DBMS Workload Control using Throttling: Experimental Insights

Wendy Powley, Pat Martin
School of Computing, Queen's University, Kingston ON
{wendy, martin}@cs.queensu.ca

Paul Bird
IBM Labs, Toronto, ON
pbird@ca.ibm.com

Abstract

Today's database management systems (DBMSs) are required to handle diverse, mixed workloads and to provide differentiated levels of service to ensure that critical work takes priority. In order to meet these needs, it is necessary for a DBMS to have control over the workload executing in the system. Lower priority workloads should be limited to allow higher priority workloads to complete in a timely fashion. In this paper we examine query throttling techniques as a method of workload control. In our approach, a workload class may be slowed down during execution in order to release system resources that can be used by higher priority workloads. We examine two methods of throttling; constant throttling throughout query execution, and a single interruption in which a query is paused for a period of time. A set of experiments using Postresql 8.1 provides insights regarding the performance of these different throttling techniques under different workload conditions and how they compare to using operating system process priority control as a throttling mechanism.

1. Introduction

Several recent usage trends are forcing DBMSs into a position where they must be able to effectively manage their workload. First, DBMSs have traditionally handled two distinct types of workload; on-line transaction processing (OLTP), characterized by frequent short data lookups or updates, and on-line analytical processing (OLAP) which are typically more complex, longer running and often read-only queries. Today's 24/7 operational requirements mean that it is no longer feasible to limit competition among these workload types by running OLAP queries at "off-peak" times such as evenings or weekends, as such periods are essentially becoming non-existent.

Second, the emerging trend towards server consolidation has led to more diversity in workloads, and increased competition for shared resources between applications executing queries in a single instance of the DBMS. Current workloads may consist of hundreds or thousands of queries running simultaneously. Workloads are no longer simply characterized as an "OLTP workload" or an "OLAP workload" as in the past, but instead consist of a mix of transactional and decision support queries.

Third, in a corporate environment, the demands of customers, management, human resources, marketing and finance, among others, must be met in a timely fashion with the correct business prioritization. These demands require

the provision of differentiated services, placing some workload classes as a higher priority than others. To provide differentiated services, a DBMS must be able to recognize, characterize, and prioritize the workloads presented to it, and to effectively control the execution of the different workloads so that each workload meets its pre-defined goals, or service level agreements. The importance of the workload to the business determines the processing priority regardless of the actual characteristics of the workload or whether it is convenient for the system. Workload management is a challenging task and solutions typically involve low level resource allocation or admission control.

Simultaneously executing multiple workloads on a system often taxes the system resources and can result in the degradation of performance. Figure 1 shows an example of this where the performance of one OLTP-type workload declines when a second OLTP-type workload is introduced to the system (at sample period 30). As shown in Figure 1, the average throughput of the workload executing in isolation (sample periods 0 - 30) is approximately 360 transactions per second. When the second workload is introduced, the throughput of initial workload drops dramatically. In an underutilized system, adjusting the DBMS tuning parameters may improve performance, but in a system that is saturated, the only viable solution is workload control.

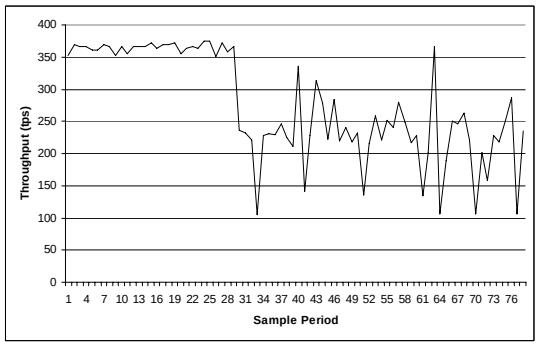

Figure 1: Detrimental effect of running simultaneous workloads.

In this paper we examine throttling as a means of DBMS workload control. Throttling involves slowing down an executing query, or query class, to free resources for other workloads executing in the system. We analyze two approaches to throttling, namely a constant slowing mechanism that institutes short pauses throughout the query and a mechanism that initiates a single interruption of the workload for a period of time. We examine the effects of different pause lengths and timing, the total throttle time and the effects of throttling on workloads of different types with varying degrees of lock contention. We also compare our approach to the use of operating system priority control as a throttling mechanism.

The contributions of the paper include an implementation of the two throttling approaches within PostgreSQL 8.1 and an experimental analysis of their effectiveness. Based on these experiments, we provide observations on appropriate situations in which throttling may be useful and comment on the effectiveness of the different throttling techniques under different conditions.

The remainder of the paper is structured as follows. Section 2 provides an overview of previous work found in the literature related to DBMS workload control mechanisms. In Section 3 we outline our approach which uses query throttling as a means to control the DBMS workload. In Section 4 we present a set of experiments using PostgreSQL 8.1 to illustrate the effects of our approach. Finally, we conclude in Section 5 and provide our ideas for future research directions in Section 6.

2. Related Work

Research related to workload management focuses on two approaches, namely dynamic resource allocation and workload adaptation, including admission control. Brown et al. [3] propose an algorithm that automatically adjusts multi-programming levels and memory allocation to achieve a set of per-class response time goals for a complex workload in DBMSs. Pang et al. [8] propose an algorithm for multi-class query workloads called Priority Adaptation Query Resource Scheduling. They use admission control, allocating memory and assigning priorities based on current resource usage, workload characteristics and performance statistics. Niu et al. [7] present a framework and a prototype query scheduler that manages multiple classes of queries through admission control. A resource allocation plan is derived by maximizing an objective function that encapsulates the

performance goals of the classes and their importance to the business.

Some commercial systems currently support resource-oriented workload control. Teradata Active System Management [2] and IBM® DB2® Query Patroller [4] control the workload presented to a DBMS by using predefined rules based on thresholds of the workload such as multi-programming levels, number of users, and estimated query costs.

Throttling techniques are used by Baryshnikov et al. [1] to reduce the amount of memory used for query compilation in a DBMS in order to improve throughput. In their approach, compilations are blocked at certain periods of their execution until resources become available. Lang et al. [6] use throttling to slow down table scans to keep multiple scans progressing at a similar rate as part of their approach to controlling multiple relational table scans. Few details are provided regarding their throttling technique.

Our approach is based on work by Parekh et al. [9] who apply a throttling technique to limit the impact of on-line database utilities such as backup/restore, data re-organization, or automatic statistic collection on user work. In their approach, a self-imposed sleep is used to slow down, or *throttle*, the utility by a configurable amount. This prototype system is autonomic in that the system self-monitors and reacts according to high level policies to decide when to throttle the utilities and to determine the appropriate amount of throttling. Our plan is to eventually implement an autonomic approach similar to that described by Parekh et al. [9] to throttle portions of the user work as opposed to the administrative utilities. In this paper, we examine the feasibility of throttling DBMS workloads in a controlled environment where throttling is imposed manually.

3. Query Throttling

We examine throttling as a means of workload control in a DBMS. Throttling involves slowing down a specific portion of the work that is currently executing in the system, thus freeing resources for other, higher priority work. Throttling can be instituted on a per-query basis or on an entire workload. In this paper, we assume that throttling is done on a per-workload basis, involving the throttling of each query contained in that workload. Typically, one or more low priority workloads could be throttled to benefit a workload with higher priority.

We employ two methods of throttling which we call ***constant throttle*** and ***interrupt throttle*** respectively. The constant throttle approach involves frequent, very short pauses that are consistent during the query execution, thus slowing the query throughout its execution. The interrupt throttle technique involves a one time delay, or pause, of the query for some specific length of time. Figure 2 depicts the two methods.

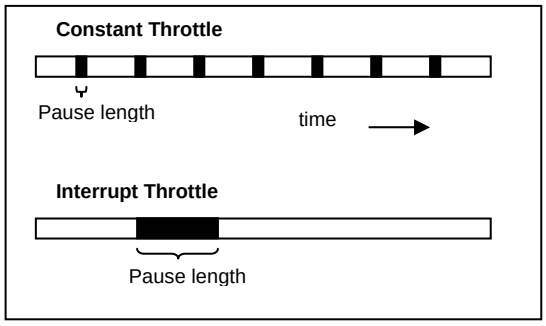

Figure 2: Throttling Techniques

Using the constant throttle technique, the length of each pause during query execution is very short (from a few nanoseconds up to a second) and at regular intervals. The amount of throttling can be controlled by either increasing the frequency of the pauses or by increasing the length of each pause.

In the interrupt technique, each query is paused once and only once for a period of time during the query execution. Pauses are longer than for the constant throttle method (from seconds to possibly minutes). This technique is similar to known admission control techniques except that the control occurs during query execution. Instead of preventing queries from executing, this method allows the queries to begin execution, but then interrupts them for a period of time. Important parameters in this approach are the length of the interruption and the positioning of the pause within a query.

Lock contention is a concern with throttling. Pausing a workload for a significant length of time while it is holding locks on data required by the competing workload defeats the goal of throttling. To account for this, the ability to suspend a throttle is necessary. An advanced implementation of our throttling approach

included this capability. When a throttled workload is holding a lock that a competing workload requires, the throttling is temporarily suspended until the lock is released. At this point, the throttle is reinstated. Depending on the amount of lock contention, a throttled workload may spend a significant amount of its execution time in a non-throttled state.

3.1 Implementation

To examine the effectiveness of the throttling approaches, we modified PostgreSQL 8.1 running on Windows® XP to support throttling. Our general architecture is illustrated by Figure 3.

For our experiments, each workload class is handled by a separate PostgreSQL backend process, the PostgreSQL process that handles query execution. Having a separate backend for each workload class allows for simple characterization and prioritization of workloads. Information about each backend process is maintained in shared memory. We augment this structure to include throttling information such as that shown in Figure 3. Setting "Throttle" to "ON" for a particular backend with this structure initiates throttling for the workload class handled by the backend.

We employ the PostgreSQL interrupt checker routine to implement the workload control. The interrupt checker runs with low overhead, and is called repeatedly throughout the execution of each query to check for interrupts. Throttling is enforced using the nanosleep(nanoseconds) function called from within the interrupt checking routine. A counter local to each query is incremented each time the interrupt handler is called, providing a mechanism by which to govern the amount, timing, and length of the throttling on a per-workload (or even a per-query) basis.

Suspension of the throttling approach is implemented within the lock handling code. When a query is placed on the lock wait queue, it determines which backend is holding the requested lock. If the backend that holds the requested lock is currently throttled, then the backend's "Throttle" value is toggled to "OFF", thus suspending the throttling. Once the lock is released and granted to the waiting backend, a message is sent to the previously throttled backend, which resets its "Throttle" value to "ON".

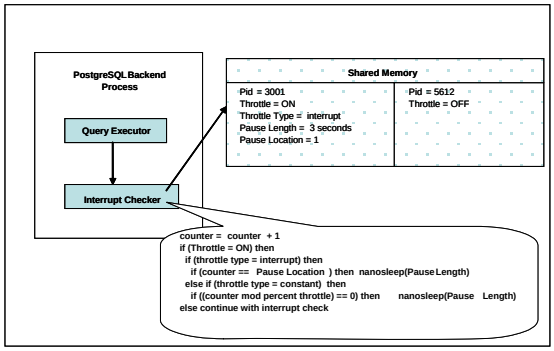

Figure 3: Implementation in PostgreSQL 8.1

4. Experimental Evaluation

A series of experiments were performed to examine the effectiveness of throttling on both the important workload and the throttled workloads. We examined the effects of varying the degree of throttling, that is, the total amount that we throttle a workload. For the constant throttle we varied the length of the pauses to examine the effects of pause length on the workload performance. For the interrupt throttle, we examined the effects of instituting the pause at various positions in the workload. The experiments were run under different workload conditions including different workload types (OLTP/OLAP) with varying degrees of lock contention. We also compare throttling to that of assigning priorities to the workloads using operating system priorities.

All experiments were run on an Intel™ Pentium™ 4 2.0 GHz machine with 512 MB of RAM with the data on a single disk. Each set of experiments was run multiple times and the average throughputs and response times were calculated and reported. The default PostgreSQL configuration was used in all experiments. Given the limited hardware environment and the modest default PostgreSQL configuration, our system was easily saturated by minimal workloads, which is the desired system state for our experimental environment.

4.1 Workloads

Two tables, ACCOUNTS and LINEITEM, were used for our experiments. The ACCOUNTS table was part of the pgbench database which was included with the PostgreSQL distribution. The LINEITEM table was from the TPC-H database

specifications [10]. The ACCOUNTS table consisted of 10 million tuples and the LINEITEM table contained 3 million tuples. Several simple workloads were used for our experiments and are described as follows:

OLTP Workload – The pgbench workload was used for the OLTP workload. Ten clients issued random queries accessing the ACCOUNTS table. The average throughput for this workload running alone under the default PostgreSQL 8.1 configuration on our test-bed environment was approximately 400 transactions per second (tps) in select-only mode (no updates).

OLAP Lineitem – This workload consisted of a single query involving an aggregation over the entire LINEITEM table. Running alone in our test-bed environment, this query took, on average, 12 seconds to complete.

OLAP Accounts – This workload consisted of a single query involving an aggregation over the entire ACCOUNTS table. Running alone in our test-bed environment, this query took, on average, 40 seconds to complete.

Lock Accounts – This workload consisted of a select only workload on the ACCOUNTS table. One of the queries explicitly holds a lock for approximately 5 seconds. This workload is a mixture of the "lock" transactions as well as simple select queries that do not require locks.

4.2 A Comparison of Constant Throttle and Interrupt Throttle (No Data Sharing)

The first set of experiments compared the performance of the constant throttle and the interrupt throttle approaches as well as investigated a number of factors related to the two different throttling approaches. The workloads used for these experiments access different tables, thus there is no sharing of data and, therefore, no lock contention.

For the constant throttling approach, we examined the effects of varying the length of the pause as well as the impact of increased amounts of throttling under varying workload conditions. We tested the granularity of the throttle length to determine the effects of using a very short throttle time (.01 seconds), but pausing more frequently during the query execution, versus a longer pause time (1 second), but pausing less frequently. We also varied the length of the total throttle time, that is, the total amount of time the throttled query spends waiting.

The important parameters in the interrupt throttle technique are the pause length (which, in this case, equals the total amount of throttle time) and the positioning of the pause within the query, that is, where in the query the pause occurs. Thus, experiments varied the pause length and the location of the pause. Approximate locations of the pause included the start of the query, the middle of the query, and the end of the query.

The timing of the pause was implemented using the incremental counter associated with each query which was incremented each time the interrupt check routine was called during query execution. The value of the counter was used to position the pause. The number of interrupt checks, which we denote here as I_t, was constant (and known in advance) for a particular query. To pause mid-query, the nanosecond routine was called when the counter reached $I_t/2$. A pause at the start of a query occurred during the first interrupt check whereas a pause at the end of the query occurred when the counter reached I_t.

Each experiment was run under two different workload conditions; OLTP/OLAP and OLAP/OLAP. In the OLTP/OLAP case, the OLTP workload was the higher priority workload and the OLAP Lineitem workload was the competing workload that was throttled. In the OLAP/OLAP case, the OLAP Accounts workload was the high priority workload, and the OLAP Lineitem workload was throttled.

The low priority OLAP Lineitem workload consisted of a query that ran alone in approximately 12 seconds. We ran experiments that throttled the OLAP Lineitem query for a total of 3, 20, 30, 50, 100, 150, 200, 250 and 300 seconds to examine the effects of increased throttle time on the more important workload.

OLTP/OLAP Workloads

The results of the constant throttle experiments with competing OLTP/OLAP workloads are shown in Figure 4 and those of the interrupt throttle approach are shown in Figure 5. In each case, the OLAP workload is the lower priority, throttled workload. The throughput (tps) for the

OLTP workload is shown on the left y-axis of the graph whereas the response time (seconds) for the OLAP query is shown on the right y-axis The first point in each graph shows the response time (OLAP workload) or the throughput (OLTP workload) of each workload run independently, that is, without a competing workload and without throttling. The second point shows the two workloads executing simultaneously with no throttling.

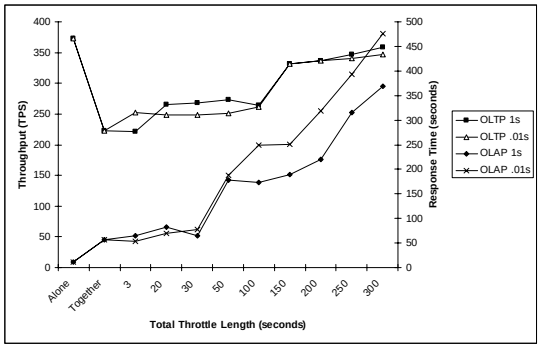

Figure 4: Constant Throttle, OLTP/OLAP

The points labeled 3-300 along the x-axis of the graphs represent the total throttle time (in seconds) for the OLAP Lineitem workload. For example, at x=3, the OLAP Lineitem query was throttled for a total of 3 seconds during the query execution. In the case of constant throttle, with the 1 second throttle length, the query was paused 3 times during execution, each pause lasting 1 second, and the delays were (approximately) evenly spaced throughout the query execution. For the .01 second pause length, the OLAP Lineitem query was paused 300 times, each pause lasting .01 seconds, with the delays (approximately) evenly spaced throughout the query execution. In the case of the interrupt throttle, the query was paused once, either at the start, middle or end of the query for a total of 3 seconds.

With the constant throttle approach, we note that although the performance of the OLTP workload improves with increased throttling, it is not until we reach an OLAP throttle time of 100 seconds that we see any reasonable improvement. The OLTP performance actually remains quite steady with OLAP throttling times of 20 to 100 seconds of throttling followed by a large increase in throughput between 100 and 150 seconds. This trend is consistent when using either a one second pause or a .01 second pause. We also note that with 50 seconds of throttle time, the response time for the OLAP workload increases dramatically. It appears from our results that the shorter pause length for the constant throttle technique is more detrimental to the OLAP workload than when a longer pause length is used, however, the pause length has no effect on the performance of the OLTP workload.

In contrast, with the interrupt throttle approach, we observe a steady increase in the OLTP performance with increased throttling of the OLAP workload, especially when the pause occurs at the beginning of the OLAP query. Overall, OLTP performance is better when the OLAP query is paused at the beginning of the query, before it has acquired resources.

With both the constant throttle and the interrupt throttle, a maximum performance of 350 tps is reached with increased amounts of throttling, however, performance improves more steadily and at a slightly faster pace using the interrupt throttle approach when the pause is instituted at the beginning of the query.

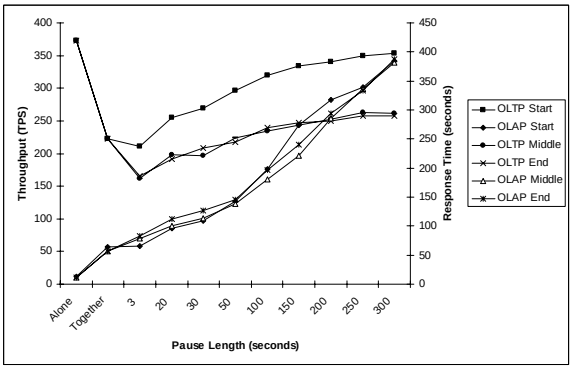

Figure 5: Interrupt Throttle, OLTP/OLAP

OLAP/OLAP Workloads

The results of the constant throttle and the interrupt throttle with competing OLAP workloads are shown in Figure 6 and Figure 7, respectively. The OLAP Lineitem and OLAP Accounts workloads were run simultaneously with the OLAP Lineitem workload throttled. The y-axis shows the average response time (in seconds) for each workload whereas the x-axis shows the total throttle length (in seconds).

The average response time for each query running in isolation is shown as the first point in

each graph. The average response time for each workload when the two are running simultaneously with no throttling is illustrated by the second point in the graph. The remaining points indicate the average response time for each workload when the OLAP Lineitem is throttled by a total of x seconds. The constant graph shows two cases; a pause length of 1 second and a pause length of .01 seconds. The graph for the interrupt throttle case illustrates 3 cases; the pause instituted at the beginning, middle or end of each query.

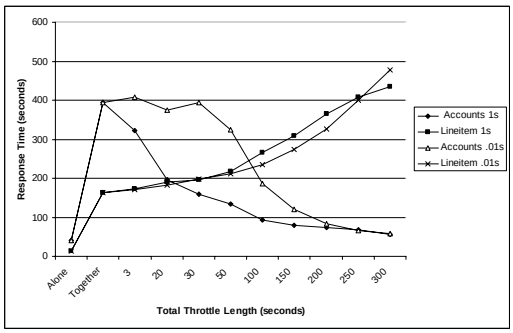

Figure 6: Constant Throttle, OLAP/OLAP

Based on Figure 6, we note that there is a more rapid performance improvement for the important workload (as seen by the decline in response time) when the one second pause mechanism is used with the constant throttle technique than when more frequent, shorter pauses are used. In fact, in this case, using a .01 second pause length, it is not until a total throttle length of greater than 30 seconds occurs that we observe significant improvement in the response time for the OLAP Accounts workload. Conversely, with the 1 second pause length, we see an immediate improvement in the OLAP Accounts response time, even with a total throttle time of just 3 seconds. Unlike the OLTP/OLAP case, the pause length did not seem to have an effect on the performance of the throttled workload.

In the interrupt throttle case, we observed similar results to the constant throttle approach using one second pauses. In this case, with even short pause lengths, as little as 3 seconds, we see some improvement, and with a 20 second pause length, the response time is cut in half. As can be seen by Figure 7, the location of the pause appears to be irrelevant in the OLAP/OLAP case.

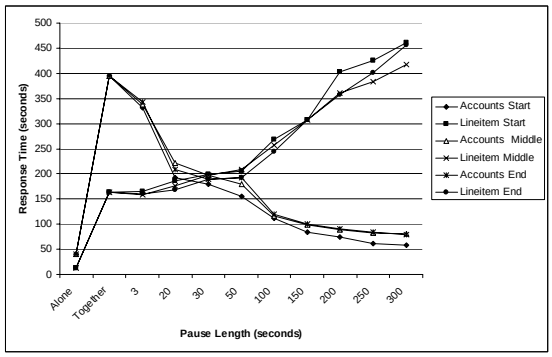

Figure 7: Interrupt Throttle, OLAP/OLAP

4.2.1 Effects on Non-throttled Workloads

As we have seen by the experiments thus far, throttling the low priority OLAP workload generally leads to an overall increase in performance in the high priority OLTP workload, however, it is interesting to look at the more subtle effects of throttling a competing workload on the high priority workload.

Figure 8 shows the variation in the performance of the OLTP workload when a constant throttle is imposed on the OLAP Lineitem workload. This case shows a constant throttle using .01 second pauses for a total of 20 seconds (a total of 2000 pauses). In this graph, each sampling period represents an average of 1500 OLTP queries that have executed while the OLAP query runs simultaneously.

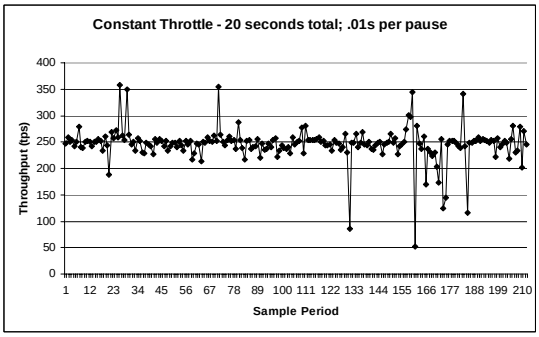

Figure 8: Constant Throttle - .01 second pause length, 20 seconds total throttle

Figure 9 shows the effects on the OLTP workload when the OLAP workload is throttled

for 20 seconds using a pause duration of 1 second (a total of 20 pauses). Figure 10 shows the effect of an interrupt throttle of the OLAP workload on the OLTP workload. In this case, the OLAP workload is paused only once for 20 seconds.

From these figures, we see that although the average OLTP throughput is more or less the same (approximately 250 tps) using either constant or interrupt throttling, there is considerable variation in the OLTP performance with longer pause lengths. The standard deviations are 31.8, 49.4, 99.6 for the three samples, respectively. We observe surges in the OLTP workload during the pauses followed by a drop in performance when the throttled workload is allowed to run. The performance of the OLTP workload is much more consistent when constant throttling is used to slow the OLAP workload with shorter, more frequent pauses. Although not shown here, this trend is consistent for the OLAP/OLAP workloads.

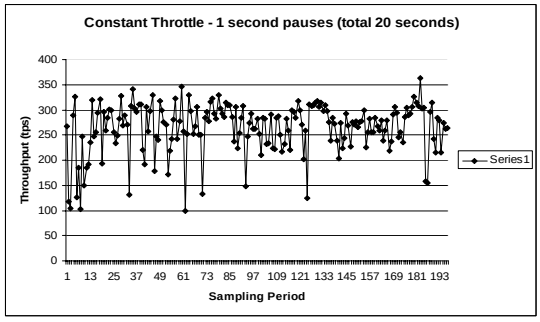

Figure 9: Constant Throttle - 1 second pause length, 20 seconds total throttle

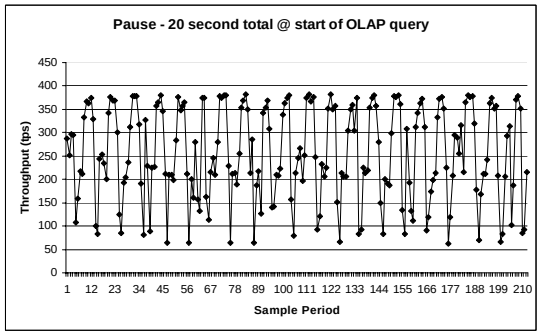

Figure 10: Interrupt Throttle – 20 second total pause at the start of the query

4.2.2 Overhead

Our constant throttle approach called the nanosleep(nanoseconds) function to throttle the query by some number of nanoseconds, possibly thousands of times during query execution. In order to measure the overhead involved in making these calls, we ran the OLAP Lineitem workload with up to three million calls to nanosleep(0). With zero sleep time, the delay in the queries is only due to the overhead of calling the function.

The results, shown in Figure 11, indicated that although significant overhead was incurred with increased numbers of invocations of the nanosleep() function, it was negligible given the number of calls made by the throttling technique. In our experiments, the maximum number of calls during the execution of the OLAP Lineitem query was 30,000, which, using the constant throttle technique with .01 second pauses introduced a delay of 300 seconds. Figure 11 shows that the response time remains unchanged from 0 – 30000 calls. We conclude that the overhead of our approach is insignificant with "reasonable" amounts of throttling. The cumulative effects of overhead, however, must be taken into consideration.

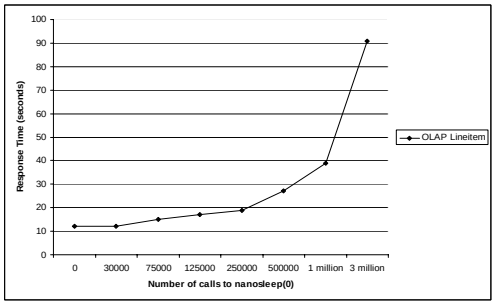

Figure 11: Overhead Results

4.3 Constant Throttle with Lock Contention (Data Sharing)

So far our experiments have involved select-only workloads, thus involving no data sharing. Lock contention, however, plays an important role in the use of throttling for workload control. Throttling a workload while it holds locks required by a competing workload degrades the performance of both workloads, thus defeating the goal of throttling. The basic constant throttle approach institutes a constant slowing of the

workload by pausing at constant intervals while the workload is running. Of course, this may result in locks being held for a longer period of time than is necessary. If competing workloads are not waiting on these locks, this presents no problem. However, if the locked objects are needed by the more important workloads, throttling will slow not only the progress of the less important workload, but that of the important workload as well. To remedy this situation, the constant throttle approach was augmented with logic such that, if a competing workload is waiting on a lock held by a throttled workload, the throttling is suspended until the lock is released.

In this experiment we examined the effects of throttling when there is potential for lock contention. We compared the basic constant throttle approach (No Logic) with the throttle approach augmented with logic to suspend the throttling (Logic).

Two select-only workloads accessing the ACCOUNTS table were run simultaneously; the OLTP workload used for the previous experiments and the Lock Accounts workload described in Section 4.1.1. The Lock Accounts workload (referred to hereafter as simply the "Lock" workload) is comprised of simple select queries on the ACCOUNTS table interspersed by queries that request and hold an exclusive lock on the ACCOUNTS table for approximately 5 seconds. The amount of potential lock contention was varied based on the number of locking transactions in the workload. We show results for 25, 50 and 75 percent lock contention, meaning that there was potential for lock contention 25, 50 or 75 percent of the time that the workloads were running. The Lock workload was considered to be the less important workload and hence was the workload that was throttled in the experiments. The throttling approach was a constant throttle with .01 second pause lengths with a total pause length of 3 seconds. Other configurations yielded similar results.

The results are shown in Figure 12. We show only the performance of the important workload, that is, the non-throttled workload. Each case shows the performance of the important workload when the competing Lock workload is not throttled (No Throttle), the performance when the basic throttling approach is applied (No Logic) and also the performance of the important workload when the throttling of the Lock workload is suspended each time the important workload is waiting on a lock that is being held by the Lock workload (Logic).

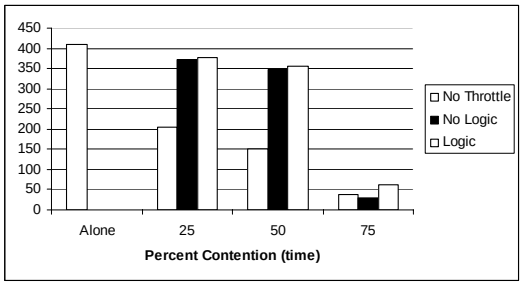

Figure 12: Constant Throttle with Lock Contention

We see from Figure 12 that throttling the less important workload improved the performance of the more important workload when the amount of lock contention is low. Suspending the throttling when the competing workload is waiting on a lock, in this case, resulted in only a very slight improvement. With increasing lock contention, however, throttling the Lock workload with no throttle suspension resulted in reduced performance for the important workload and only minor improvements with throttle suspension.

4.4 A Comparison of Constant Throttle and Operating System Priority Control

In this set of experiments we compared our constant throttling approach to the use of operating system (OS) process priority control to slow down the less important workload. The Cygwin [5] "nice" command was used to lower the priority of the PostgreSQL process that is servicing the less important workload. We compared different "nice" values (5, 10, 15, 19 where higher values represent lower priority) and found that the nice factor, because we are running only two workloads, had little to no effect. We therefore report results only for a nice factor of 5.

In the first experiment, we ran the OLAP/OLTP workloads with no lock contention using the operating system priority (the nice command) to throttle the OLAP workload. The results are shown in Figure 13. This figure shows the performance (transactions per second for the OLTP workload, response time for the OLAP

workload) of the workloads running separately (ideal performance), together (conflicting) and with the OLAP workload throttled by lowering the OS priority of the PostgreSQL backend handling this workload. The performance of the OLTP workload improved dramatically when the OLAP workload was throttled using a reduced OS priority level. The OLTP workload reaches on average 314 transactions per second.

Comparing these results to those of our constant and interrupt throttling approaches, all resulted in an increase in performance for the important (OLTP) workload. With our throttling approach, however, the OLTP workload was able to reach a maximum of 350 transactions per second as compared to the maximum 314 transactions per second achieved by the OS priority approach.

Both the OS priority control technique and the throttling approaches incurred a significant performance decrement to the OLAP workload. Using the OS priority control, the resulting OLAP workload performance was, on average, 500 seconds whereas with the throttling approaches it averaged approximately 400 seconds for best overall OLTP performance. In fact, to reach the same level of OLTP performance achieved by the OS priority technique (that is, 314 tps), the throttling techniques slowed the OLAP workload to less than 200 seconds (as compared to 500 seconds for the OS priority technique). Thus the throttling approaches allowed the less important workload to continue at a faster pace than with the OS priority approach while, at the same time, improving the OLTP performance.

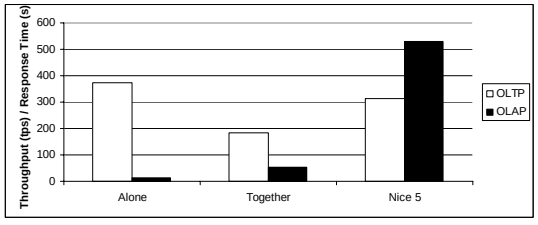

Figure 13: OLAP/OLTP Using Operating System Priority Control (no lock contention)

Figure 14 shows the results of throttling using OS priority control with competing OLAP workloads. In this case, the Lineitem workload was throttled using the "nice" command. Using this technique, the response time of the important workload improves to, on average, 297 seconds. As shown in Figure 6 and Figure 7, using the constant or interrupt throttling approaches, the response time for the important workload can be reduced to less than 200 seconds with little impact on the throttled workload. Therefore, although both our throttling approaches and the OS priority control improve the performance of the important workload class, our throttling approaches show greater improvement with the added benefit of less impact to the throttled workload.

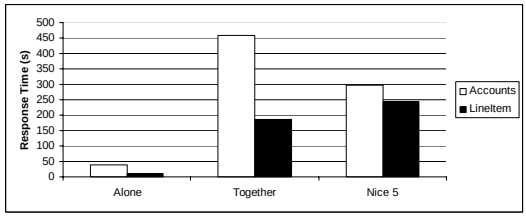

Figure 14: OLAP/OLAP Using Operating System Priority Control (no lock contention)

In the final experiment, we introduced lock contention into the workloads. The same workloads were used as in Section 4.4. The amount of potential lock contention was varied based on the number of locking transactions in the workload. We show results for 25, 50 and 75 percent lock contention. The Lock workload was considered to be the less important workload and hence was the throttled workload.

The experiments compared the best results obtained by our throttling techniques with the OS priority control technique. We show only the performance of the important workload, that is, the OLTP workload, in Figure 15.

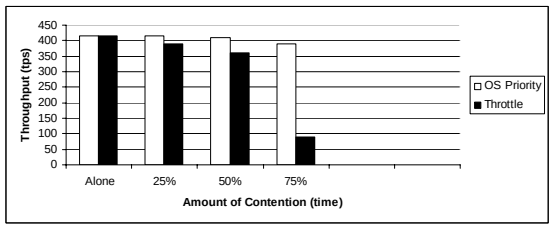

Figure 15: OS Priority Control vs Constant Throttle with Lock Contention

We see that as the amount of lock contention increased, using the OS priority control was clearly superior over our throttling approaches. However, the reason for this phenomenon lies in the implementation of the throttling approach. In

the throttling technique, the timing of the throttle is based on the number of times the interrupt handler routine is called during the execution of the workload. The timing of the pauses is determined by the total number of interrupt checks for the workload and the pauses are inserted when the incremental counter reaches a certain number. Due to the fact that longer queries have more interrupt checks than shorter queries, the pauses are more likely to occur during these longer queries. The Lock workload consists of short queries interspersed with longer queries that hold exclusive locks. Obviously more interrupt checks are done throughout the longer query. Because of this, the throttling is more likely to occur during the queries that hold the locks, thus delaying not only the throttled workload, but the important workload that is waiting on the lock. In the OS Priority control technique, the throttling is more consistent throughout the workload, slowing both the simple select queries as well as those queries bthat hold the locks.

5. Conclusions

Competing workloads in a DBMS can have a detrimental effect on performance. In order to provide differentiated levels of service and to ensure that high priority workloads are favoured over lower priority workloads, some degree of workload control is required. Slowing a workload class by way of constant throttling or by interrupting the queries is suggested as a means of controlling the amount of work executing in a system simultaneously. We provide a set of experiments to show how these approaches perform under a variety of workloads and conditions.

The results of our experiments show that slowing down a workload class by throttling or pausing can result in improved performance for a competing class. In most cases, however, this benefit is achieved only with a significant sacrifice to the performance of the workload that is throttled. Our experiments showed that the pause length was significant in determining the net effect on the throttled workload. Longer pause lengths, in the case of the constant throttle, were less detrimental to the throttled workload than shorter, more frequent pauses.

The need to slow a workload down significantly may be acceptable in some conditions, such as when the low priority workload is “best effort”, or has no service level agreements that must be satisfied. Delaying such a workload allows the work to continue, just at a slower pace. With other workload control mechanisms, this “best effort” class may be prevented from entering the system for long periods of time when there is a high volume of high priority work.

The variation in the performance of the high priority workload is substantial when the low priority workload is slowed using the interrupt throttle method. In contrast, this variation is lower when the constant throttling technique is used. Therefore, the importance of performance consistency for the high priority workload may influence the mechanism used to slow down the low priority workload. If improving overall throughput is the only goal, then either constant or interrupt throttle is suitable. However, if consistent performance for the high priority workload is crucial, then constant throttling techniques are more advisable.

Our experiments suggest that, for the interrupt throttle, in the case of high priority OLTP workload and low priority OLAP workload, pausing the OLAP query at the start of the query before it begins execution is superior to pausing the query at the middle or the end of the query.

The amount of expected lock contention among competing workloads will have an impact on the effectiveness of throttling the less important workloads. When the amount of lock contention is high, measures must be taken to ensure that throttled workloads do not hold locks for unnecessary amounts of time. The throttling technique must take this into consideration; otherwise throttling such workloads should be avoided.

The constant throttle technique was compared to the use of operating system process priority to control a workload. In most cases, the throttling techniques resulted in greater improvement to the important workload with less impact on the throttled workload. The exception to this claim was when there was lock contention. In this case, the OS priority technique performed better due to the fact that it was more consistent in slowing the entire workload as opposed to our approach which had a tendency to throttle more often during the longer queries that were holding the locks.

6. Future Work

We have provided some insights into the advantages and disadvantages of using constant throttle and interrupt throttle on different combinations of workload types and also compared these approaches to that of using process priority control. These are very preliminary results to examine the feasibility of this approach. It is clear from our findings that workload throttling for DBMSs is a promising research area.

In the current work, we have a priori knowledge regarding query response times, making it easy to control the throttling by placing constant pauses throughout the workload. In reality, however, the system does not have this knowledge about the workloads being handled. We will investigate methods for controlling the amount and timing of throttling for ad hoc queries.

Our long-range plan is to implement workload class throttling as an autonomic feature of a DBMS. As an autonomic workload control mechanism, the system will react to high level policies, and will adjust the mix of workloads currently executing in the system without human intervention. The system will determine which workload classes it should slow down, choose a mechanism for throttling the workload, and determine the amount and length of throttling required.

Acknowledgements

The authors wish to thank Neil Conway of the PostgreSQL development team for his valuable insights regarding our implementation and IBM, NSERC and OCE-CCIT for their financial support.

About the Authors

Patrick Martin is a Professor in the School of Computing at Queen's University. He holds a BSc from the University of Toronto, MSc from Queen's University and a PhD from the University of Toronto. He is also a Faculty Fellow with the IBM Centre for Advanced Studies in Toronto, ON. His research interests include database system performance, Web services and autonomic computing systems.

Wendy Powley is a Research Associate and Adjunct Lecturer in the school of Computing at Queen's University. She holds a BA in psychology, a BEd, and an MSc in Computer Science from Queen's University. Her research interests include database systems, web services and autonomic computing.

Paul Bird is a Senior Technical Staff Member in the DB2 Database for Linux®, UNIX®, and Windows® Development organization within the Information Management group of IBM. His areas of interest include workload management, security, monitoring, and general SQL processing.

References

[1] B. Baryshnikov, C. Clinciu, C. Cunningham, L. Giakoumakis, S. Oks, and S. Stefani. "Managing Query Compilation Memory Consumption to Improve DBMS Throughput", 3rd Biennial Conference on Innovative Data Systems Research (CIDR), January 7-10, 2007, Asilomar, California, USA.

[2] D. P. Brown, A. Richards, R. Zeehandelaar, and D. Galeazzi. "Teradata Active System Management", http://www.teradata.com/t/page/145613/index.html.

[3] K. P. Brown, M. Mehta, M. J. Carey, and M. Livny. "Towards Automated Performance Tuning For Complex Workloads", Proceedings of the 20th Very Large Data Base Conference, Santiago, Chile, 1994.

[4] IBM Corporation. DB2 Query Patroller Guide: Installation, Administration, and Usage, 2003.

[5] CYGWIN http://www.cygwin.com/.

[6] C. Lang, S. Padmanabhan and K. Wong. "Increasing Buffer-Locality for Multiple Relational Table Scans through Grouping and Throttling", Proceedings of the 23rd International Conference on Data Engineering (ICDE), Istanbul, Turkey, April 17-20, 2007.

[7] B. Niu, P. Martin, W. Powley, R. Horman and P. Bird. "Workload Adaptation in Autonomic DBMSs". Proceedings of CASCON 2006, Toronto, October 16 – 19, 2003.

[8] H. Pang, M. J. Carey, and M. Livny. "Multiclass Query Scheduling in Real-Time Database

Systems", IEEE Transaction on Knowledge and Data Engineering, Vol. 7, No. 4, Aug. 1995.

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

[10] Transaction Processing Performance Council, TPC-H Specifications, http://www.tpc.org.

Trademarks