustrates a strategy and defines policies for making decisions based on a simple progress indicator that we implement in PostgreSQL, to select the time and type of execution control action to apply to recover from a declining workload performance in a DBMS. Based on the progress of the long running OLAP queries, they are either throttled or killed at different progress points to maximize system efficiency.

We define a utility measure to indicate throughput recovery of the important workloads with respect to the increase in the total workload execution time. Our experiments show that throttling combined with a progress indicator results in faster and better system recovery.

Acknowledgements

It would not have been possible to finish this thesis without the help and support of the kind people around me.

I cannot find words to express my sincere gratitude to my supervisors, Prof Patrick Martin, and Dr. Farhana Zulkernine for their valuable advice, insight, support and encouragement in the preparation of this thesis. For me they are with no doubt among the most understanding and supportive supervisors.

I would like to thank Wendy Powley particularly whose supports, guides and encouragement helped me finishing my thesis. It is a pleasure to thanks my colleagues and friends specially my dearest friends Sima Soltani and Parisa Abedi who patiently listened to me and my road blocks on many occasions and supported me throughout preparing this work.

I would like to acknowledge the financial, academic and technical support of the School of Computing and School of Graduate Studies at Queen's University for providing me the opportunity to pursue my master's degree.

Lastly, I would like to thank my parents, for their love and support and believing in me and encouraging me to pursue this degree. My special thanks to my brothers, especially Hamoun who have given me their continual support and love in several aspects to accomplish this work. Thank you.

Table of Contents

Abstract i
Acknowledgements iii
Table of Contents iv
List of Figures vi
List of Tables.. vii
Chapter 1: Introduction 1
1.1 Research Objective.. 4
1.2 Contributions... 6
1.3 Thesis Outline 7
Chapter 2: Background... 8
2.1 Workload Management 8
2.2 Execution control techniques... 11
2.3 Query Progress Indicator 15
2.4 Throttling... 20
Chapter 3: Implementation of a Progress Indicator 23
PostgreSQL 24
3.1 PostgreSQL statistic collector.. 24
3.2 Experiments and Validations: 26
3.3 Test results for Queries Q_1 and Q_4 28
3.4 Conclusion 30

Chapter 4: Execution Control with a Progress Indicator.. 31
4.1 Execution Control Actions.. 35
4.2 Experimental Scenarios.. 37
4.3 Policies:... 39
4.4 Integrating Progress Indicator with Execution control mechanism: 40
4.5 Experiments and Validation:.. 41
4.6 Discussion and Critical Analysis:... 56
Chapter 5: Conclusion and Future Work... 65
References 68
Appendix A 72
Appendix B 75

List of Figures

Figure 1: Workload Management System controls. 3

Figure 2: Estimation of percentage completed over time (test for Q_1) 28

Figure 3: Estimated completed percentage over time (test for Q_4) 29

Figure 4: Throughput recovery after applying ECA with the usage of PI 57

Figure 5: Throughput recovery percentage with the usage of PI 58

Figure 6: Execution time increase percentage with the usage of PI 59

Figure 7: Execution time as a result of killing/throttling with the usage of PI........ 60

Figure 8: Utility of system 63

List of Tables

Table 1: Abbreviations used in Description of Experimental Study........................ 34

Table 2: Applying throttling with different pauses on OLAP 37

Table 3: Scenarios and actions 39

Table 4: Observations with no execution control 45

Table 5: Taking killing action on W_{OLAP} without PI 49

Table 6: Taking Throttling action without PI observation..................................... 50

Table 7: Taking Killing action while using PI.. 54

Table 8: Taking Throttling Action while using PI 55

Chapter 1: Introduction

In today's digital world of big data, we are continuously searching for innovative ways to manage and serve requests for data more efficiently and faster than before. Database management systems (DBMS) are the essential tools to handle, monitor and control data in databases. An identifiable set of requests to a database, say from an application or set of clients, is called a workload and techniques and algorithms have been proposed to effectively handle ad-hoc and unpredictable workloads [13].

There are mainly two distinct types of workloads. First, those which are characterized by many short transactions like data lookups or updates. This type of workload normally has high priorities and needs to be served quickly. They are well known as Online Transaction Processing (OLTP) workloads [18], [24]. The other type of workloads, called Online Analytical Processing (OLAP), represents complex and long-running queries that are often read-only. They tend to read a huge amount of data and make complex relations between numerous tables to handle complicated requests.

A system can receive a combination of both OLTP/OLAP workloads. The number and mix of workloads a database needs to handle varies depending on the nature of the associated applications which store and retrieve data from the database. With the fact that applications need to be accessible and operational 24/7, making any kind of restrictions for OLTP/OLAP workloads (such as the running OLAP queries at evenings or

midnights when the load on the database is less, or forcing a limitation on the number of OLAP/OLTP running at the same time) is not feasible [25].

Considering the variability of workloads, the ubiquitous nature of data access by an increasing user group and the limited number of physical resources that are available during any time period (such as CPU and memory), workload management plays an essential role in DBMSs. Users of short OLTP queries want quick responses and are, therefore, often considered to have higher priority than the long running OLAP queries. The higher priority workloads need to have access to the resources even if the required shared resources are being used by other lower priority workloads [2]. So one of the main tasks of a DBMS is workload management, which is to recognize, schedule and prioritize the execution of workloads.

Different techniques and policies have been proposed for managing workloads at various levels during their life-time in databases. A workload management process is considered to involve three typical controls namely *admission control*, *scheduling control*, and *execution control* [21] as shown in Figure 1.

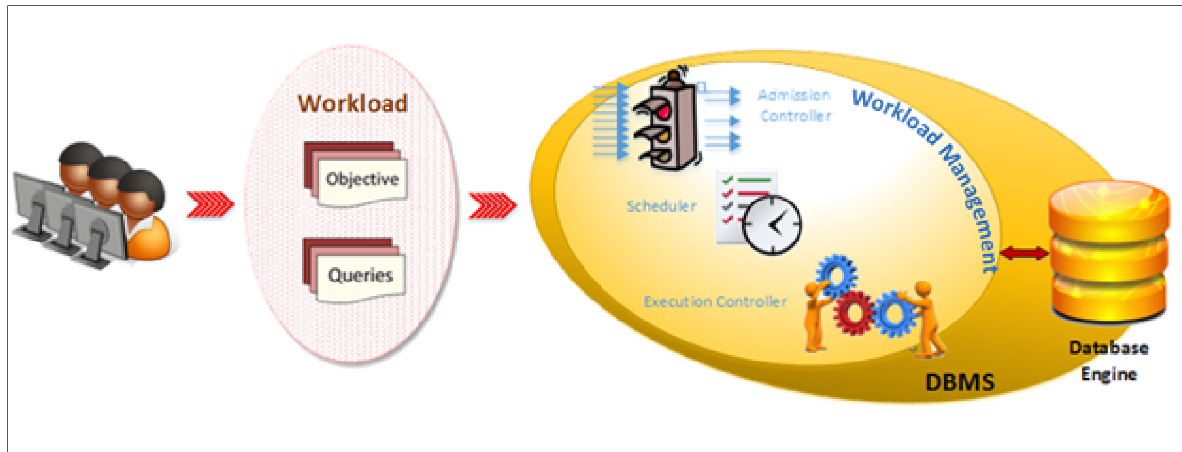

Figure 1: Workload Management System controls.

A workload management policy guides and controls the techniques to be used at these three levels [21]. In *admission control*, a DBMS considers different elements such as estimated cost of a query and resource demands to make a decision whether to allow a query to run or not. Thus it can stop a busy system from getting overloaded.

Each request that gets admitted and passes this step needs to go through an execution order. There are different techniques for scheduling admitted queries to be sent to database engine for execution [21], [22]. Thus *scheduling control* tries to schedule as many admitted queries as possible for execution while maintaining the system in a stable and optimal state. It also takes into consideration that the queries vary by their performance objectives and priorities.

While admission control and scheduling control deal with queries that have not yet started their execution, there is another control which cares about the workloads that are currently being executed in the system. Although the queries with high risk

have been recognized in admission control and scheduling control and are prevented from getting executed, some queries end up having unexpected resource demands and during execution they show that their estimated cost and estimated time were not accurate. These unexpected queries deteriorate the overall system performance. Sometime while running long-running OLAP queries with low priority, some high priority short OLTP queries pass admission and scheduling control and want to use shared resources which are being used by the OLAP queries. In this scenario, it is *execution control's* responsibility to monitor the running queries and make the best decision to slow down the lower priority queries and free up shared system resources.

So in particular, the main focus of DBMS workload management is to manage and execute workloads with various resource demands in a way to achieve the required performance objectives in a complex environment and prevent problematic workloads from making negative impacts on system's performance.

1.1 Research Objective

The objective of our research is to apply proper execution control and investigate the feasibility of autonomic workload management. In the area of workload management, dealing with long-running queries where execution time ranges from seconds to hours in an unpredictable manner is one of the challenging problems. Recognizing errant long-running queries and taking appropriate actions can minimize the negative effects on workloads in general.

Having multiple workloads running on a machine in parallel and taking into consideration that these workloads share machine resources, our goal is to give higher priority to the more important workloads while minimizing the deleterious impact of long running queries on overall performance of the system.

Monitoring the execution time of the workloads and the throughput is one of the major responsibilities of a DBA in order to ensure that the execution times and throughputs of all the workloads meet their Service Level Objectives (SLOs) [22]. In this research we investigate the usefulness of a query *Progress Indicator (PI)* in execution control by employing a simple progress indicator. Query progress indicators are useful for the DBAs to observe the percentage of the work completed and make more informed decisions about taking necessary workload management actions for different workloads with different resource demands running concurrently on a system [7][8][20].

Various execution control actions have been proposed for workloads, including *Query Throttling*, *Query Suspend and Resume* and *Query Kill and Requeue*. *Query Throttling* periodically pauses a running query to slow down its execution [1], [25], [26]. *Query Suspend and Resume* terminates a query's execution, stores the necessary intermediate results and restarts the query from the suspended point at a later time [1]. *Query Kill and Requeue* terminates the executing query and re-submits it again to the system at a later point, which is widely used in workload management facilities of commercial databases [10], [15], [17]. The choice of the execution control action is dependent upon the progress and nature of the currently executing query.

The research hypothesis studied in this thesis is that the use of a query progress indicator can improve the effectiveness of decision-making for execution control of unpredictable complex queries. We create a simple Query Progress Indicator, integrate it with an existing execution control mechanism, and run experiments with various scenarios containing different execution control actions. We observe a considerable improvement in the effectiveness of execution control while tracking query progress.

We limit the scope of the execution control actions that we apply in our work to *Query Throttling* and *Query Kill and Requeue* for running workloads in a *PostgreSQL* database running in a client-server network. Based on the progress of a query, a query is either throttled or terminated to maximize system efficiency.

1.2 Contributions

The main contribution of our work is the experimental evaluation of the impact of integrating a progress indicator with execution control mechanisms on the management of long-running queries in PostgreSQL. We illustrate a strategy, and thereby, define policies for making decisions regarding the execution control action needed to recover from declining performance in a DBMS. Our experiments show that throttling combined with a progress indicator results in faster and better system recovery.

1.3 Thesis Outline

The rest of the dissertation presents the following. In chapter 2 we discuss workload management and the different execution control actions that are applicable for workload management. The throttling mechanism and existing progress indicators in the literature are discussed in chapter 2. Our simple progress indicator is described in chapter 3. Chapter 4 presents an experimental evaluation of our approach. Chapter 4 also includes how we add our simple progress indicator to work with the throttling mechanism. In chapter 5 we conclude our work and talk about the possible future work in this area.

Chapter 2: Background

Our work focuses on workload management in DBMS. Different controls and techniques have been proposed [21], [22] for workload management. We investigate the usefulness and applications of a Query Progress Indicator in our research. We discuss various workload management techniques and actions in the following section. Different Query Progress Indicators and the ongoing research in this area are presented in section 2.3.

2.1 Workload Management

A database workload is a set of requests that have some common characteristics such as the application that generates the requests, type of query, business priority, and performance objectives [5]. Current trends such as server consolidation [6], [12] and the growing use of applications for business analytics [9] are increasing the complexity of managing these workloads. Therefore, handling and controlling them while they are running simultaneously is one of the main objectivess of DBMSs.

The trend of server consolidation means that several types of requests are mixed and are concurrently present on a single data server. The requests can be any combination of OLTP and OLAP queries. OLTP queries are those which need only

milliseconds of CPU time and very small amounts of disk I/O, and they complete in sub-seconds. On the other hand OLAP queries need more resources to execute, are typically more complicated, and deal with a large amount of data. Some OLAP queries can run for a long time and are known as long-running queries. Long-running queries are a known problem in all commercial database products [21].

Workload management policies help users (customers) meet their performance objectives. A very common and inevitable objective is returning the results in an acceptable time. Users do not expect to wait so long for their short, high priority requests like adding new records or updating their data. At the same time they expect a fast response for their requests for large reports which can require data from multiple tables and implement complicated join conditions. So the main goal of a workload management system is to implement appropriate policies to achieve these objectives.

Typically, workload management policies are defined by a DBA and can act at three different points. They involve admission control, scheduling control and execution control components as shown in Figure 1.

These components manage queries and workloads during their life-time from the moment they enter the system to the time they complete their execution in the system. They control which queries are admitted into the database system, the structure of the queue, and the order of execution of the admitted queries, and finally, what actions must be taken for running queries to meet the performance objectives.

2.1.1 Admission control

Admission control determines whether or not a query should be accepted into the system. Different policies can be defined based on the objectives of the application, e.g. the number of queries running concurrently, the number of concurrent users, or the cost of a query. Different actions are taken by the admission control such as *warn*, which accepts the query but warns about the risk; *hold*, which holds a query for a later time to get investigated, and *reject,* which rejects the query. Policies are also defined in the system for high-priority queries to bypass this control component and for the scheduling component to start executing the query immediately [21].

2.1.2 Scheduling control

The main goal of scheduling control is to avoid DBMS overload. This component defines different queues for queries so that the database engine can run them in a specific order. Different queues can be defined based on the characteristics of the query, such as:

- *Priority*: Orders queries based on their priorities.
- *Size*: Put queries in a stack ordered by their size.
- *One*: One FIFO queue for all admitted queries.
- *None*: All queries start immediately without any need to get into a queue [21].

2.1.3 Execution control

While admission control and scheduling control focus on applying policies on queries before execution, execution control focuses on controlling workloads during their runtime. The admission control system makes an estimate about the run time and resource requirements for each admitted workload when they enter the system. Some queries deviate from these estimated measures during the execution time because their execution time and their resource demands differ from the time they enter the system. For example some queries are expected to be short but prove otherwise during runtime and they are the ones which raise problems for the other queries running simultaneously on the system. They are identified as unpredictable queries or queries that need more attention and require the DBA to apply execution control techniques for handling concurrent workloads at runtime to meet performance objectives as discussed in the following sections.

2.2 Execution control techniques

Execution control tries to lessen the detrimental impact of unpredictable and long-running queries (which were not supposed to take a long time to execute) on the total performance of the system. Execution control looks at runtime statistics to catch unpredictable queries. There are various techniques for execution control, which consider both query cost estimates and runtime monitoring to meet performance goals. Execution control policies are defined for decision making at different levels of performance, which are measured based on a number of monitoring metrics. Based on

an observed condition and execution control policy, an appropriate action is fired to manage a workload. The monitoring metrics are also used to compare and evaluate the effects of the execution policies.

Some conditions to evaluate are CPU time above a threshold, or elapsed time exceeding the predicted time by absolute or relative amount. Another metric to measure the performance is throughput, which is the number of queries completed in a unit time.

Execution control actions include [13][25]:

- *None*: let the query run to completion.
- *Warn*: print a message to log; query continues unless the DBA takes an action.
- *Reprioritize*: change the priority of the query.
- *Stop*: stop processing query; return results so far.
- *Kill*: Abort the query execution and return an error.
- *Kill & Requeue*: Abort the query; put it in a scheduling queue to start over.
- *Suspend & Resume*: Stop processing query; put saved state in scheduling queue.

We discuss the implementations proposed for some of the above actions in the following sections.

2.2.1 Suspend & Resume

Chandramouli et al. [1] propose a technique to deal with unpredictable queries which have a negative impact on performance. They propose a query *suspend & resume* approach as a query execution control technique to handle long-running queries that are resource-intensive. It suspends a long-running low-priority query in a system when a high-priority query is being executed and resumes the suspended query when the high-priority query finishes its work. Two new phases are defined in the lifecycle of the query named suspend and resume.

When a query is sent to the database engine, the database query optimizer chooses an execution plan for a query. Until the query receives suspension request from database engine it stays in execution phase. After receiving suspension request, the query enters its suspension phase. A specific structure named *SuspendedQuery* is produced in this phase, which is necessary to encapsulate all the data needed to resume the query later. This structure may be written on disk. The suspended query enters the resume phase after receiving resume request. The structure is read back into memory and the database engine starts executing it from the suspended point.

2.2.2 Reprioritize, kill and re-submit

Krompass et al. [22] propose execution control components for a dynamic workload management system for Business Intelligence (BI) workloads. The decision-making for their execution control employs fuzzy logic. They use *reprioritize, kill* and *resubmit after kill* as the appropriate actions for execution control. In their approach

they observe different metrics such as priority, number of cancellations, operator progress, resource contention, resource progress, and database time. Based on these metrics, they come up with three actions to control the workloads on system in order to reduce the impact of the problematic queries. In the reprioritizing action, the controller increases the priority of the queries and then redistributes the resources again among the queries. The controller kills a running query and right after that frees the resources used by the killed query. All the generated results are lost. The killed query can be resubmitted to the database later. The query is en-queued again for resubmission based on the chosen scheduling control policies.

2.2.3 Throttling

Using the work of Parekh et al. [23], Powley et al.[26] propose an automatic approach to throttle long-running queries in a database system. Query throttling is a workload execution control technique that slows down the execution speed of long-running queries to release shared resources for the high priority queries. Two different methods for throttling are developed in their approach, called constant throttling and interrupt throttling. With constant throttling many short pauses are evenly and consistently distributed throughout a query's run time to slow down the query's execution speed. The pause-length is a parameter which is defined by the user and the number of pauses is determined automatically by an autonomic controller. The interrupt throttling method imposes just one long pause during the execution time of the long-running query.

2.3 Query Progress Indicator

Progress Indicators (PIs) are well-known in software systems to present the degree of completion for a running task. Typically a progress indicator shows users how much work has been done and how much of it remains. It presents these estimates in different forms such as percentage of the work completed and the remaining work, or the elapsed time and the estimated completion time. Progress indicators continuously update their information based on the degree of task completion.

Dealing with big data stored in large databases and fetching desired results is one of the challenging ongoing problems. Complex Business Intelligence Systems need fast interaction with users to present the appropriate services to them, which means prioritizing and managing incoming requests. Knowledge of the degree of completion of queries can help DBAs to come up with the best decision in order to keep the system performing optimally. So it is important for both the users and the DBAs to have the information about the progress of query executions.

Some of the DBMSs provide trivial progress indicators in a way that they break down the query execution plan into steps, and then they monitor the execution progress in each step and the completion status of each step at any specific time [14]. This approach may not work properly in all cases as some long-running queries have larger execution steps. While the program tells how many steps are left, it does not give accurate information about how long the steps will take to complete. On the other hand

a long-running query may have just a few steps. So the estimated runtime for the few steps of a long-running query would not be accurate.

Almost all of commercial database vendors provide query optimizers [10][15][17]. Query optimizers calculate the cost of different query execution plans in order to compare the plans and choose the most efficient one. They also give information about the size of queries at different levels of the query execution plan. A way to seek information about the progress of a query is to use the optimizer's estimate of query execution time. Providing information about the query execution time based on the optimizer's estimate is easy. If an optimizer's estimate of query execution time is t seconds, and the query has run for $t^{'}$ seconds, then one can calculate the remaining query execution time would be $t\text{-}t^{'}$ seconds. While such a trivial progress indicator is better than nothing, it is likely to be inaccurate. Lue [7] lists two main reasons for this inaccuracy:

1) Query cost estimates of optimizers typically have errors. Additionally optimizers are designed to choose good plans over bad ones. Correct relative estimates are needed for choosing the best query execution plan but not necessarily correct actual cost estimates. The job of predicting the actual query execution time is more challenging than choosing good plans over bad ones.

2) We are investigating systems with high interactions with users, so the system load may vary significantly from time to time. So even if an optimizer's estimate for a given query is precise for an unloaded system, it may be different in a loaded system.

Desirable properties of progress indicators include the following [19]:

- Accuracy: The estimated completed work and remaining work at any point should be close to the actual completed and remaining work.
- Fine granularity: An estimator should be able to provide estimations at fine granularity over the duration of the query execution time.
- Low overhead: For adding the capability of estimating query progress, a reasonable overhead on the actual execution of the query is acceptable, although the overhead should be low.
- Leveraging feedback from execution: During query execution time, some intermediate results become available and the estimator should be able to use them.
- Monotonicity: As a query is in general monotonic in its execution progress, the estimated progress also needs to be monotonic over the duration of its execution.

In 2004, progress indicators for SQL queries were first published separately in the same conference with two distinct approaches by two different research groups. Chaudhuri et al. [19] and Lue et al. [7] proposed and implemented query progress indicators for two different DBMSs. Later variants [3], [4], [6], [16] of these two approaches were proposed by the authors or other researchers to explore different

issues such as reducing cardinality, working on concurrent queries and investigating the interactions between queries to improve query estimation.

Chaudhuri [19] proposed a query progress indicator named MSRPI (Microsoft Research Progress Indicator). In MSRPI, an execution plan is used as the root of progress estimation. An execution plan is a tree of physical operations chosen by the query optimizer. MSRPI divides the query execution plan into a set of pipelines and calculates the percentage of GetNext() calls finished to estimate the execution time of running queries.

Each operator in a query execution plan uses a standard interface for query processing (including Open(), Close() and GetNext()). They propose their model of work done as the total number of GetNext() calls issued throughout the duration of executing a query.

MSRPI models a query's completion percentages as the fraction of the total number of GetNext() calls that have finished. Suppose that an execution plan has n operators and N_i is the total number of tuples output by operator Op_i (which indicates the number of GetNext() calls made by that operator). If we consider K_i to be the total number of tuples processed so far by the operator Op_i, then the percent of a query executed so far would be:

$$percent = \frac{\sum_{i=1}^{n} K_i}{\sum_{i=1}^{n} N_i}$$

The progress indicator proposed by Lue [7], called *wiscPI*, uses an execution plan chosen by the query optimizer similar to MSRPI. However, WiscPI estimates the remaining query execution time by implementing a model based on the number of bytes processed in each pipeline (U_i) and tracks of the total number of bytes that have not yet been processed. In a simple word the remaining work for a query is the sum of all the bytes that have not been processed (S_i).

$$RT = \frac{\sum_{i=1}^{p} U_i}{\sum_{i=1}^{p} S_i}$$

Li [11] implements both MSRPI and WiscPI in the same commercial RDBMS and investigates their performance. The work concludes that a wrong uniform future speed assumption in both approaches leads to estimation errors. The assumption is: “At any point in a query's execution, the time to process a unit of work is uniform throughout the remainder of its execution [11]" where a unit of work in MSRPI is one GetNext() call and a processed byte in WiscPI. Li proposes a new query progress indicator similar to the previous progress indicators but without the uniform speed assumption. The progress indicator designed and implemented by Li, called GSLPI, is a cost-based progress indicator. The basic idea of GSLPI is to separate a complex query execution plan into a set of speed independent pipelines where the boundaries of pipelines are defined by blocking operations. An operator is called a blocking operator if it does not produce

any output before it has processed all tuples in at least one of its inputs (e.g. Hash join, Group by, Sort).

A speed-independent pipeline is therefore a group of operators that execute concurrently and process tuples according to the execution plan. For each pipeline, the speed of processing for the remaining work is estimated by using the wall-clock pipeline cost. *Wall-clock pipeline cost* is defined as the maximum amount of non-overlapping CPU and I/O done by a pipeline during its execution time.

2.4 Throttling

In the throttling approach a workload is slowed down in order to release system resources for the other workloads which need them.

Powley et. al [25] implement throttling with two different methods called *Constant Throttling* and *Interrupt Throttling*. In constant throttling, the low-priority query is slowed down using frequent short pauses (from nanoseconds to seconds) at specific intervals. The intervals are calculated automatically based on the amount of throttling. The pause length is specified by the user.

Interrupt throttling, on the other hand, uses one long pause of a predefined length determined by the amount of throttling for the executing workload. In this method the pause-lengths are longer than those in constant throttling (from seconds to minutes). The long pauses are applied at different positions during a query execution period including at the start, middle, or at the end of the query execution period.

The throttling approach is examined by Powley et al. [26] in terms of its effectiveness by modifying PostgreSQL. For implementing these techniques, the authors use the interrupt checker routine in PostgreSQL. The interrupt checker incurs low overhead, and runs frequently and continuously during query execution time to check for interrupts. A macro named check_for_interrupts() is called at strategically located spots where it is normally safe to accept a *cancel* or *die* interrupt. If the throttle handler, which checks the performance of system, is turned on, then the interrupt checker routine gets notified and the throttling process starts. For the executing query a counter is increased each time the interrupt handler is called. A nano-sleep function is used by the throttling mechanism to pause the query. The throttle handler governs the amount, time, and the length of throttling for each query.

The results of their experiments show that slowing down or pausing the low-priority query by throttling improves the overall system performance. Slowing down such workloads does not require re-execution of a workload, or storage of intermediate results, and lets the work continue at a slower pace. They compare different types of throttling with different pause lengths and intervals in their experiments for OLTP + OLAP workloads and OLAP + OLAP workloads. The results show that for the high priority workloads (OLTP workloads), constant throttling provides substantially better performance as compared to interrupt throttling. So, in a system with mixed workloads and the need for better performance for the high priority workloads, constant throttling can achieve the desired system performance. Powley's work also shows that a shorter

pause length for the constant throttling technique has a more negative impact on the total execution time of OLAP queries compared to a longer pause length.

Considering the results of Powley's work, in order to lessen the detrimental impact of low-priority workloads on a system, we use the constant throttling technique in our experiments. Longer pause lengths as compared to shorter more frequent pauses, provide better results for the throttled workload in terms of OLAP execution time in the case of constant throttling. So we use a longer pause length in our work and set the necessary parameters for constant throttling accordingly.

Chapter 3: Implementation of a Progress Indicator

Today's database systems do not give enough information about how far a query execution has been completed and how much still remains. Having such information can help DBAs as well as end users or even applications to make a decision whether to terminate the query or to allow it to finish, change the query parameters or take other actions. As decision support applications typically contain long running tasks and queries executing concurrently, having a progress indicator to help to determine how much of work has been done so far is very helpful and valuable.

In the chapter 2, we discussed different techniques for developing progress indicators as proposed by researchers. One of the objectives of our work is to know if a progress indicator can enable better execution control. Powley's technique [25] is implemented on PostgreSQL database engine which is a powerful open source object-relational database system. So in order to extend the authors' work we decided to implement our progress indicator on PostgreSQL database engine. We run progress indicator to estimate OLAP queries progress in our work.

PostgreSQL

PostgreSQL is a powerful open source object-relational database system that has been active for development more than 15 years and has a good reputation for reliability, data integrity, and correctness [16]. One of its characteristics is its ability to run on all major operating systems such as Linux, UNIX, and Windows. It supports the ACID properties, which are Atomicity, Consistency, Isolation and Durability, of a reliable DBMS. It has native programming interfaces for C/C++, Java, .Net, Perl, Python, Ruby, Tcl, and ODBC (Object Database Council). [16]

Its open source feature gives us the opportunity to change the codes to have more control on what exactly we want to do for research purposes. For our research, we want to see the effectiveness of a progress indicator on execution control policies (throttling/killing) and throttling techniques (Powley's et al.[25]).

3.1 PostgreSQL statistic collector

We look for a way to implement a simple progress indicator for the PostgreSQL database system. The most useful measure to report to the end user is the total execution time required for the query to complete. However, since concurrent queries running together on the same database use shared resources, total execution time varies for each query under different situations and, therefore, the measure is subject to

uncertainty. So we decided to concentrate on the percentage of completion of work rather the required time to finish. In order to provide such information we need to focus on the number of rows that have been read so far by the database engine for the given query. By using “rows read” as an indicator of progress we have a good estimate of progress of a query. In order to make this approach applicable we need to know the exact number of rows that the query will read, and at any time by knowing the number of rows read so far we have a good estimation of the progress-percentage. One of the shortcomings of this simple approach is that we assume that we know the total number of tuples to be read for a query. This can be obtained by running the query beforehand and monitoring the total number of tuples read by the query. The other limitation of this approach is at any time we assume that only one query is running on the database which we talk about the reason in this section.

Examining the tools available in PostgreSQL, there is a subsystem that supports collection and reporting of information about the activities on the server [17]. The statistic collector tracks the activities over tables and indexes and gives information about each table and the total number of rows in each table [17]. Statistics are collected during the running of all queries and the stats are stored into tables in the database. To enable statistics collection, two parameters are set in the configuration file postgresql.conf namely *track_counts*, and *track_activities*.

Predefined views can be queried to show the collected statistics. One of the available views is *pg_stat_database* which contains one tuple per database. Statistics

collected include <database name, rows returned, rows fetched, rows updated, rows inserted, blocks read, blocks hits, etc>. Querying the tup_returned column from this view shows the total number of tuples that have been returned so far.

We assume that we know the total number of tuples read for a query. This can be obtained by running the query and obtaining the value via monitoring (tup_{total}). The "progress indicator" is implemented by repeatedly querying the pg_stat_database view to determine how many tuples have been returned at a specific point in time (tup_{read}). From this, we can calculate the progress of the query.

$$\text{progress (\%)} = tup_{read} / tup_{total} * 100$$

As the statistics shown in pg_stat_database view are database wide for each of the instances running on the DBMS, in order to use "tup_returned" metric for one query, only one query is run at a time on the database instance. So we use two separate database instances and run each type of query, OLTP and OLAP, on separate instances. As one of the assumptions in our work is to have interference between two workloads we need to make sure that by separating the databases we still have such interference. As the two workloads share the CPU and the IO subsystem, running the OLTP and OLAP workloads in parallel will still have an interference.

3.2 Experiments and Validations:

In this section we explain how we run the experiments to evaluate the performance of our progress indicator.

Experimental Setup: The experiments are performed on a server which has a 2.80 GHz processor, 3 GB main memory, and Ubuntu operating system with the PostgreSQL client application.

We use standard TPC-H Benchmark relations (Appendix A) [24]. The TPC-H benchmark is a decision support benchmark, developed by the Transaction Processing Performance Council. It's main purpose is to be used to evaluate the performance of decision support systems by executing queries under controlled conditions [24]. Each experiment is repeated three times for each of the queries.

We evaluated the performance of our progress indicator in the following way:

1) Before running queries, we execute the PostgreSQL statistics collection program for the selected queries as described in section 3.1.
2) We tested two standard queries of the benchmark TPC-H: Queries Q_1 and Q_4 (Appendix A). The reason we chose these queries is explained in section 3.3.
3) Before running each query we restart the computer to make sure that we have a cold buffer pool. (We also repeated the experiments with a warmed-up database and observed that the results are the same). A cold buffer pool refers to the buffer pool right after restart. The data cache is not loaded (cold) and requires physical reads to populate the cache. On the other hand warming up database means that we fill up the database buffer pool with data and index pages after starting the database.

3.3 Test results for Queries Q_1 and Q_4

a) The purpose of the tests with query Q_1 is to show that our progress indicator gives a reasonable estimate for a query which does not have multiple relations.

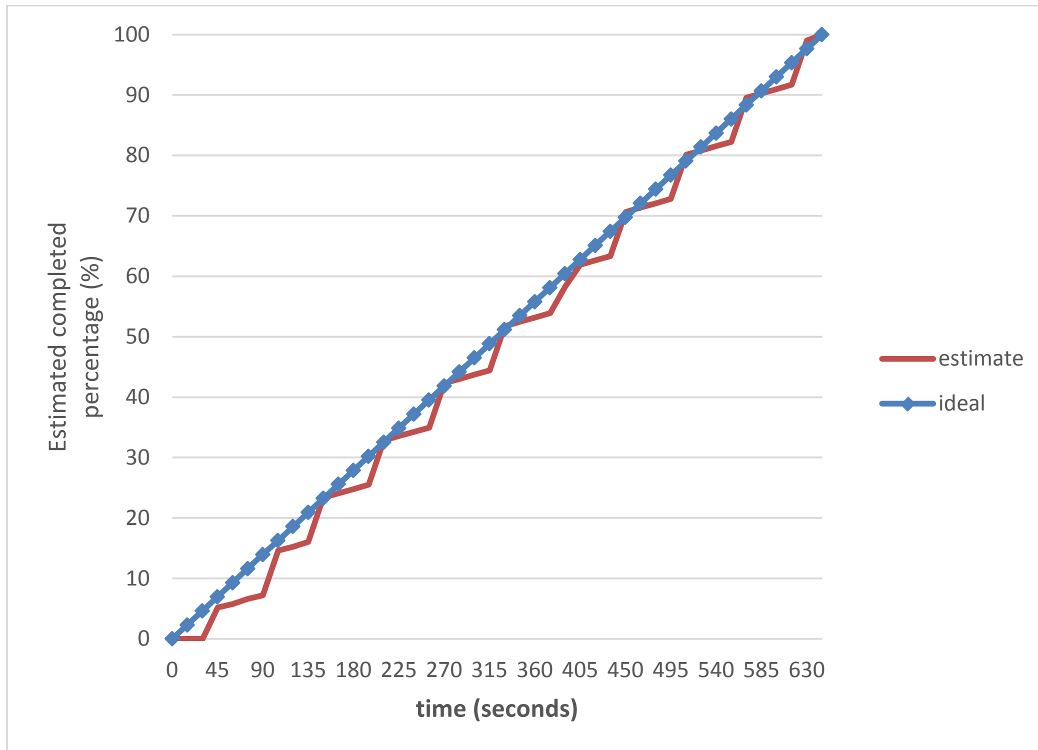

Figure 2: Estimation of percentage completed over time (test for Q_1)

In an ideal environment the curve which shows the estimated completed percentage over time, as work is continuously being done at a steady speed, is a straight line. Figure 2 shows the progress indicator's estimate of the percentage of Q_1 that has been completed over time. The curve is fairly close to a straight line.

b) The purpose of the test with query Q_4 is to show that how our progress indicator works fairly accurate for a query that has sub-queries and multiple relations.

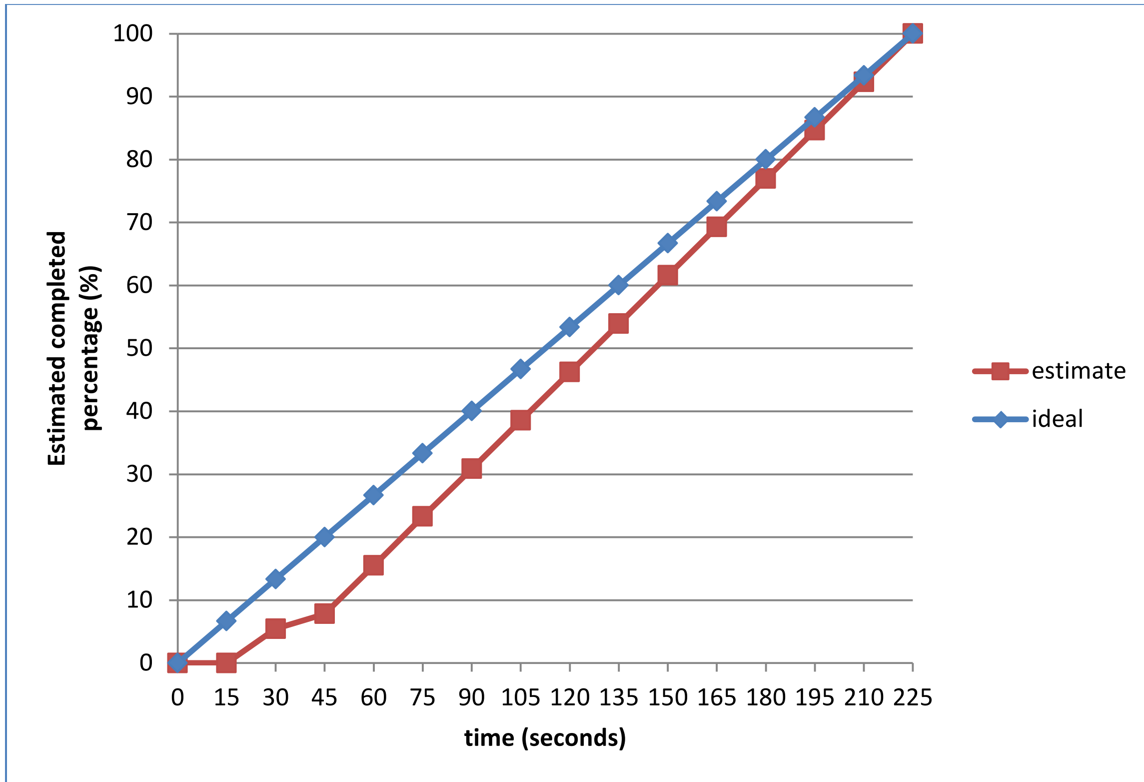

Figure 3: Estimated completed percentage over time (test for Q_4)

Figure 3 shows the progress indicator's estimate of the percentage of Q_4 that has been completed over time. As we can see in the figure the percentage keeps increasing with time as work is continuously being completed and it is fairly close to the straight line.

The PostgreSQL query optimizer chooses query plans for queries Q_1 and Q_4 and generates an execution tree for each of the query plans. Since Q_4 has a sub-query and multiple relations, it does not have a good estimation of the number of tuples returned by the sub-query at the beginning of the execution. So compared to the first query PI does not give an accurate estimation of progress.

3.4 Conclusion

We developed a simple progress indicator for PostgreSQL database and we showed that it provides a representation of progress for sample long-running queries. We evaluated our work for two queries from the standard TPC-H benchmark. We use the calculated progress of the OLAP query as an output of our progress indicator in the next chapter to show how a progress indicator can be used in the execution control of a long running query.

Chapter 4: Execution Control with a Progress Indicator

When multiple workloads running on a machine in parallel share machine resources, some may not meet their SLOs due to the lack of resource availability. This may result in important workloads violating their SLAs and in decreased system performance. Our goal is to give higher priority to the more important workloads while minimizing the impact on overall performance of the system.

Monitoring the execution times and the throughputs of the workloads is one of the major responsibilities of a DBA in order to ensure that the execution times and throughputs of all the workloads meet their SLOs. In this research we implement a query *Progress Indicator (PI),* which shows the incremental progress of a query execution process. A PI can, therefore, assist in determining the best approach to workload management to achieve the above mentioned goals. In this section we validate whether the use of a PI can make execution control more effective (i.e. provide better overall system performance). We use the PI to monitor primarily the progress of long running OLAP queries (*pp*). We define a set of *actions* and *policies* which dictate when execution control actions are taken to meet the performance goals. By observing

the progress of OLAP query we determine the best point in the query execution time to apply the execution control action.

The objective of an efficient execution control is to maintain a guaranteed throughput level for the more important OLTP workloads during the time of heavy or mixed workloads. In this work, we define the guaranteed throughput by a ***goal throughput value (tp_{goal})*** that is close to the throughput observed for the OLTP workload when it runs alone in the system. We consider OLAP or the long running queries to be less important workloads as small increase in their execution times are often ignored. Therefore, for a mixed workload which consists of both OLTP and OLAP workloads, we try to guarantee better throughput for the faster OLAP workloads.

We define ***average throughput (tp_{avg})*** as the throughput observed for the OLTP workload when both OLTP and OLAP workloads are running together without any execution control actions. Several *actions* can be taken for execution control. We measure the effectiveness of an execution control action by the improvement observed in the average throughput of an OLTP workload as a result of the action. We call this ***throughput of the OLTP workload under execution control (tp_{ec})*** when both OLTP and OLAP workloads are running together. So in our experiments we observe how close we are to the goal throughput (tp_{goal}) and how the execution control actions improve the average throughput when we have both workloads running together.

The second objective of efficient execution control is to minimize the total overall execution time while meeting the goals of the important workload. Application of any

execution control action typically increases the total execution time of both workloads. Therefore, while applying execution control for the OLTP workloads to meet the goal throughput tp_{goal}, we need to keep the total execution time of both workloads as close as possible to the maximum time of their individual execution times when no execution control action is taken. In other words, if each workload type is running alone with any execution control in the system and T_{OLTP} and T_{OLAP} are the average total execution times of the OLTP and OLAP workloads respectively then the ***goal for average total execution time (tt_{goal})*** would be ($max(T_{OLTP},T_{OLAP})$). We use ***tt_{avg}*** to denote the ***average total execution time*** of both workloads without any execution control.

When an OLAP workload starts running on a system which is already executing an OLTP workload, we see a decrease in the throughput of the OLTP workload. If the original throughput of the OLTP workload is tp_{goal} and tp_{avg} is the new throughput then we define the ***percentage drop off in OLTP throughput (tp_{drop})*** without any execution control action as:

$$tp_{drop} = (tp_{goal} - tp_{avg}) * 100 / tp_{goal} \quad \text{.....................................} \quad (1)$$

We use this measure as a trigger to take an execution control action for improving the OLTP throughput.

We define ***pp*** as the ***percentage of work completed for the W_{OLAP},*** which is used to decide when to take an execution control action. The PI is used to estimate *pp*.

The abbreviations or symbols we use in our experiments are listed in Table 1 for easy referencing:

Table 1: Abbreviations used in Description of Experimental Study

Symbol	Definition
W_{OLTP}	OLTP workload.
W_{OLAP}	OLAP workload.
T_{OLTP}	Execution time of OLTP workload while running alone in the system.
T_{OLAP}	Execution time of OLAP workload while running alone in the system.
tt_{avg}	The execution time of OLAP workload when it runs with OLTP without any execution control in the system.
tt_{ec}	The total execution time when all workloads finish their job while applying execution control.
tp_{goal}	Goal throughput for OLTP workload which is the average OLTP throughput when it runs alone in the system.
tp_{avg}	The average throughput of OLTP workload when both OLTP and OLAP workloads run in parallel in the system.
tp_{drop}	Percentage drops of OLTP throughput from tp_{goal} when both OLTP and OLAP workloads run in parallel in the system.
tp_{ec}	The average throughput of OLTP workload when running OLTP and OLAP together, applying execution control.
tp_{rec}	Percentage recovery of OLTP throughput applied to tp_{avg} when OLTP and OLAP workloads run together, after applying killing/throttling actions.
tt_{inc}	Percentage increase of execution time from tt_{avg} after applying killing/throttling actions when both OLTP and OLAP workloads run together in the system.
d	Percentage of throughput drop off observed that triggers killing/throttling without the usage of PI.
pp_{ec}	Query progress percentage of OLAP which triggers an execution control action.
pp	Query progress percentage of OLAP observed while using PI.

The rest of this chapter is structured as follows. We explain the execution control actions we apply in our experiments in section 4.1. In section 4.2, we list a number of scenarios for our experimental study to observe how a PI helps to provide better execution control. Different policies that govern the execution control actions for each of the scenarios are defined in section 4.3. We explained how the progress indicator integrated with execution control in section 4.4. In section 4.5 we present our experimental study in detail. Finally we discuss the results in section 4.6.

4.1 Execution Control Actions

A number of execution control actions can be applied when multiple mixed workloads run in the system which consist of both OLAP and OLTP queries. In this work we consider the following two actions:

1) Killing the less important workload
    - Pros
        a. The more important workload gets more resources to complete the job the less important workload frees the shared resources for the other important workloads.
    - Cons
        a. Killed jobs have to be executed again, and therefore, the total execution time of the workloads increases.

2) Throttling the less important, long running workload

- Pros

    a. By throttling the less important workload we do not lose the intermediate data.

    b. Resources are freed for more important workloads to complete their jobs.

- Cons

    a. Execution time of the less important workloads increase because of getting throttled.

Based on Powley's work [25], their experiments suggests that the application of constant throttling on the low priority OLAP query in a mixed OLTP and OLAP workload gives better performance when throttling is applied at the start of the execution of OLAP query than in the middle or towards the end of the query. Also the work shows that the long-pauses give better execution control for constant throttling than the short pauses. Based on the above observations published in [25], since we also use a combination of OLAP and OLTP workloads with the higher priority for OLTP query, we use constant throttling with 500 pauses each lasting for 1 second in our experiments. We chose this number based on an empirical study of the effect of the number of pauses on the query execution time and throughput as shown in Table 2:

Table 2: Applying throttling with different pauses on OLAP while running with OLTP

tt_{avg}	tp_{goal}	tp_{avg}	number of pauses	throughput	execution-time
11	3503	2521	100	2646	12
			200	2679	12
			300	2793	14
			500	3013	15
			600	3034	17
			700	3076	17

As we can observe from above table, by applying 500 pauses we can reach a throughput average of 3000 with an execution time of 15 minutes. The goal is to find a compromise to achieve a throughput that is close to the goal throughput and an execution time that is close to the average execution time.

4.2 Experimental Scenarios

Assumptions: We consider the OLAP long-running queries as less important queries and the OLTP queries as more important or high priority queries that need to be completed quickly. So if the progress of the OLTP queries slows down we need to take an action. OLTP and OLAP queries are running together at the same time. We also consider OLTP and OLAP workloads running on different database instances.

We use 4 different scenarios to validate the usefulness of the PI in providing better execution control. In each scenario, we demonstrate the effectiveness of an execution control action with or without the use of our PI as explained below.

Scenario 1: Killing action without PI

Kill the OLAP query to increase the throughput of the OLTP workload to meet tp_{goal} without using the PI.

Scenario 2: Throttling action without PI

Apply throttling to the OLAP query whenever OLTP throughput decreases beyond a predefined threshold value without using the PI.

Scenario 3: Killing action with PI

Kill the OLAP query to increase the throughput of the OLTP workload to meet tp_{goal} based on the following observations:

a) Observe OLTP throughput and take action when it drops below tp_{goal}.
b) Use PI to observe the progress of OLAP query and take action at different points of execution of the OLAP query.

Scenario 4: Throttling action with PI

Apply throttling to the OLAP query based on the following observations:

a) Observe OLTP throughput and take action when it starts decreasing and drops below tp_{goal}.

b) Monitor the progress of the OLAP workload by observing the outcome of PI and apply throttling action at different points of execution of the OLAP query.

The scenarios based on the different actions are summarized in Table 3:

Table 3: Scenarios and actions

Execution Control	Without PI	With PI
Kill	Scenario 1	Scenario 3
Throttle	Scenario 2	Scenario 4

4.3 Policies:

We define the following policies which dictate when the execution control actions are applied for the four different scenarios explained earlier in section 4.2.

Scenario 1:

- If OLTP throughput decreases that reaches a predefined threshold value *d,* we kill the OLAP query.

(*d* is the percentage drop in OLTP throughput. The value of *d* is determined experimentally and varied in the range of OLTP throughput drop off).

Scenario 2:

- If OLTP throughput decreases that it reaches a specified number *d*, we throttle the OLAP query.

(As stated above, *d* is determined experimentally and varied in the range of OLTP throughput drop off)

Scenario 3:

- If OLTP throughput decreases that it reaches the threshold value *d* as in the first scenario and the query progress percentage pp_{ec} of the less important workload W_{OLAP} increases that reaches a specified number *pp,* we kill the W_{OLAP}. The value of *pp* is varied in the experiments from 10~90 % by 10.

Scenario 4:

- If OLTP throughput reduces by *d (*the parameter we observed in the second scenario) and OLAP workloads progress percentage pp_{ec} increases to reach a specified number *pp* (which again is varied in the experiments from 10~90% by 10) we decide to throttle the OLAP query.

4.4 Integrating Progress Indicator with Execution control mechanism:

The execution control mechanisms used in this thesis are throttling and killing. In PostgreSQL source code a class named acmanager.c is known as Query Controller. In

this class we calculate the progress percentage of OLAP query (as indicated in 3.1) and decide whether to kill or throttle the query based on the policies discussed in 4.3. To set up throttling we just need to turn the throttling process `on` in the acmanager.c class in the PostgreSQL source code by setting the *throttled* parameter and calling *kill* function:

throttled = 1;

kill(ids[0], SIGURG);

And to kill the OLAP query we just need to change the second parameter in *kill* function from *SIGURG* to *SIGINT* after turning off the throttle parameter and call this function in acmanager.c:

throttled = 0;

kill(ids[0], SIGINT);

4.5 Experiments and Validation:

Experimental Setup: The experiments are performed on a server which has a 2.80 GHz processor, 3 GB main memory, and Ubuntu operating system. The *pgbench* [17] tool is used for generating the OLTP workload. *pgbench* is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second).

Typical output from *pgbench* looks like:

```
transaction type: TPC-H
scaling factor:10
number of clients: 30
number of transactions per client: 2500
number of transactions actually processed: 75000 / 75000
tps: 3554
```

The first four lines report some of the most important parameter settings. The next line reports the number of transactions completed and intended (the latter being just the product of number of clients and number of transactions per client); these will be equal unless the run failed before completion. The last line reports the number of transactions per second. More explanation on how pgbench works is available in Appendix B.

We use the following workload combination:

- **OLTP Workload –** 30 clients send simultaneous requests to a PostgreSQL database where each client request represents an OLTP workload, W_{OLTP}, consisting of 2500 simple *select* statements. So the total number of requests for the OLTP workload is 75000 which take 10 seconds to run.
- **OLAP Workload** - We used (Q_1) from TPC-H standard queries (Appendix A) and use it as the OLAP workload W_{OLAP}. It is the only query in the series of TPC-H standard queries that executes more than 10 minutes and we choose it as our long-running OLAP query.

We start running OLAP workload in a separate console window right after starting the OLTP workloads. Before running each query we restart the computer to make sure that we have a cold buffer pool. (We also repeated the experiments with a warm buffer pool and observed that the results are the same). The metrics are collected in a specific output file by our software application running on the same machine, which includes the throughputs.

Each experiment is repeated three times for each of the scenarios and the mean value is taken as the final result. The standard deviation is calculated and shown in the appropriate tables for each set of experiments.

4.5.1 Experiments:

Objective: We execute two sets of experiments. The first set of experiments is performed to determine the base and goal values for performance comparisons without any execution control actions. We use the results of this section to set some of the metrics such as T_{OLTP}, T_{OLAP}, tp_{goal}, *and* tp_{avg} for the next set of experiments which help us decide when to take an action for execution control. Then we execute the second set of experiments for the four scenarios to validate the usefulness of the PI.

4.5.1.1 Experiment Set 1: No Execution Control

We set the results obtained in this phase as the best or desired goal values as explained in section 4.3. We let the workloads run without taking any action and use the

results for comparing the performance of the workloads and assessing the effectiveness of the execution control actions in the other experiments.

We determine:

- Execution time T_{OLTP} of the OLTP workload W_{OLTP} when it is running alone in the system. We use T_{OLTP} in the other experiments anytime we want to use the OLTP execution time.
- Average throughput tp_{goal} of W_{OLTP} workload when it is running alone in the system.
- Execution time T_{OLAP} of the OLAP workload W_{OLAP} when it is running alone in the system. We use T_{OLAP} in the other experiments anytime we want to use the OLAP execution time.
- Average total execution time (tt_{avg} = max (tt_{OLTP}, tt_{OLAP})) when running W_{OLTP} and W_{OLAP} together at the same time in the system. We use tt_{OLTP} and tt_{OLAP} to show the execution times of the OLTP and OLAP when running the two workloads together.
- Average throughput for the OLTP queries tp_{avg} when running W_{OLTP} and W_{OLAP} together at the same time in the system.

Table 4: Observations with no execution control

No Execution Control	Observations	
	Execution Time (min)	Avg. Throughput (TPS)
Part 1 - Only OLTP	$T_{OLTP} = 8$	$tp_{goal} = 3503$
Part 2 - Only OLAP	$T_{OLAP} = 10$	-
Part 3 - OLTP + OLAP	$tt_{OLTP} = 10$ $tt_{OLAP} = 11$ $tt_{avg} = 11$	$tp_{avg} = 2521$

Observations: From Table 4 , it takes 8 minutes on average for the OLTP workload to run, without having the OLAP query executing at the same time. When running the OLAP workload alone in the system, it finishes its work on average in 10 minutes. When both OLTP and OLAP workloads run together in the system, on average the OLTP workload takes 10 minutes and the OLAP workload takes 11 minutes to finish. So the OLTP and OLAP workloads when run together take more minutes to complete than when each of them runs alone in the system.

Total execution time tt_{avg} indicates the time span from when the first workload starts to when both workloads complete.

4.5.1.2 Experiment Set 2: Apply Execution Control with and without the PI:

Objective: In this set of experiments, by monitoring the OLTP throughput we determine whether it is decreasing or not (by comparing it with the average throughput

tp_{avg} as measured in the previous experiments) and we monitor the percentage of throughput drop off. If the throughput decreases beyond a predefined threshold value, *d*, we apply killing or throttling actions to recover the throughput of the OLTP workload, tp_{avg}, to a value closer to the desired goal throughput tp_{goal} as determined in the previous experiments in section 4.5.1.1. We run experiments for the four scenarios as explained in section 4.2. The first two experiments (scenario 1 and 2) show the effects of applying execution control without using the PI. In these experiments we also try to determine the values of *d* which give the best tp_{ec}, OLTP throughput with execution control, and the best tt_{ec}, the average total execution time of both workloads with execution control. The next two experiments (scenario 3 and 4) show the effects of using the PI with execution control with a view to validate the usefulness of the PI. In these experiments we use the values we observe for *d* in the first two experiments.

Apply Execution Control without the PI - Scenario 1 and 2

Objective: In these experiments, we monitor impact on performance of taking an *execution control action, ECA,* on the long running OLAP query without using the PI. When the OLTP and OLAP workloads run together in the system, there is an increase in the total average execution time and a decrease in the average OLTP throughput as observed in experiment 1. So the main objective is to increase tp_{ec} closer to tp_{goal} and to have tt_{ec} as small as possible. We apply the ECA on the OLAP query at different points of its execution. In each experiment we observe tt_{ec}, and tp_{ec}.

We start the OLTP workloads W_{OLTP} first and then the OLAP workload W_{OLAP}. We monitor the throughput for the OLTP workload tp_{avg}. Initially the average throughput is high and close to tp_{goal} but it drops rapidly as we start running W_{OLAP}. We observe the drop off percentage of OLTP throughput tp_{drop} as given in Eq. 1

At first we don't observe that much drop in the throughput but right after starting W_{OLAP} we notice a 25% drop in the throughput which gets to its worst point of 31% and then remains consistent at around 30-31%. So in our case study we have the range of 25~31% drop off of the OLTP throughput during which we need to apply an ECA and observe the impact on the average OLTP throughput tp_{ec} and the total execution time tt_{ec}.

We study the effect of an ECA by dividing the drop off range (tp_{drop}) of 25~31% into several action points d, and by applying the ECA on W_{OLAP} at these points to observe the improvements of tp_{ec} and tt_{ec} by comparing them with tp_{avg}, tp_{goal} and tt_{avg}. By comparing the new results with tp_{avg}, we observe how much we recover tp_{avg} and with the comparison we make to tp_{goal} we perceive how close we are to the final goal. When tp_{drop} becomes greater than a predefined threshold d, we apply the ECA on W_{OLAP}. Therefore, the action policy can be expressed as:

If $tp_{drop} > d$ *then apply ECA* .... (2)

In our experiments, we divide the drop off range of 29~36% into 1% intervals each (*d=25%, d=26%, d=27%,..*) and assign that to *d* in each experiment to determine the best point to apply the ECA. We repeat each experiment several times (5 times), and note the mean values of tp_{ec} and tt_{ec}.

Observation and Results - Scenario 1: Taking Killing Action without using PI

Here we **kill** the $\boldsymbol{W_{OLAP}}$ as the ECA. We monitor tp_{ec}, tp_{drop}, and the total execution time tt_{ec}. In this case tt_{ec} is calculated from the observed tt_{OLTP} and T_{OLAP} since W_{OLAP} was killed and needs to be run again after completion of W_{OLTP} as given below:

$$tt_{ec} = tt_{OLTP} + T_{OLAP} \quad \text{.. (3)}$$

The mean values of tp_{ec} and tt_{ec} are reported in the following table.

Table 5: Taking killing action on W_{OLAP} without PI

d (%)	tp_{goal}(TPS)	tp_{avg}(TPS)	tt_{avg}(min)	tp_{ec} (TPS)	tt_{ec} (min)
25	3503	2521	11	3448	18.11
26				3438	18.23
27				3413	18.33
28				3343	18.68
29				3312	19.1
30				3292	19.21
31				3090	19.33

As we can see the best value for parameter *d* is when we see a drop off of 25% in the system at which we gain the best throughput for OLTP queries by killing the OLAP query, (tp_{ec} = 3448). As the drop off percentage of throughput increases, killing the OLAP query increases the average throughput only by a small amount. We also observe that the total execution time is not important in this case because killing the OLAP workload at different points has very little impact on the total execution time.

We make a note of this parameter *d* = 25% for ECA= 'killing' to use it in future experiments while using PI.

Observation and Results - Scenario 2: Taking Throttling Action without using PI

Here we **throttle** the W_{OLAP} as the ECA whenever tp_{drop} becomes greater than d. We observe tp_{ec} as in the previous experiment, tp_{drop}, and the total execution time tt_{ec}, which in this case is calculated by $(max\ (tt_{OLTP} + tt_{OLAP}))$ as given below since both W_{OLTP} and W_{OLAP} run in parallel in the system:

$$tt_{ec} = max\ (tt_{OLTP} + tt_{OLAP}) \qquad (4)$$

The mean values of tp_{ec} and tt_{ec} are reported in the following table.

Table 6: Taking Throttling action without PI observation

d (%)	tp_{goal}(TPS)	tp_{avg}(TPS)	tt_{avg}(min)	tp_{ec} (TPS)	tt_{ec} (min)
25	3503	2521	11	2994	16.16
26				2882	16.2
27				2811	16.61
28				2796	16.81
29				2773	17.16
30				2736	17.23
31				2708	17.33

As we can see from the Table 6 the best point to apply throttling is whenever we observe a drop off of 25% in the system. By waiting to apply the throttling action at d > 25%, we will not get better results since we see from the table that tt_{ec} does not have a

considerable change with *d* (we observe small changes in seconds) and throughput decreases.

We make a note of this parameter *d* = 25% for ECA = 'throttling' to use it in future experiments while using the PI.

Apply Execution Control with the PI - Scenario 3 and 4

Objective: In these experiments, we monitor the performance of taking an execution control action, ECA, on the long running OLAP query while using the PI. When the OLTP and OLAP workloads run together in the system, there is an increase in the total average execution time tt_{avg} and a decrease in the average OLTP throughput tp_{avg} as observed in experiment 1. To decrease the total execution time while not hurting throughput, we apply ECA on the OLAP query at different points of its execution. In each experiment we observe the total execution time, tt_{ec}, and the average throughput, tp_{ec}, percentage drop off of OLTP throughput, tp_{drop} and the query progress percentage, pp_{ec} using our PI. When tp_{drop} exceeds a predefined threshold *d*, we check pp_{ec} to see if it reaches a predefined threshold *pp*, and if so, we apply an execution control action, ECA, on the OLAP query. Here *d* has been determined as 25% from the previous experiments for both killing and throttling actions. In these experiments, we determine the value of

pp_{ec} in similar way. We also monitor T_{OLAP} of the controlled OLAP query taking into consideration that if W_{OLAP} is killed, it has to be executed again.

As we mentioned we compare the throughput achieved after applying ECA with tp_{goal} and tp_{avg} to see how close we are to the goal throughput and by how much we recovered the average throughput value.

Also we determine the throughput recovery percentage of OLTP throughput tp_{rec} where,

$$tp_{rec} = (tp_{ec} - tp_{avg}) * 100 / tp_{avg} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5)$$

We calculate this metric to show that how close tp_{ec} gets to tt_{avg}. We observe the increase in execution time after applying an ECA by comparing it with tt_{avg}. We measure the percentage increase in total execution time of the mixed workloads as:

$$tt_{inc} = (tt_{ec} - tt_{avg}) * 100 / tt_{avg} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (6)$$

We use the PI primarily to estimate pp_{ec}, i.e., the percentage of work completed for the W_{OLAP}. While pp_{ec} varies from 10~90%, we start with pp = 10% and increase it by

10 in each experiment to see which value of *pp* gives the best tp_{ec} *and* tt_{ec}. We repeat each experiment several times, and note the mean values of tp_{ec} and tt_{ec}.

The action policy can, therefore, be expressed as:

If ($tp_{drop} > d$ *and* $pp_{ec} >= pp$) *then apply ECA* ……………………… (7)

To validate the usefulness of the PI, we compare tp_{ec} and tt_{ec} with the values obtained from the previous experiments where the same ECA was applied without using the PI.

Observation and Results - Scenario 3: Taking Killing Action while using PI

Here we **kill** the $\boldsymbol{W_{OLAP}}$ as the ECA. We monitor tp_{ec}, tp_{drop} and tt_{ec}. In this case tt_{ec} is calculated as shown in equation (3) because of similar reasons. The value for *d* in this scenario is 25% based on our previous observations. (*d* = 25%).

The results are presented in the Table 7.

Table 7: Taking Killing action while using PI

tp_{goal} (TPS)	tp_{avg} (TPS)	tt_{avg} (min)	pp_{ec} (%)	tt_{ec} (min)	tp_{ec} (TPS)	tp_{rec} (%)	tt_{inc} (%)	*Standard deviation % (For throughput)*
3503	2521	11	10%	18.7	3404	35.02	62	8.8
			20%	18.81	3327	31.97	63	9.0
			30%	19.11	3259	29.27	66	10.2
			40%	19.23	3203	27.05	67	11.4
			50%	19.68	3037	20.46	71	12.0
			60%	20.16	2948	16.93	75	12.3
			70%	20.2	2863	13.56	76	12.4
			80%	20.61	2768	9.79	79	11.2
			90%	20.81	2689	7.02	81	9.7

We have the explanation of the columns in Table 1 and equations (5), (6) and (7).

As we can see the best throughput is 3404 and the best execution time is 18 minutes at the point of 10% of work completed for the W_{OLAP} when we apply killing action.

By comparing the largest throughput while using PI (tp_{ec} = 3404) with the largest throughput without using PI (tp_{ec} = 3448 from Table 5) we can observe that PI helps us having a good view of the progress of OLAP query but the best point to apply killing action is when we observe a drop off in the system. It means that it is better to kill the query as early as possible. The more a query has progressed the less the benefit in killing it.

Observation and Results - Scenario 4: Taking Throttling Action while using PI

In the fourth scenario we **throttle** the W_{OLAP} as the ECA. We monitor tp_{ec}, tp_{drop}, tt_{ec} and pp. In this case tt_{ec} is calculated as shown in Eq. (4) because of similar reasons. The value for d in this scenario is 25% based on our observations (d = 25%).

The observations are presented in Table 8.

Explanation of table columns are the same as the description we defined for Table 7.

Values:

Table 8: Taking Throttling Action while using PI

tp_{goal} (TPS)	tp_{avg} (TPS)	tt_{avg} (min)	pp_{ec} (%)	tt_{ec} (min)	tp_{ec} (TPS)	tp_{rec} (%)	tt_{inc} (%)	Standard deviation % (For throughput)
3503	2521	11	10%	14.5	2928	16	26	5.9
			20%	14	2857	13	22	6.7
			30%	13.75	2839	12	19	8.0
			40%	13.33	2814	11	16	8.1
			50%	12.66	2751	9	10	8.2
			60%	12.33	2716	7	7	8.5
			70%	12.16	2683	6	6	8.1
			80%	12.88	2633	4	3	7.5
			90%	11.61	2603	3	0.9	6.4

As shown in the table the best value for tp_{ec} is 2928 and it is applying throttling on W_{OLAP} when only 10% of OLAP work is done. Compared to the best value with the same ECA while not using PI (tp_{ec} = 2994 from Table 6) we can observe that although PI gives us a good observation of the progress of the OLAP query the best point to apply ECA is not let the system continue working in contention. The sooner we take the action the better the result would be in terms of throughput and execution time.

4.6 Discussion and Critical Analysis:

In this section we do a critical analysis of the experimental results as presented in the previous section. For each set of experiments of the four different scenarios we compare the results to show the impact of using the PI.

4.6.1 Discussion

Figure 4 illustrates the throughput recovery for the killing and throttling actions when they are applied at different states of the execution of workloads. As it can be seen in this figure, selecting the killing action shows more improvement of the throughput tp_{ec} as compared to throttling. The effect of both killing and throttling on the throughput improvement decreases when the action is taken later during the execution of the queries.

Also we observe that taking action at any point of execution of the W_{OLAP} recovers the throughput, and hence, taking an action is better than taking no action.

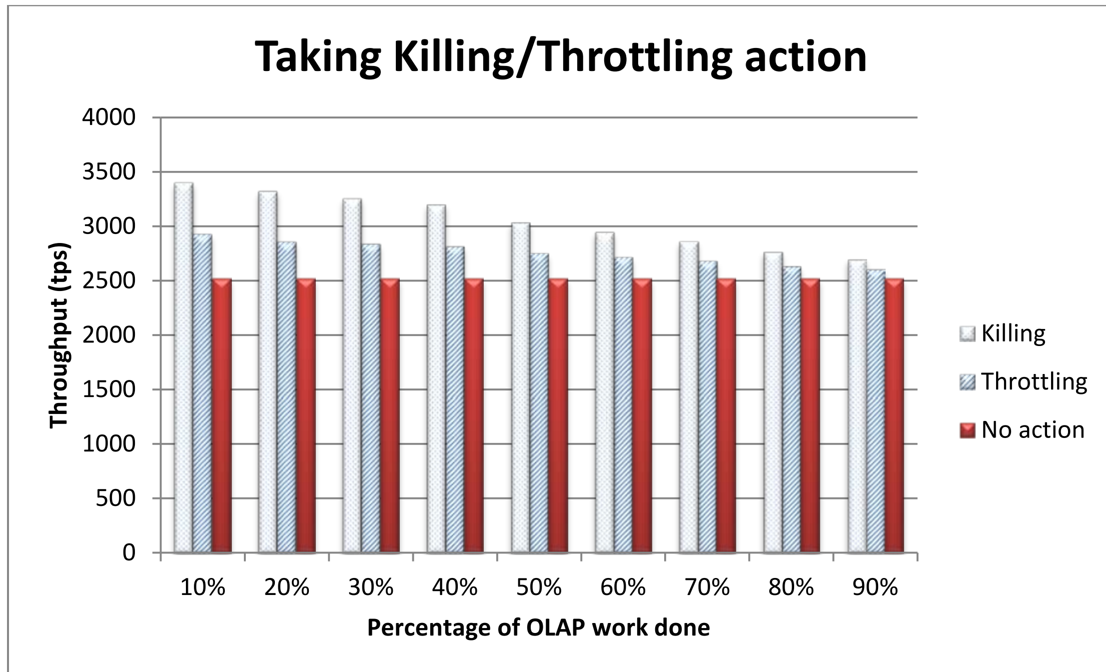

Figure 4: Throughput recovery after applying ECA with the usage of PI

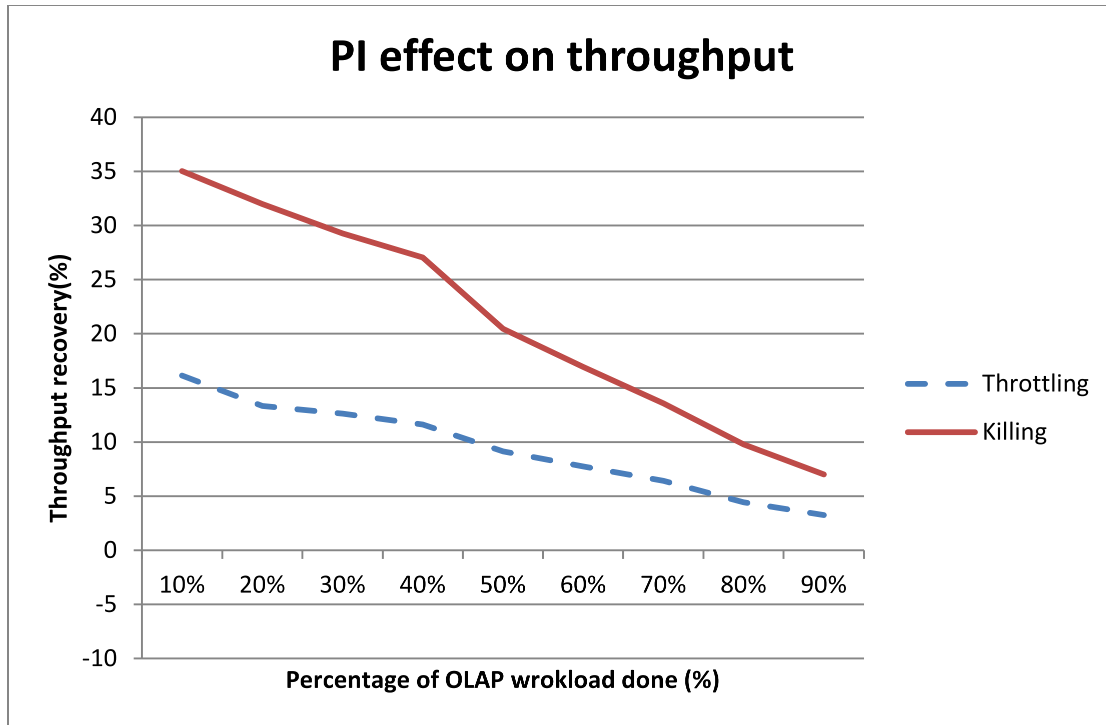

Figure 5: Throughput recovery percentage with the usage of PI

Figure 5 shows the percentage of throughput recovery of the system after applying ECA at different states of execution of W_{OLAP}. The best time to take an action to have the best recovery on throughput is when just 10% of the W_{OLAP} has been finished. In this case we observe a throughput recovery tp_{rec} of 35% for killing and 16% for throttling. This is because by killing the OLAP query we release shared resources for OLTP queries therefore the throughput increases significantly. We observe a slight decrease of throughput recovery while taking throttling action. After 70% completion of OLAP workload there is a recovery of just 3-6% in throughput. The reason that we observe better throughput recovery for killing as compared to throttling is that in throttling, W_{OLAP} continues to run with pauses and still uses the shared resources that

are used by W_{OLTP}. As a conclusion the amount of recovery percentage is significant when taking killing action as an ECA early in the execution of the query.

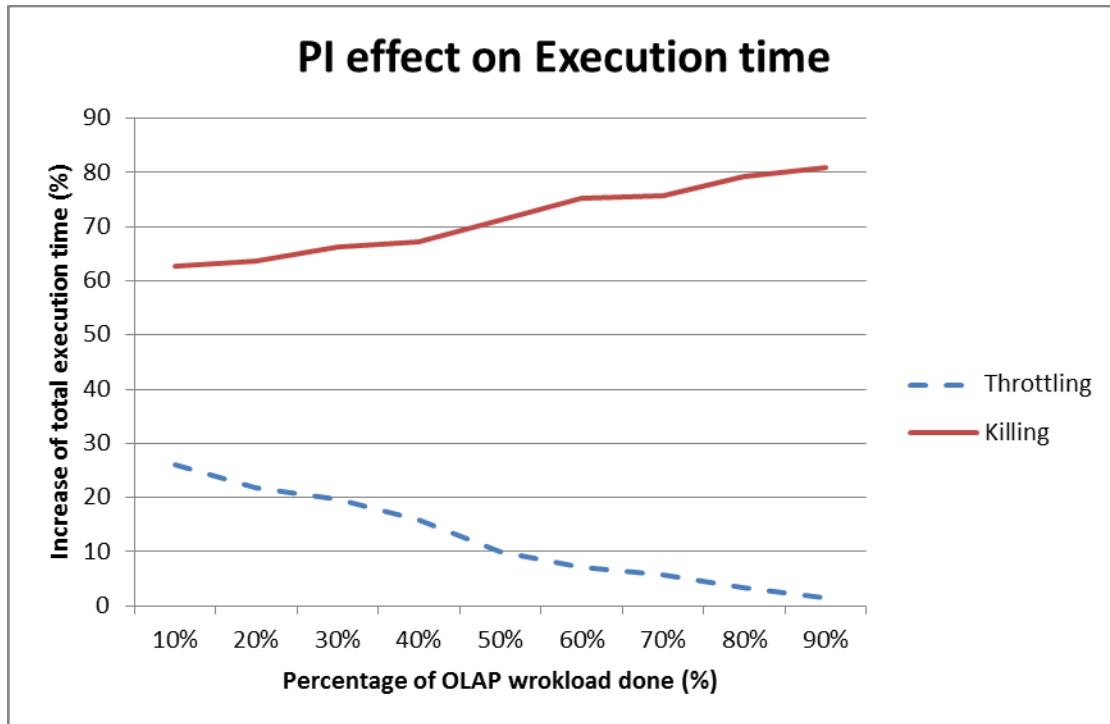

Figure 6: Execution time increase percentage with the usage of PI

Figure 6 shows the percentage of increase in total execution time when killing or throttling is applied as compared to when we take no actions (when OLAP/OLTP workloads are running together). As we can see killing increases the execution time by 62% (if the action takes place at the point of 10% completion) while throttling increases execution time by only 26%. Study of the effects of ECA on the execution time shows that for killing the execution time is far from the average execution time and as time

goes on it gets worse. Throttling has less obvious effects and we observe just 1.5% increase in execution time if throttling is applied when OLAP workload is close to finish.

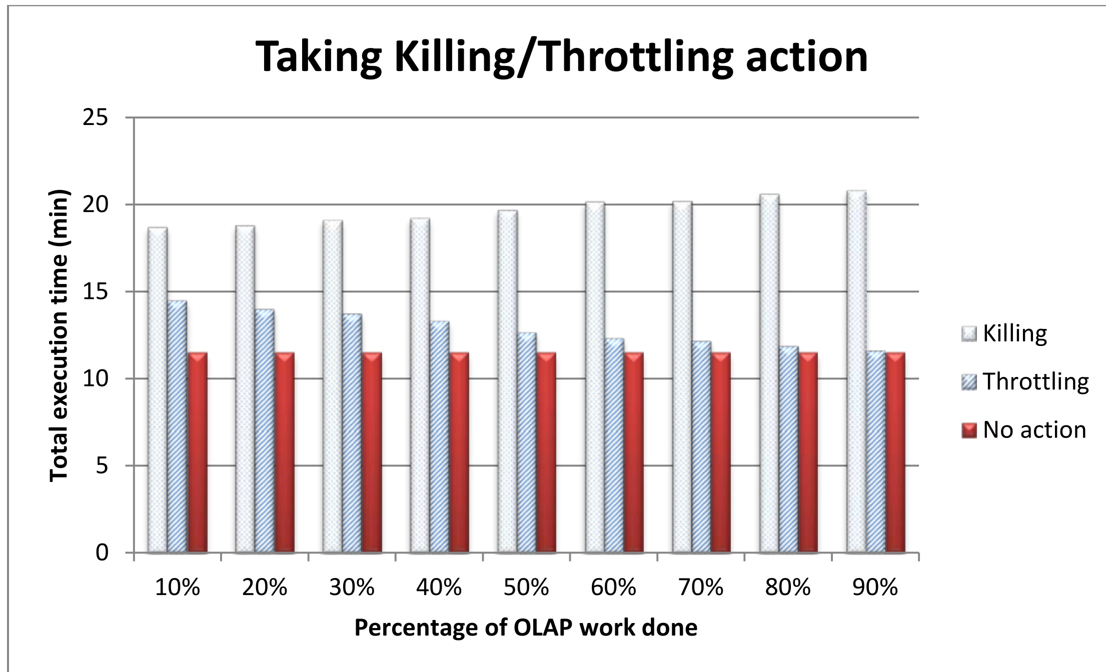

Figure 7: Execution time as a result of killing/throttling with the usage of PI

Figure 7 shows PI effect on total average execution time. The total execution time shows two completely different trends based on what action is taken, killing or throttling, and how much of W_{OLAP} has been completed. If W_{OLAP} is closer to being finished, applying killing action at that point increases the total execution time more than that obtained by applying the throttling action. This is because of the killing action; W_{OLAP} has to be executed again from the start. Therefore, killing action is not a good choice when the goal is to maintain the total execution time as low as possible. Also

with throttling since throttling terminates when execution of W_{OLAP} finishes, taking action closer to the start of W_{OLAP} causes more interrupts in the system than if the action is taken towards the end of W_{OLAP}.

4.6.2 Critical Analysis

We discussed above the effects of the killing and throttling actions on the average OLTP throughput and the total workload execution time. Given that the OLTP starts at the same time of the OLAP the best time to take an action for the best recovery of throughput is when just 10% of the OLAP process has been finished. From Figure 6 and Figure 7, both throttling and killing increase the execution time compared to the time when no action is taken in the system. The best case is when 10% of the W_{OLAP} is finished and the throttling action is taken, which increases the total execution time by 26%. We conclude that if the total execution time is the important factor for the user, then no action is the best action as both execution control actions have negative effects on the total execution time.

In addition to our observation that PI is an effective tool to help us make a better decision we want to see which of the actions helps in gaining better performance with the usage of PI and taking both execution time and throughput as metrics.

So we define utility, a metric for measuring the effectiveness of the ECA, for our system. Utility represents how much we have recovered of the throughput and execution time compared to when we run the OLTP and OLAP together in parallel and

no control action is taken. To define utility for the system we define two terms, *Benefit* and *Cost*. We define *Benefit* as the amount of throughput recovery made with respect to the average throughput (tp_{avg}) using some execution control action. So the definition is:

$$\frac{tp_{ec} - tp_{avg}}{tp_{avg}} * 100 \qquad (8)$$

We define *Cost* as the increase in execution time while applying execution control as compared to the average execution time (how much we lost time). So we have *Cost* as:

$$\frac{tt_{ec} - tt_{avg}}{tt_{avg}} * 100 \qquad (9)$$

(tt_{ec} is always greater than tt_{avg} which means we have cost in all cases)

$$Utility = \frac{Benefit}{Cost} \qquad (10)$$

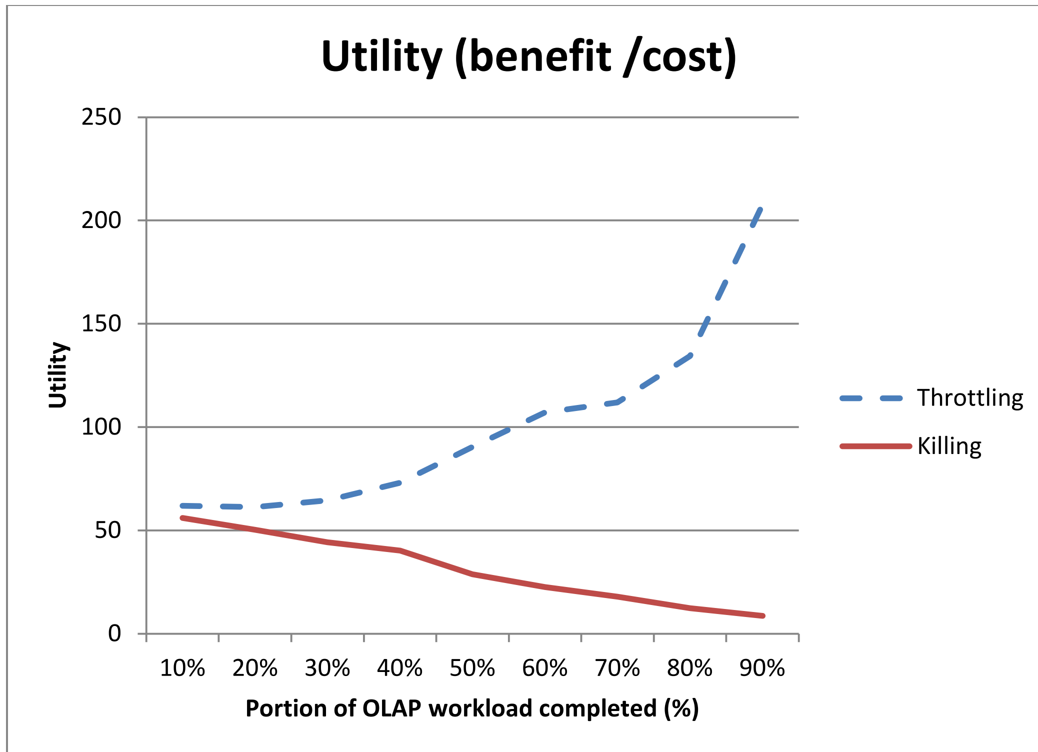

Figure 8: Utility of system

Figure 8 shows the utility using throttling and killing. This chart is useful when it is important for users to reach a balance between the improvement in the throughput and the cost in terms of the total execution time of the workloads. It can be inferred from the figure that the total utility for improving throughput and execution time is greater for the throttling action than for the killing action. Another impression of these illustrations is that the system gains a considerable amount of utility if we take throttling action at the end of execution of the OLAP workload because we have improvements in throughput and also in execution time as the OLAP workload progresses. On the contrary, for the killing action, the system gains less when we let the OLAP workload

execute longer and then apply killing action. The reason for the above is that as time goes on we observe a decrease in throughput and an increase in execution time.

As a conclusion if we consider execution time and throughput as the important factors of workload management then by choosing throttling we gain more utility compared to killing. Also based on the utility values shown in Figure 8 taking an action is always better than no action.

Chapter 5: Conclusion and Future Work

By running OLTP and OLAP workloads together in a system with shared resources, the system may encounter serious problems such as increase in execution time and decrease in throughput. So taking an applicable action at suitable points of execution of the workloads, in particular, the long running OLAP workloads, gives a better overall performance.

Without tracking the progress of an OLAP query we can only rely on applying ECA at different points of throughput drop off of the workload. With this approach, although we observe a recovery in throughput there is no guarantee that the achieved recovery is the best throughput recovery since we do not take into account the OLAP query progress. So taking OLAP workload progress into consideration we have a better view of the overall system performance and are able to find the best point to apply ECA to get a better performance.

We validate the usefulness of tracking the progress of OLAP query by comparing the effects of execution control in terms of total execution time and throughput with and without the Progress Indicator (PI). We showed when the best point to apply execution control mechanisms is. The results shows that the sooner we apply execution control actions the better. We did experiments for various scenarios, combinations of

different ECAs and policies, and with and without the PI. By dividing the scenarios into four different groups first we observe the effects of the ECAs namely killing and throttling, on throughput and total execution time without tracking the progress of the OLAP workload. Then in the second two sets of scenarios we repeat the same experiments while tracking the progress of the OLAP workload and enhanced policies.

We show experimental results and charts to demonstrate the effectiveness of the killing and throttling execution control actions on achieving a better overall system performance. We define a utility metric to further demonstrate the usefulness of using a PI. The charts show that a more informed application of execution control using a PI can help achieve a better overall performance of the workloads.

We limited the scope of our research on running OLTP and OLAP at the same time, we can think of running these workloads at different time but have overlapping periods of execution and observe the results of taking actions at different progress points of OLAP workload as a future work. As a future work, we can have multiple OLAP workloads with different priorities and apply execution control. Also this work can be integrated in an automatic workload management system and it can get the information about queries in admission control as an input, classify and prioritized the query based on their behavior and query the workloads based on their priorities. We can also give different values to the workloads during admission control and based on the given values, we can think of applying execution control actions at different progress points of

the OLAP workloads. In this study we gave the same value to throughput and execution time. We also can think of giving different importance to throughput and execution time and studying the behaviour of system as an extension to this work.

References

[1] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. “Query Suspend and Resume”. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, Beijing, China. June 11-14, 2007, pp. 557-568.

[2] B. Niu, P. Martin and W. Powley, “Towards Autonomic Workload Management in DBMSs”. In *Journal of Database Management*, 20(3), 1-17, July-September 2009, pp. 154-173.

[3] C. Mishra and N. Koudas, “The design of a query monitoring system,”*ACM Trans. Database Syst.*, vol. 34, 2009, pp. 1:1–1:51.

[4] C. Mishra and N. Koudas, “A lightweight online framework for query progress indicators,” *ICDE*, 2007, pp. 203-214.

[5] D. P. Brown, A. Richards, R. Zeehandelaar and D. Galeazzi, “Teradata Active System Management: High-Level Architecture Overview”, a *White Paper of Teradata*. 2007. [Online]. Available: http://www.teradata.com/white-papers/Teradata-Active-System-Management-High-Level-Architectural-Overview-b4685/?type=WP.

[6] D. Hornby, B. Walker, and K. Pepple. “Consolidation in the Data Center: Simplifying IT Environments to Reduce total Cost of Ownership”. Pearson Education, 2002.

[7] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. “Toward a Progress Indicator for Database Queries”. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2004.

[8] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke, “Increasing the accuracy

and coverage of SQL progress indicators," in *ICDE* Workshops, 2005.

[9] IBM Corp., "Basics of Business Intelligence Systems - An Executive Overview". *A White Paper of IBM*. June 2003. [Online]. Available: ftp://ftp.software.ibm.com/software/emea/dk/frontlines/IBM_Basics_of_BI.pdf.

[10] IBM Corp., "IBM DB2 Database for Linux, UNIX, and Windows Information Center". *On-line Documents*. https://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp.

[11] J. Li, R. V. Nehme, J. Naughton. "GSLPI: a Cost-based Query Progress Indicator". *ICDE* Workshops, 2012, pp. 678-689.

[12] M. Badaloo. "An examination of server consolidation: trends that can drive efficiencies and help businesses gain a competitive edge". [Online]. Available: http://www-03.ibm.com/systems/data/flash/virtualization/pdf/server_consolidation_whitepaper.pdf. IBM White Paper, August 2008.

[13] M. Zhang, P. Martin, W. Powley, P. Bird, K. McDonald: "Discovering Indicators for Congestion in DBMSs". *ICDE* Workshops, 2012, pp. 263-268.

[14] M. Dempsey. Monitoring Active Queries with Teradata Manager 5.0. http://www.teradataforum.com/attachments/a030318c.doc, 2001.

[15] Oracle Corp., "Oracle Database Resource Manager". *On-line Documents*. http://download.oracle.com/docs/cd/B28359_01/server.111/b28310/dbrm.htm#i1010776.

[16] PostgreSQL homepage, 2012. http://www.postgresql.org.

[17] PostgreSQL development documentation, 2012, version 9.2, http://www.postgresql.org/docs/manuals

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

[19] S. Chaudhuri, V. Narassaya, R. Ramamurthy, "Estimating Progress of Execution for SQL Queries", in *SIGMOD*, 2004, pp. 803-814.

[20] S. Chaudhuri, R. Kaushik, and R. Ramamurthy, "When can we trust progress estimators for SQL queries?". *SIGMOD*, 2005, pp. 821-832.

[21] S. Krompass, H. Kuno, J. L. Wiener, K. Wilkison, U. Dayal and A. Kemper, "Managing Long-Running Queries". *Proc. of the 12th Intl. Conf. on Extending Database Technology: Advances in Database Technology*. Saint Petersburg, Russia. March 23-26, 2009. pp. 132-143.

[22] S. Krompass, H. Kuno, U. Dayal and A. Kemper, "Dynamic Workload Management for Very Large Data Warehouses - Juggling Feathers and Bowling Balls", In *Proc. of 33rd Intl. Conf. on Very Large Data Bases.* University of Vienna, Austria, September 23-27, 2007. pp. 1105-1115.

[23] S. Parekh, K. Rose, J. Hellerstein, S. Lightstone, M. Huras and V. Chang. "Managing the Performance Impact of Administrative Utilities". In *Proc. of Self-Managing Distributed Systems*, Springer Berlin, Heidelberg, February 19, 2004. pp. 130-142.

[24] TPC Homepage. TPC-H benchmark, http://www.tpc.org.

[25] W. Powley, P. Martin and P. Bird, "DBMS Workload Control Using Throttling: Experimental Insights". In *CASCON '08: Proc. of the 2008 Conf. of the Center for*

Advanced Studies on Collaborative Research. Toronto, Canada. October 27-30, 2008. pp. 1-13.

[26] W. Powley, P. Martin, M. Zhang, P. Bird and K. McDonald. "Autonomic Workload Execution Control Using Throttling". *IEEE* 26th International Conference on Data Engineering Workshops (5th International Workshop on Self-Managing Database Systems), Long Beach, CA, USA. March 1-6, 2010.

Appendix A

TPC-H Benchmark

The TPC-H benchmark [24] is a decision support benchmark, developed by the Transaction Processing Performance Council. It's main purpose is to be used to evaluate the performance of decision support systems by executing queries under controlled conditions. This benchmark consists of 22 business oriented queries. The TPC-H database contains 8 tables. The table names and schemas are shown in Figure A. 1. The size of each table depends on scale factor. For example if with a scale factor of 1, the PART table contains 800,000 records and with a scale factor of 5, it will be 4,000,000 records.

Each of the 22 queries is designed to answer a specific business question. For example Query 4 answers the following business question:

The Order Priority Checking Query counts the number of orders ordered in a given quarter of a given year in which at least one lineitem was received by the customer later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order

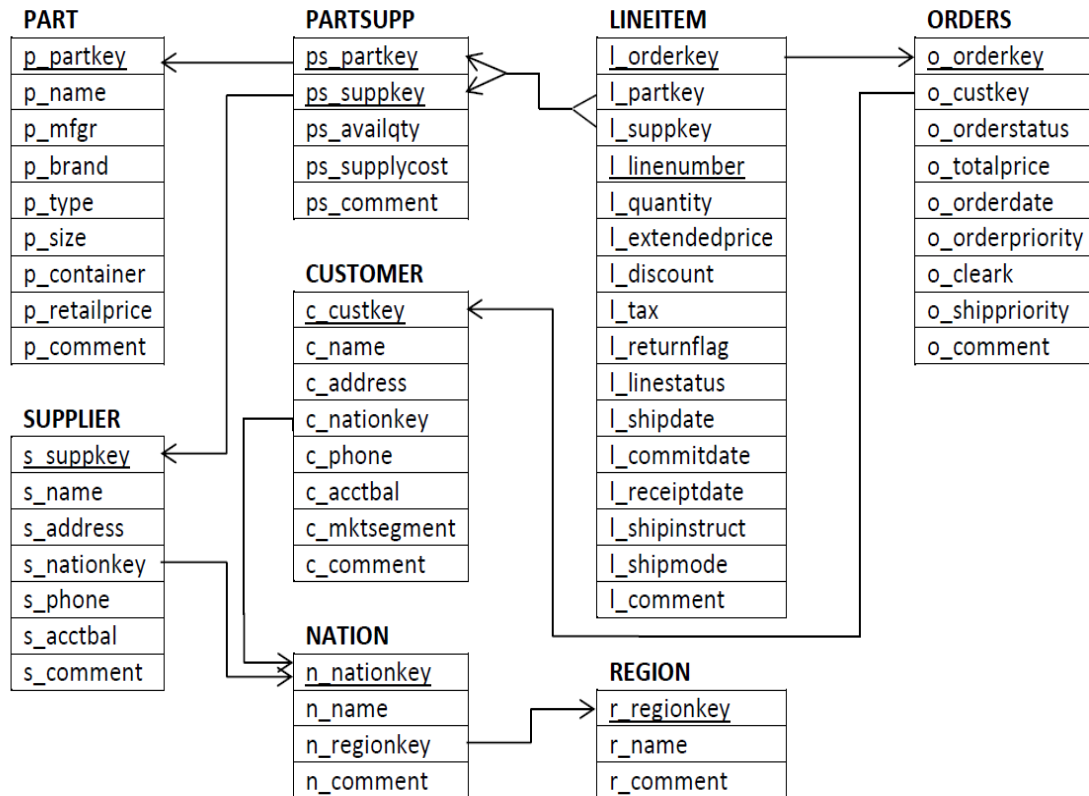

Figure A. 1: TPC-H Schema

Query Q_1:

```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as
sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax))
as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    lineitem
where
```

```
        l_shipdate <= date '1998-12-01' - interval ':1' day
(3)
    group by
        l_returnflag,
        l_linestatus
    order by
        l_returnflag,
        l_linestatus;
```

Query Q_4:

```
    select
        o_orderpriority,
        count(*) as order_count
    from
        orders, lineitem
    where
        o_orderkey = l_orderkey
        and o_orderdate >= date ':1'
        and o_orderdate < date ':1' + interval '3' month
        and exists (
              select * from
                    lineitem
              where
                    l_orderkey = o_orderkey
                    and l_commitdate < l_receiptdate
        )
    group by
        o_orderpriority
    order by
        o_orderpriority;
```

Appendix B

pgbench

pgbench is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, pgbench tests a scenario that is loosely based on TPC-H, involving five SELECT, UPDATE, and INSERT commands per transaction [17]. However based on the type of test the default transaction script can be changed and it is easy to test other cases by writing your own transaction script files. The default TPC-H transaction test requires specific tables to be set up beforehand. pgbench should be invoked with the `-i` (initialize) option to create and populate these tables. Initialization call is like:

pgbench -i [other-options] dbname

Where dbname is the name of the already-created database to test in. pgbench -i creates four tables pgbench_accounts, pgbench_branches, pgbench_history, and pgbench_tellers.

At the default "scale factor" of 1, the tables initially contain this many rows:

table	# of rows
pgbench_branches	1
pgbench_tellers	10
pgbench_accounts	100000
pgbench_history	0

The number of rows can be increased by using the `–s` (scale factor) option. After setting the appropriate set up the benchmark can be run with a command that does not include `–i`, which is

```
pgbench [ options ] dbname
```

The available useful options that we used in generating OLTP workload are –c (number of clients), -t (number of transactions).

Typical output from *pgbench* looks like:

```
transaction type: TPC-H
scaling factor:10
number of clients: 30
number of transactions per client: 2500
number of transactions actually processed: 75000 / 75000
tps: 3554
```

The first four lines report some of the most important parameter settings. The next line reports the number of transactions completed and intended (the latter being just the product of number of clients and number of transactions per client); these will be equal unless the run failed before completion. The last line reports the number of transactions per second.