BI Batch Manager: A System for Managing Batch Workloads on Enterprise Data-Warehouses

Abhay Mehta
HP Labs

Chetan Gupta
HP Labs

Umeshwar Dayal
HP Labs

abhay.mehta@hp.com

chetan.gupta@hp.com

umeshwar.dayal@hp.com

ABSTRACT

Modern enterprise data warehouses have complex workloads that are notoriously difficult to manage. An important problem in workload management is to run these complex workloads 'optimally'. Traditionally this problem has been studied in the OLTP (Online Transaction Processing) context where MPL (Multi Programming Level) is used as a knob to achieve optimality. However, MPL is a tricky knob in a BI (Business Intelligence) scenario, since a low MPL can easily result in underload and a high MPL can easily result in overload and 'thrashing'.

In this work we present BI Batch Manager, a workload management system to run batches of queries 'optimally' on an Enterprise Data Warehouse (EDW). It is comprised of three components: an admission control component, a scheduler and an execution control component. In order to automatically avoid underload and overload, we introduce a novel execution control mechanism, PGM (Priority Gradient Multiprogramming). In PGM, a priority gradient is created for the workload, with each query running at a distinctly different priority level. We demonstrate that this stabilizes the execution of a workload across a wide operating range. We use memory as the controlling factor for our admission control policy – admitting batches of queries such that their memory requirement equals the available memory on the system. Our scheduling policy of largest memory query as the highest priority query further stabilizes the execution.

We validate our BI Batch Manager using varying workloads on a commercial, enterprise class DBMS. We show that it effectively avoids underload and overload (thrashing) and can automatically run BI workloads with 'optimal' performance.

1. Introduction

Many organizations are creating and deploying Enterprise Data Warehouses (EDW) to serve as the single source of corporate data for business intelligence. Not only are these enterprise data warehouses expected to scale to enormous data volumes (hundreds of terabytes), but they are also expected to perform well under increasingly complex workloads, consisting of batch and incremental data loads, batch reports and complex ad hoc queries. A key challenge for an EDW is to manage complex workloads to meet stringent performance objectives: for instance, batch load tasks are required to finish within a specified time window before reports or queries can be serviced, batch reports may issue thousands of "roll up" (aggregation) queries that are required to complete within a specified time window; ad hoc queries may have user-specified deadlines and priorities etc. Workload management is the problem of admitting, scheduling and executing queries and allocating resources so as to meet these performance objectives.

A common use of an enterprise data warehouse is to run a workload consisting of a batch of queries. The objective in this case is to minimize the response time of a workload. The batch problem is an important problem since an EDW (Enterprise Data Warehouse) system often spends a considerable fraction of its time running batch workloads such as rollups and reports. Typically, these batch workloads are distinct from the more interactive, ad hoc queries submitted by a user. Our focus in this paper is on the batch workloads. The trend is towards even larger queries and batches as data mining and predictive BI (Business Intelligence) reports are increasingly becoming central activities for a large data warehouse.

The response time of a batch workload running on a system depends on many things: the number and type of queries, the system configuration, the number of concurrent streams of queries running on the system etc. One metric of measuring the response time of a workload is throughput. The throughput is measured in queries completed in a unit time. Throughput has been extensively studied in the literature by various communities. It is important to note that in the context of a batch of queries, the individual response time of a query is not important. Rather what is important is the overall response time for a batch of queries and it is this problem that we focus on in this work.

A common way of looking at throughput is by means of throughput curves where the throughput is plotted against the 'load' on the system. In the case of a DBMS the load is usually measured in number of queries running concurrently on the system. This number is known as the multiprogramming level or MPL. MPL is also typically used to control the load on the system. In Figure 1 we have plotted the 'typical' throughput curves of two different hypothetical workloads: A workload consisting of several large, resource intensive queries ('large
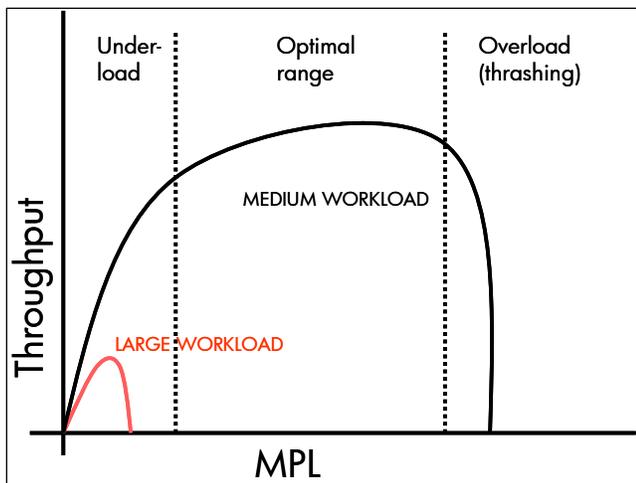
Figure 1: Sample Throughput Curves

workload') and another workload consisting of several medium queries ('medium workload'). The x-axis is the multiprogramming level or MPL and the y-axis is throughput. Both throughput curves can be divided into three regions: (i) Underload (where, by increasing the MPL, a higher throughput can be achieved) (ii) Optimal load (also known as saturation, where by increasing the MPL, there is no significant change in throughput) (iii) Overload (or 'thrashing', where increasing the MPL results in significantly lower throughputs). The goal of a workload manager is to keep the execution of a workload within the optimal load region.

When a user first confronts a new workload the precise shape of the throughput curve is unknown to him/her and the user has to determine the MPL at which to execute the workload. Typically the user does not want to be on the left part of the curve since increasing the MPL can lead to an increase in throughput. But as the MPL is increased there is a danger of entering the overload region where higher MPLs mean a significantly lower throughput. At the boundary between the optimal region and the overload region (which we will call the 'right knee'), increasing the MPL by even one, can cause severe performance deterioration rather than a gradual decline in performance.

The problem is further compounded in the enterprise data warehouse space, by the fact that a typical Business Intelligence (BI) workload can fluctuate rapidly between long, resource intensive queries, and short, less intensive queries. As time progresses, the system can experience a different mix of queries and thus, to be in the optimal region, an EDW requires a different optimal setting of MPL as the workload changes. This dynamically changing nature of the optimal MPL setting makes it very challenging, if not impossible for a human (or a system) to keep the workload in an optimal region by adjusting the MPL setting.

In this paper, we introduce BI Batch Manager, which is a database workload management system for running batches of queries while protecting against both underload and overload and keeping the system in the optimal region.

Our main contributions in this work are the following: (i) We have created a new way of executing queries, PGM, Priority Gradient Multiprogramming, that stabilizes the system over a wide operating range. (ii) Our scheduling algorithm further

enhances the stability of execution. (iii) We have created a mechanism for using memory as a basis for admission control in EDWs. There has been a significant amount of work in the OS community on memory and thrashing. We discuss that in the related works section and we build upon that work in our solution.

The rest of the paper is as follows: In section 2 we present the related work. In section 3 we give a brief overview of the BI Batch Manager. Sections 4, 5, 6 present the execution control, admission control, and scheduler respectively. We present our results and a general discussion in section 7. Finally we conclude with section 8.

2. Related Work

The related work falls into three areas: thrashing control in operating systems; creative memory management in DBMSs; feedback control of workloads.

Multiprogramming was invented in the 1950s. The basic idea was that if a job was waiting for an I/O request to complete, the CPU could process another job in the meanwhile. This would increase the throughput of the number of jobs being processed by the system. Then, in the 1960s, the concept of Virtual Memory (VM) was introduced. Multiprogramming combined with VM enabled higher throughputs, but also created the potential for a system to 'thrash', where more time is spent replacing pages in physical memory, and less time is available for the actual processing of those data pages. The problem of thrashing is inherent in all multiprogramming VM systems, and we continue to this day, to creatively work around this problem.

The problem of oversubscription of memory, the primary cause of thrashing has been studied extensively since the 1960s. The techniques are very often admission control in one form or the other.

A very well known solution to over subscription of memory is the Working Set model [DENN68-WS] [DENN80] . The working set model is based on the assumption of locality. The basic idea is to examine the last *n* page references. The set of pages in the last *n* page references constitutes the working set of the process. A process is not allowed to take a page from another process' working set, and a new process is only introduced if there's enough free memory to accommodate its (predicted) working set. Thus, at its core, the working set mechanism is a feedforward mechanism that prevents problems from occurring.

The Working Set model has a few drawbacks: a process' working set is unknown at the time it is launched; a process' locality can change suddenly and drastically; additional hardware may be required to keep track of a process' working set. For instance, If the working set is overestimated, memory may be under-utilized resulting in sub-optimal performance. Similarly, if the working set for a process is underestimated, it will incur a high cost of page faults, and thus sub-optimal performance. Since, in a working set model, a process can not take memory away from another process, it results in a local page replacement policy. Local page replacement policies can result in serious inefficiencies because overestimation and underestimation errors add together instead of canceling out. Some of these drawbacks have been addressed by [CARR81] [DENN80] [RODR73] .

Several heuristics have been proposed for doing admission control

either by explicitly controlling the MPL or otherwsie. These include the Knee Criterion, the L=S criterion, the Page Fault Frequency algorithm, and the 50% rule [DENN76]. However, thrashing is still an unsolved problem in operating systems and work continues in this area [JIAN02] .

Our focus, in this paper, is in preventing thrashing in a database. We benefit from the existing technologies that are built into the operating system to control overload. We use the characteristics of databases to control overload in the database. The most important characteristic is that a database query has a plan and therefore its behavior is inherently more predictable than an arbitrary program presented to the operating system. Our work takes advantage of the added predictability of database queries and we propose an execution mechanism that stabilizes the system so that it becomes less sensitive to estimation errors.

The second area of related work is in the design of memory managers for DBMSs. Several proposals have been made: [CARE89] [BROW93] [BROW94]. The drawback of these methods is that the internal workings of the database memory manager have to be changed. Our approach achieves the goal of preventing overload and underload without any internal changes to the DBMS.

The third area of related work is in the feedback control of workloads. The basic idea in the feedback approach is to sample some performance metric. If the performance metric exceeds a certain target value then the rate of admitting jobs into the system is reduced. If the performance metric is less than a certain minimum, then the rate of admitting jobs into the system is increased. Thus, the performance metric is kept at an optimal rate, by controlling the admission of jobs into the system. Most of the previous work using this approach has been targeted towards OLTP (On-line Transaction Processing) systems where thrashing due to data contention has been the main problem. Some examples of the feedback approach include: Adaptive control of the Conflict Ratio, Half and Half method, Analytic model using a fraction of blocked transactions as the performance metric, Wait-depth limitation, etc. Several of these methods have been summarized in [MOEN92] . Another good demonstration of this approach is provided by [PANG94] that deals with real-time database systems, and by [SCHR06] . More recently, Web servers have employed a feedback loop approach [LIU03] [CHEN01] [KAMR04] [ELNI04] .

The main problem with the feedback approach is in choosing the sampling interval over which the performance metric is measured. If this sampling interval is too small, then the system could oscillate and could end up being very unstable. Similarly, if the sampling interval is too large, then the system could end up being very slow to react to a changing workload and thus not be quick enough to prevent overload and underload behavior. Typical Business Intelligence workloads shift rapidly between small queries and huge queries. A performance metric and an associated sampling interval which is appropriate for one kind of workload may not work for a different kind of workload that runs only an instant later on the system. Thus the feedback loop approach is not appropriate for a rapidly changing BI workload.

There has been very little published in the area of workload management of BI workloads. These workloads are very different from OLTP and Web server workloads, which has been the main focus of workload management research. To our knowledge, the
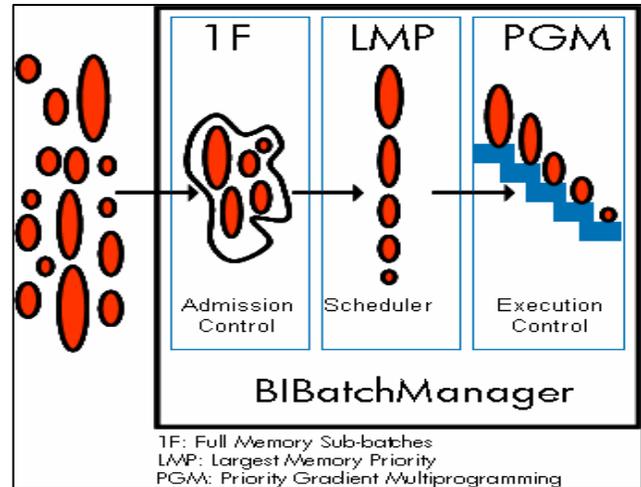

Figure 2: Component Design for BI Batch Manager

most common approach used by commercial BI systems is a 'static MPL' approach. In this approach, a 'typical workload' is run multiple times through the system. Each run is performed at a different MPL setting, and the corresponding throughput is measured. An optimal MPL is then chosen based on these trial and error experiments, or based on guesswork on the part of the DBA (Database Administrator). The workload is then 'throttled' down to this static MPL, which may be different for different times of the day.

There are several problems with this approach. Firstly, it is expensive to perform the trial and error experiments that this method calls for. Secondly, it results in a very approximate and inaccurate setting. The resulting MPL might work marginally well for the workload that was used in the testing, but is unlikely to work well with other workloads. Thirdly, it is static, and therefore cannot handle a dynamic shift in the workload. However, even with all its failings, it is still used by commercial systems because it is relatively easy to do. Unfortunately, the difference between the throughput of a well managed BI workload versus a poorly managed one can be an order of magnitude or more.

In the BI Batch Manager, we borrow the feedforward idea from the working set model and build upon it. We stabilize the system through a novel execution control component, so that it is tolerant of a wide range of prediction errors. The result is a workload management system that automatically avoids overload (and underload) while consistently running batch workloads at 'optimal' performance.

3. Overall System Design

Our BI Batch Manager has three primary components: An Admission Control component, a Scheduler and an Execution Manager. A schematic has been depicted in Figure 2. (The components are the solution components and we will discuss each of them in the text) The overall approach we follow to design these components is summarized as a four step process:

1. Identify a manipulated variable whose predicted value is suitable for BI workload management (for example: memory).
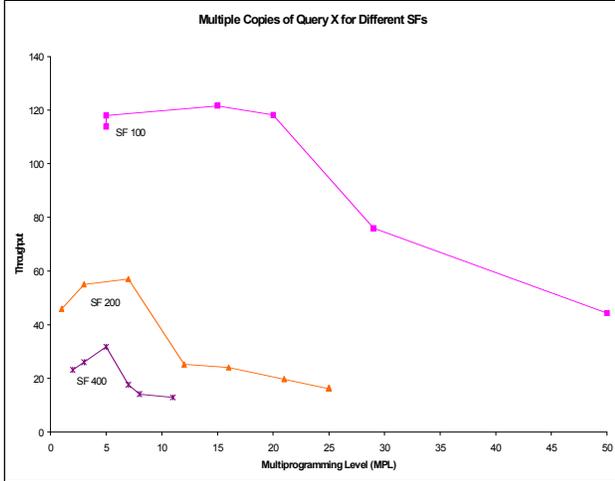
Figure 3: Throughput as a Function of MPL

Figure 4: Throughput as a Function of Memory

2. Use the manipulated variable for admission control, i.e. admit queries based on some value of the manipulated variable.

3. Schedule the queries so that the system behaves optimally for the admitted batch.

4. Make the system stable over a wide range of this variable, i.e., the system should not go into either underload or overload over a wide range of prediction errors for this manipulated variable.

Traditionally, MPL has been used as the manipulated variable of choice for workload management. A problem with MPL is that as the workload changes, the MPL needs to be changed too. This is illustrated in Figure 3, where we have plotted the throughput curves for three different workloads. These workloads are composed of multiple copies of TPC-H Query X at three different SFs (Scale Factors, which indicate the size of the database. Higher the scale factor, larger the database). The workloads were executed on a 32 node enterprise class, commercial database system. It can be clearly seen that the three different workloads have very different optimal regions. SF100 has an optimal region from MPL 7 to MPL 20, for SF200 the optimal region is from MPL 5 to MPL 10, whereas SF400 has an optimal region around 5. This figure clearly illustrates the problems with using MPL, as identified in Section 1. Namely, that it is impractical to fix an MPL for a mixed workload and then dynamically change the MPL as the workload progresses.

Memory, however, behaves much more predictably as a manipulated variable. We demonstrate this with the help of an experiment.. In Figure 4 we have plotted the same curves from Figure 3, with a different x-axis. Instead of MPL, the independent variable is now the total memory required by the workload per CPU (sum of the peak memory per CPU required by individual queries in the workload). It can be seen that the different SFs become suboptimal around 4 GB of memory per CPU, which is the average amount of free memory available per CPU for that experiment. (For the sake of simplicity we are assuming at this point that all CPU's have the same amount of available memory, and that the queries are fully parallelized, using the same amount of memory per CPU). This shows that the overload behavior can be predicted more accurately with memory than with MPL. A
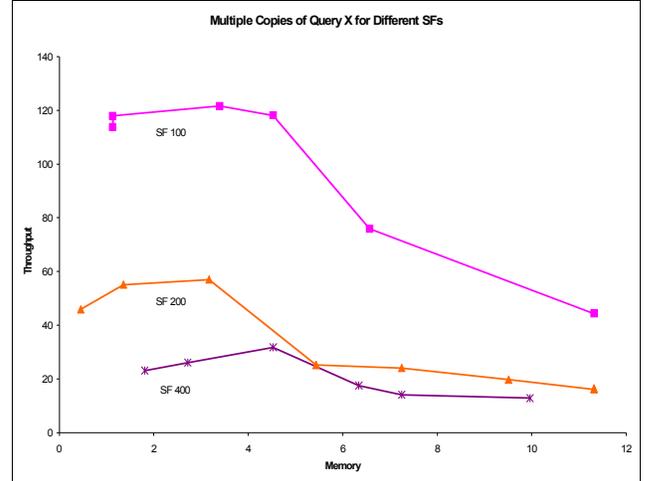
workload thrashes whenever its cumulative peak memory requirement per CPU exceeds the available memory per CPU.

A good choice of a manipulated variable would be any resource that causes a bottleneck when processing queries on the system, i.e., CPU, disc, memory, lock contention and message buffers. Saturation of CPU, disc or message buffers or lock contentions are limiting factors, i.e., if they are saturated the throughput cannot be improved. Any of these variables could be used as a manipulated variable for a feedforward system. Memory is different than these other variables. As we have seen, over subscription of memory can lead to serious degradation in performance because of thrashing. So, in this work we focus on memory and use it as the manipulated variable.

We introduce a memory based admission control scheme, where each batch is divided into sub-batches such that the memory requirement of queries in each sub-batch adds up to the available memory on the system. Our admission control uses memory to admit queries in a feedforward manner, i.e., compute a value for the manipulated variable (in our case: memory) that would give us desirable system behavior and then feed the system such that the manipulated variable has that desired value. The advantage is that the system behaves optimally. The obvious disadvantage to a feedforward system is that the system is vulnerable to errors in the value of the manipulated variable. We tackle this problem with the help of our scheduler and the execution control.

Before going further we present two definitions that we will use throughout the paper:

Definition 1: *For a query Q_i, with execution time E_i its memory requirement m_i is given as:*

$$m_i = max \{avg(m_{ict}) \mid 0 < t \leq E_i \}$$

Where m_{ict} is the memory required by query Q_i at time t at CPU c and the average is taken over all the CPUs at time t.

The memory requirement of a workload W, denoted by M_w is:

$$M_w = \Sigma m_i$$

Definition 2: *For a batch of queries Q_1, Q_2, ..., Q_n ε W , where a query Q_i belongs to a workload W and the minimum available memory across all CPUs is M, the size of the workload is given as*

xF where:

$$x = M_w / M$$

This means that the workload is a factor x times the size of the minimum available memory across all CPUs and F is a unit that indicates the factor of memory available. This is also called the F factor of the workload.

Example 3: *Suppose that the memory requirement of queries in the workload add up to 12 GB and the minimum available memory across all CPUs is 4 GB, then the workload is of size 3F.*

Remark*: We have assumed that the queries use the same amount of memory per CPU but it varies over time, hence we take the maximum of the averages in Definition 1. We have assumed that there are no dependencies between queries in a workload. We also assume that the memory requirement of queries are independent of each other.*

The heart of the technique is in stabilizing the system for prediction errors in the size of the workload. We begin by describing our execution control in section 4, which is the key component that stabilizes the system against estimation errors. We then describe our scheduling policy in section 5, which is designed to further stabilize the execution over a range of prediction errors. These lead to a simple admission control policy, which is described in section 6.

4. Execution Control

We saw in Figure 4 that if the size of the workload is greater than the amount of memory available on the system it can result in thrashing, which in turn results in severe performance deterioration. The queries in Figure 4 were executed at the same priority. We call this method of execution EPM or Equal Priority Multiprogramming. EPM is robust for a reasonable range of overestimates, i.e., if we overestimate the size for a workload and actual memory required is less then that, the throughput would still be in the optimal region. However, EPM is very unstable for underestimates, as depicted in Figure 4 where there is a sudden drop in throughput as the size of the workload increases beyond the available memory. For instance, from Figure 4, it can be seen that the example workload is optimal under the EPM execution control between the workload sizes of 1/3 F and 1F (there was approximately 4 GB of memory available per CPU during the experimental runs). To overcome the sensitivity to thrashing for workloads of sizes greater than 1F, we introduce Priority Gradient Multiprogramming, or PGM.

In PGM, queries are executed at different priorities such that a gradient of priorities is created. This results in queries asking for, and releasing resources at different rates (especially memory which is a primary cause for thrashing). This solution has proved to be very effective in protecting against overload. This makes admission control based on memory more feasible.

In our experiments, typically, PGM extends the stable region to workloads of size between 1/3 F and 3F. We will now describe PGM in detail, demonstrate its efficacy through some experiments and discuss why it works.

4.1 Priority Gradient Multiprogramming

PGM is a mechanism for executing queries in a database system where every query is assigned a different priority. More precisely:

Definition 4*: A mechanism for executing a batch of queries Q_1, Q_2, ..., $Q_n \varepsilon W$, where a query Q_i belongs to a workload W, in a DBMS is understood to be a Priority Gradient Multiprogramming mechanism if it has the following characteristics:*

1. *Order all queries: Specifically, all queries are uniquely ordered according to some ordering function F_{ord}, such that, $F_{ord}(Q_i) = j$, where $j \varepsilon [1, ..., n]$ and for all $i, j \varepsilon n$, $F_{ord}(Q_i) \neq F_{ord}(Q_j)$).*
2. *Pick queries in order and assign priorities in that order: Specifically, pick query Q_a, where for Q_a, $F_{ord}(Q_a) = 1$ and assign the highest priority P_1 to it. Then, pick query Q_b where for Q_b, $F_{ord}(Q_b) = 2$ and assign a priority P_2 such that $P_2 < P_1$. This is done until all the queries in the workload have been assigned a priority or the range of permissible priorities runs out.*

The difference between any two successive priorities, P_{i+1} and P_i is known as the step size and is a constant *k*. Since some operating systems have a fixed number of allowable priorities, setting *k = 1*, permits for the largest possible number of queries being assigned a valid priority. For some systems where different operations of a query are assigned different priorities by the executor, *k* can be larger. For instance in our experiments we have had to use *k = 2*.

Example 5: *Suppose there are ten queries in a workload: Q_1, ..., Q_{10} and the highest permissible priority for a query is 200. Say we choose $F_{ord}(Q_i) = i$ and k = 1. Then the priority of Q_1 is 200, Q_2 is 199 and so on, assigning Q_{10} a priority of 191. Then admit the workload with these priorities into the executor. Note that, in this example a priority of 200 is higher than a priority of 199.*

The ordering function, F_{ord}, could be a function that assigns order based on say CPU, memory or some other system variable. One useful function would be a function that assigns a random order. This would be useful because it doesn't require hard-to-compute characteristics of a query like: expected time taken, expected memory usage or some such resource requirement. Later, we introduce a memory based ordering that is useful in our context. Even in such a case, precise computations are not required, only an ordering is required. For example, there is no need to say Q_a requires x memory, just that Q_a requires more or less memory than Q_b.

An important consideration with PGM is that it requires that the operating system has a preemptive priority scheduler. A preemptive priority scheduler is a scheduler such that, when a process arrives at the ready queue, its priority is compared with the priority of the currently running process. If the priority of the currently running procedure is lower than the priority of the newly arrived process, the newly arrived process will preempt the CPU. This feature is standard on many commercial systems, including the HP NonStop Kernel, LINUX etc. [BOVE2000]

4.2 Experimental Evidence for PGM

To evaluate the performance of PGM we ran the workloads that have been plotted in Figure 3 and Figure 4. We now give the complete test bed specification:

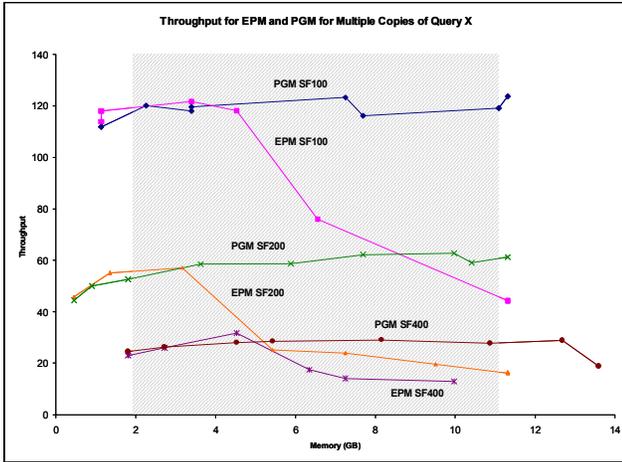1. Machine: 2 Segment (32 Node) commercial, enterprise

Figure 5: PGM Stabilizes Execution Over a Large Memory Range

class EDW, with 8GB physical memory per CPU.

2. Database: TPC-H. Scale Factors 100, 200, 400.

3. Workload: Complex Workload (multiple copies of Query X from the TPC-H benchmark).

4. The order function, F_{ord}, for the complex workload is not important since they are multiple copies of the same query. We specifically choose this type of construction to get results that are independent of the order of queries.

A run is comprised of a workload, a scale factor, the number of streams used, and the strategy used (EPM or PGM). The time of a run is converted to a throughput number. For example, if 50 queries take 30 minutes to complete, then the throughput = (50q/30m)*60m = 100qph. Each run is then represented by this single throughput number. We create a workload and run it with different MPLs under both schemes.

Figure 5 shows the throughput curves of three different workloads composed of multiple copies of a TPC-H Query X. (Please note that due to sensitivity considerations we cannot publish the query numbers) The top two curves are for a workload with SF = 100, the next two are for a workload with SF = 200 and the bottom two are for a workload with SF = 400. For each workload, two throughput curves have been plotted: a PGM curve and an EPM curve.

Most of Figure 5 is self evident. For all the three different scale factors for EPM, as the memory increases, the throughput first increases a little (underload), stabilizes at some high value (optimal) and finally falls down (overload). The initial rise is because the CPU has spare cycles which can be used with higher values of MPL. In the stable part of the region, there are not many spare cycles and the system is fully utilized. Finally as the MPL values are increased further (as indicated in the figure by increasing in the size of the workload), thrashing occurs and throughput falls.

If we look at the grey region we can see that using PGM has stabilized the execution. As memory increases from 1.5GB (approx 1/3F) to 13 GB (approx 3F), PGM stays in the optimal region, whereas after 4.5GB (approx 1F) EPM enters the overload region, and thrashes.

More specifically, if we compare the EPM curves with those of PGM we observe the following three differences:

1. For the first two regions (underload and optimal) the behavior is similar to that of EPM. However, for SF = 100 and SF = 200, PGM has no overload region - even as the size of the workload is increased, the system does not thrash and we continue to achieve high values for throughput. This means that for these two workloads *we have eliminated thrashing*.

2. For the SF = 400 workload, beyond a certain memory requirement value there is a drop in throughput for PGM. But the workload size at which this occurs is greater than 3F. So for the SF = 400 workload, PGM *increases the memory requirement at which thrashing occurs* and *reduces the amount of thrashing* (as measured by drop in throughput).

3. For all three scale factors, PGM *extends the optimal region to 13 GB which is greater than 3 times the average available memory on the system*. For SF = 100, the EPM optimal region is from memory = 1 GB to memory = 4 GB whereas for PGM the optimal region is memory = 1 GB to memory = 11 GB. For SF = 200, the region is extended from 1 – 3.5 GB of memory for EPM to 1 - 11 GB for PGM and for SF = 400 the region is extended from 1 – 4.0 GB of memory for EPM to 1 – 13 GB for PGM.

Additionally, we have conducted a series of experiments where we looked at a mixed workload (small queries intermixed with large queries). These are presented in the experimental results section. These were created by randomly mixing multiple copies of TPC-H queries. The behavior was similar to that of the previous experiments where PGM extends the optimal region.

The experimental results show that with the help of PGM we can achieve the following things:

1. Eliminate Thrashing in some cases.

2. In other cases, we increase the memory/MPL at which thrashing occurs and reduce the amount of thrashing.

3. Extend the region of memory/MPL in which a workload can run optimally.

We can further extend the optimal region by creating a suitable scheduling policy.

4.3 Why PGM Works

PGM is a mechanism for stabilizing the throughput curve over a wide range of workload memory (the manipulated variable). PGM stabilizes the right side of the throughput curve, i.e., it protects against overload, or in other words thrashing. The primary cause of thrashing is severe memory contention. The PGM scheme helps in regulating the peak memory requirement for a batch of queries, and effectively moves the right knee further to the right.

Consider a system where all queries are running with the same priority (EPM). In such a system, when a process *p* page faults, it goes to the wait queue, and when its page arrives, it goes to the end of the ready queue. All the processes prior to *p* in the ready queue are either finished or they page fault before the CPU gets to
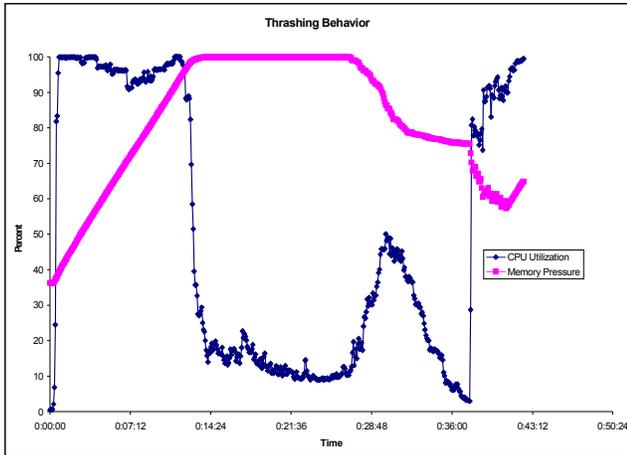
Figure 6: Memory Pressure and CPU for a Typical Thrashing Behavior

Figure 7: Memory Pressure Curves for EPM and PGM

p. In this way all the processes in the system get a 'fair' share of the resources.

In a PGM setting the sharing of resources is not so 'fair'. Say the highest priority process *q* page faults. When the page required by *q* arrives in memory, then instead of going to the back of the ready queue, *q* will preempt the currently running process in the CPU. Process *q* will get an 'unfair' share of resources. Thus, in a PGM mechanism (as in all priority based mechanisms) the highest priority query gets an unfair share of system resources – it gets what it needs quickly and efficiently. The remaining resources are automatically allocated to the query running at the second highest priority level. This continues down the priority gradient.

There can be a valid concern with resource starvation for low priority queries in that they might take very long to finish. However, in the case of a batch workload, the response time of an individual query is not important, rather the response time for the whole workload is of importance. As the higher priority queries finish, they leave the system and the lower priority queries get a larger share of system resources.

Let's look at thrashing a bit more carefully, to see why PGM works so effectively. Thrashing is caused by memory contention. We explain it in the context of a global page replacement policy – it replaces pages regardless of the process to which they belong. Suppose that a process needs more frames. It starts page faulting and taking away frames from other processes. These processes need those pages and so they also fault, taking frames from other processes. These faulting processes must use a paging device to swap pages in and out. As they queue up for the paging device the ready queue empties. As processes wait for the paging device the CPU utilization drops. The CPU scheduler sees the decreasing CPU utilization and increases the number of processes. The new processes start by taking frames from the existing running processes further exacerbating the problem and CPU utilization drops further. As a result the CPU kicks in more processes. Thrashing has occurred and the throughput plunges significantly, since processes are spending most of their time page faulting.

In Figure 6 we plot the CPU utilization and memory pressure against time for a system where thrashing occurred. As can be seen in the figure, once the workload starts, the CPU quickly
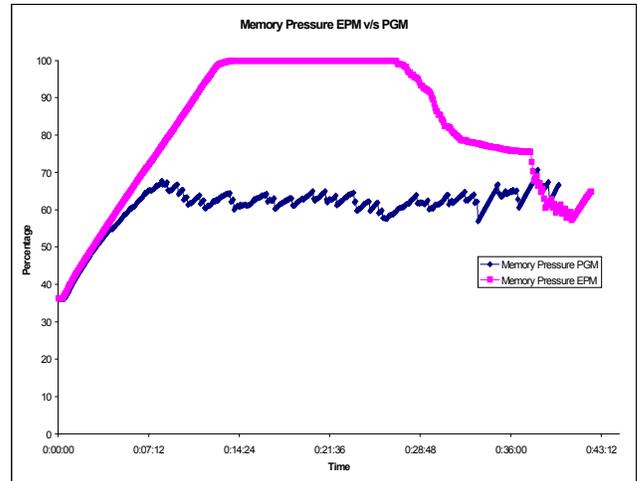
attains a 100% utilization. The memory pressure (percentage of physical memory that is being used by a processor – averaged across all the processors) slowly begins to build up as an increasing number of processes demand memory. Once the memory pressure builds up beyond a certain point, the CPU starts to drop and very quickly falls to around 20 % utilization. Only after the memory pressure begins to go down do we see the CPU utilization going up. From this discussion we can reasonably say that if the rise in memory pressure can be halted without losing too much CPU utilization then we have a solution to the problem of thrashing. This is precisely what PGM achieves. We explain this with the help of Figure 7.

Figure 7 depicts two memory profiles (Memory used as function of time) for a typical workload. The smooth light curve indicates the memory profile for a scheme where every query in the workload is assigned the same priority – the EPM Scheme. The dark, jagged curve indicates the memory profile for a PGM scheme.

From the curves, it can be noted:

1. The peak memory requirement for PGM is substantially lower than that of EPM. This clearly indicates that PGM reduces the peak memory requirement. This happens because PGM starts freeing memory sooner than EPM. The higher priority queries get all the resource they need and get done quicker. They then free their memory. The saw-toothed behavior of the PGM curve is an indicator that, as the higher priority queries get done, they release their memory hence reducing the peak value of memory pressure.
2. The initial slope of the PGM memory profile has a lesser slope than that of the EPM memory profile. This happens since the queries lower down in the priority order do not get a chance to ask for all the memory they need.

So, by asking for memory at a slower rate and releasing memory quicker, PGM reduces/eliminates thrashing. Thrashing is eliminated if the peak memory requirement never gets so high as to cause thrashing and even if it does become high, the peak values are still less compared to an EPM scheme.

5. Scheduler

Our execution control requires that the queries be given a priority, or in other words, they need to be arranged in some order F_{ord}. The Scheduler component performs this task.

If a workload is in the optimal region the order of priorities is not significant (There is further discussion on this in the experiment section). The throughput penalty for being in the overload region is much higher than being in the underload region. Hence we designed a scheduling order that stabilizes the system for memory underestimation errors (the overload region).

We call our ordering scheme LMP, or Largest Memory Priority. Under this ordering scheme the queries are assigned a priority in the order of their memory requirement. As the name suggests, the query with the largest memory requirement is given the highest priority.

Definition 6*: An order F_{LMP} of a batch of queries Q_1, Q_2, ..., Q_n ε W , where a query Q_i belongs to a workload W, in a DBMS is understood to be a Largest Memory Priority order iff:*

$$m_i > m_j => F_{LMP}(Q_i) > F_{LMP}(Q_j), i,j \; \epsilon \; [1..n]$$

LMP works for our purposes since the query with the largest memory requirement gets the highest priority and is amongst the earliest to be done and releases its memory. Typically, in a workload, the queries start building up their memories and the memory requirement continues to rise unless some memory is released. In EPM this causes thrashing since all queries have the same priority. In PGM we extend the optimal region as queries finish up and release their memories. LMP further extends the PGM stability by giving the highest priority to the query that would consume and then release the largest amount of memory.

5.1 Some Experiments with LMP

We compared the performance of LMP with a few other candidate scheduling strategies:

1. Random ordering of queries.

2. The shortest job was given the highest priority (SJP)

3. A Mix of priorities was created so that the memory was dispersed over priorities. For instance, consider the queries with the three largest memory sizes. Under this scheme, the query with the largest memory requirement would be given the highest priority, the query with the second largest memory requirement would be given the lowest priority and the third query would get the middle priority.

Since we are interested in the overload region we looked at workloads with size in and around 3F. In Figure 8 we have presented the results from five such experiments. Here the y-axis indicates throughput and the x-axis indicates the size of the workload (recall that size is measured in terms of memory).

It can be seen from Figure 8 that, for such large memory requirement workloads, LMP has greater throughput than any of the other ordering schemes.
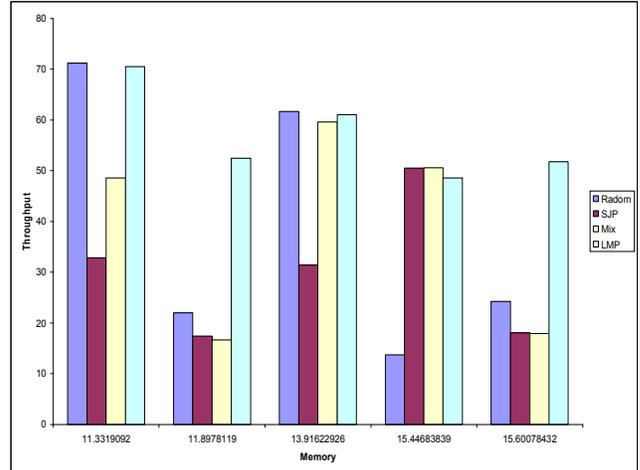

Figure 8: Throughput for Different Scheduling Policies

In the next section we present the admission control component of the BI Batch Manager.

6. Memory Based Admission Control

With memory as the manipulated variable, PGM as the execution control and LMP as the scheduling policy, the admission policy is a simple three step process:

1. Divide the batch of queries (a Workload) into sub-batches such that each sub-batch is of size 1F.

2. Admit a sub-batch into the system and assign priorities to the queries based on the LMP scheduling policy.

3. When a sub-batch is 'done', introduce a new sub-batch into the system and repeat until all the sub-batches are done.

The division of a batch of queries into sub-batches can be reduced to the problem of a 1-Dimensional bin packing problem (The problem of packing irregular 1-Dimensional object into bins of fixed size such that the number of bins is minimized) with 1F being the size of the bin and the queries as the packages that need to be packed in the bins. This problem is a NP-Hard problem. A number of approximate algorithms have been suggested in the literature [JOHN74]. A simple and useful one is FFD or First Fit Decreasing. FFD has known bounds of the number of bins being at most (11/9) + 1 times the optimal number of bins. In our context FFD can be rewritten as:

Algorithm 7 - FFD:

1. *Arrange queries in a descending order of memory requirement m_i*

2. *For every query Q_i in this order, insert Q_i into the first sub-batch S_{sub} that can accommodate the query (without the size of the sub-batch exceeding 1F).*

3. *Repeat Step 2 till all the queries have been assigned a batch S_{sub}.*

We now define 'done':

Definition 8: *We say a batch of queries is done when all three of*

the following conditions are satisfied:

1. *When a threshold T_{finish} fraction of the queries in the workload finish execution.*

2. *The minimum memory pressure over all CPUs falls below a threshold T_{mem}, where memory pressure is understood as the ratio of memory used to that of available memory.*

3. *The Average CPU utilization falls below a threshold T_{cpu}.*

We can now state the admission control policy:

Definition 9: Admission Control Policy: *A Query Memory based Admission Control Policy for executing a batch of queries Q_1, Q_2, ..., $Q_n \varepsilon W$, where a query Q_i belongs to a workload W, and the system has average memory available as M is:*

1. *For each query Q_i compute the memory requirement m_i.*

2. *Divide the queries into sub batches using FFD such that for a sub batch $\Sigma M_i < M$. Repeat Steps 3 to 4 till the all the queries in workload W finish.*

3. *Pick a batch S_{sub} that is not done. Prioritize all queries in sub batch S_{sub} using LMP.*

4. *Execute the queries in S_{sub}. When the batch S_{sub} is 'done' (Definition 8), goto Step 3.*

Sub-batches do not have to be run serially. They can be overlapped, using definition 8 of when a sub-batch is considered 'done'. In practice, however, we have observed that the CPU utilization stays at an average of close to 100% to the point where the batch finishes. Hence, as a practical simplification, we can let the whole sub-batch of queries complete, before introducing a new sub-batch.

As discussed before, the idea of using memory for admission control has been around for a while in the OS community, for example the Working-Set approach is based on this idea. The challenge with these approaches is to find out how many frames a process might need or to keep track of a processes' working set (The working set model starts by looking at how many frames a
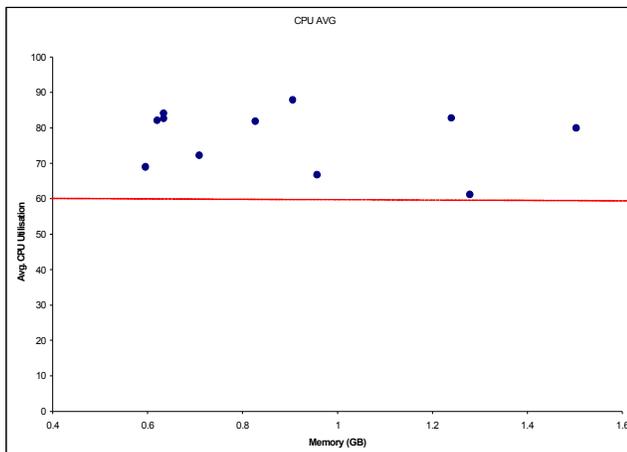
process is currently using).

In the world of DBMS, we are more fortunate. Every query has to have a plan before it is executed. This plan details the cardinality of each and every operator in the plan. This information can be used to estimate the memory requirement of a query. Literature suggests that memory can be predicted to a reasonable level of accuracy [SACC86].

A drawback of predicting memory in a real DBMS is that the cardinalities of various operators can sometimes be off by an order of magnitude or more. However, to deal with this, commercial DBMSs often restrict the maximum memory that can be consumed by a large memory operator. This gives an upper bound on the memory estimation error for a query. This is the key reason that enables us to use memory prediction as a manipulated variable for BI Batch Manager.

Furthermore, the memory estimation error of an entire workload is less than the estimation errors of the individual queries, since some overestimates cancel out underestimates. We present this analysis more formally in the next section below.

7. Experiments and Discussion

We have done a series of experiments to test various aspects of our BI Batch Manager. Recall that the BI Batch Manager (Figure 2) consists of three components (i) Admission Control with 1F memory and FFD as the algorithm (ii) Scheduler with LMP as the algorithm and (iii) Execution Control with PGM as the technique. The experiments were conducted in the following framework:

1. Machine: Same as discussed previously, namely, a 2 Segment (32 Node) commercial class Enterprise Data Warehouse , with 8GB physical memory per CPU

2. Database: TPC-H, SF50, SF100, SF200.

3. Workload: 48 mixed workloads of random sizes were created by uniform random sampling (with replacement) of TPC-H queries.

We also introduce some workload metrics to measure the performance.
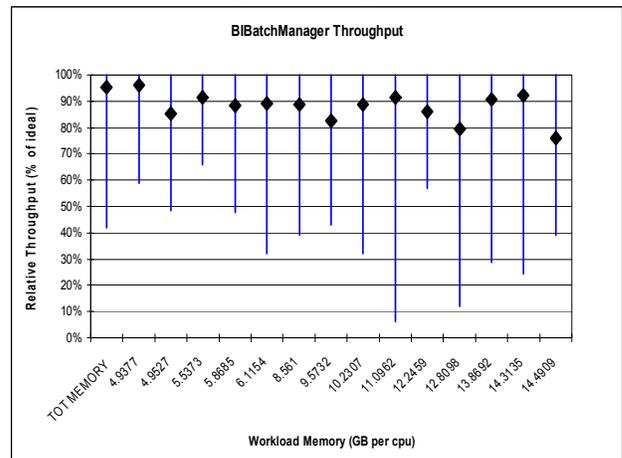
1. Ideal Throughput (IT) Compute the CPU work done by

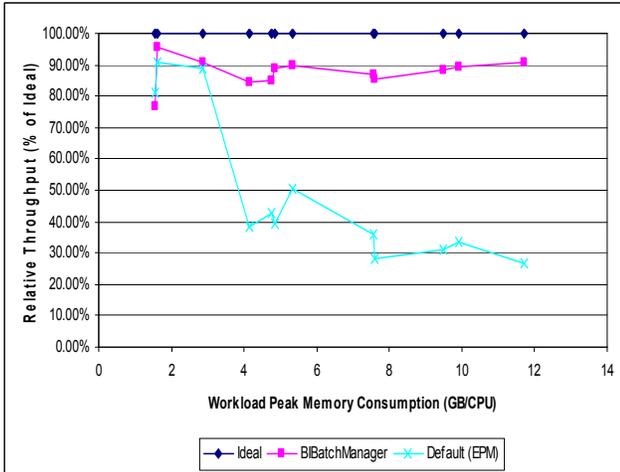**Figure 9: Average CPU Consumption Results For Workloads of Size Around 1/3 F**

Figure 10: Throughput Results for Workloads of size 1F to 3.5F

Figure 11: Throughput results for workloads from 1/3F to 3F

Figure 12: Throughput Results for Workloads Randomly Created With a Mix of TPCH Queries with Various Scale Factors

each query in the workload (in CPU seconds) by finding the area under the CPU utilization curve. This is sometimes called the 'path length' of a query. Add up the CPU work done for all the queries in the workload. Divide by 1 (100% utilization). This gives a lower bound on the amount of time the workload can take to run under ideal conditions. Convert this to a throughput number. This is the Ideal Throughput (IT), which is the upper bound on the throughput that can be achieved for this workload.

2. BI Batch Manager throughput (BT): Actual workload throughput under the BI Batch Manager system.

3. EPM throughput (ET): Actual workload throughput under the default Equal Priority Multiprogramming.

Note that, in practice it is impossible to obtain the ideal throughput, since even for a highly parallelized query there are a number of serial operations. Thus, the ideal throughput should be viewed as a good upper bound, but not necessarily achievable.

The aim of a BI Batch Manager is to maximize the throughput of BI Batch Workloads while protecting against underload and overload.

Our overall claim for the BI Batch Manager is the following:

Claim 12: *If the workload size is 1F then the BI Batch Manager does not go into underload or overload. Hence it works in the optimal region of the throughput curve.*

This is easy to see why: we know that if the CPU is kept busy then there is no underload. If the memory is full, then in the case of queries, some big memory operator (BMO) is using memory. The CPU needs to process the data in the memory and that will keep it busy, hence avoiding underload. Similarly, if the memory does not exceed 1F then there is no reason for the system to go into overload. Since the BI Batch Manager does not allow the system to go into underload or overload, it works in the optimal region of MPL.

A similar claim can be made for EPM, but what makes the BI Batch Manager useful is that it is stable for underestimates in the size of the workload. Both EPM and BI Batch Manager are stable for overestimates in the size of the workload.
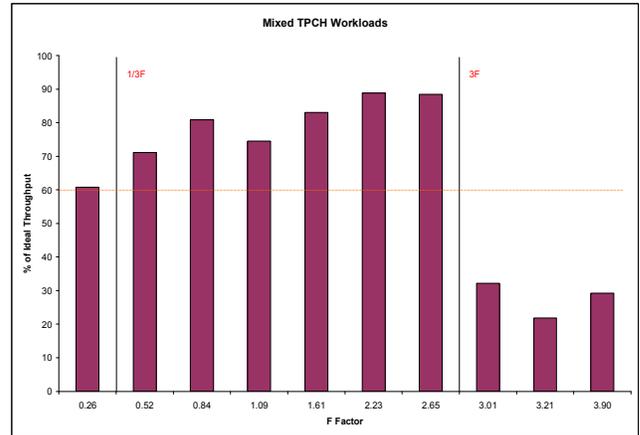
We begin with experiments and a discussion that shows that the BI Batch Manager avoids underload for overestimation errors in the workload size. We then present experiments and discussion that shows that BI Batch Manager avoids overload for underestimation errors in the workload size. Finally we present overall results that show a BI Batch Manager extends the optimal range and performs optimally in it.

7.1 Avoiding Underload

Overestimation errors in the size of the workload might cause workloads of a small size being executed. This can cause underload. Experimentally, we show that for a left bound of workload size 1/3F, the BI Batch Manager on our system does not cause underload. In Figure 9 we have plotted the average CPU utilization across different CPUs for experiments run with the BI Batch Manager, for workloads of size from 1/10F to 1/3F. It can be seen that the CPU consumption is higher than 60 % which is understood to be not an underload situation.

7.2 Avoiding Overload

Underestimates in the size of the memory might cause workloads of a large size being executed. We show empirically, that our BI Batch Manager system is robust up to a workload size of 3F. In Figure 10 we have plotted the throughput results for fifteen mixed workloads of size varying from F to 3.5 F that were executed both under EPM and BI Batch Manager. The throughputs have been plotted as a percentage of the ideal throughput. For the same workload, the black dots are the ratio of the BI Batch Manager Throughput (BT) to that of the Ideal Throughput (IT) and the bottom of the vertical lines indicate the ratio of the EPM throughput (ET) to the ideal throughput (IT). The x-axis is an enumeration of the memory requirement for fifteen different experiments.

It can be seen from the plots that for all the workloads, BI Batch Manager outperforms EPM. The EPM results with low throughputs are the result of overloading. BI Batch Manager outperforming EPM clearly indicates that for a BI Batch Manager, either there was no overload or significantly reduced overload. For fourteen of the fifteen workloads BTF is at least 80 % of IT.

80% of ideal throughput is certainly considered good by most practitioners.

7.3 Optimal Region

To study PGM's optimal region, we created two separate sets of workloads. Each workload set consisted of workloads that varied in size from very small to very large.

The first set had twelve mixed workloads from TPC-H SF200. Workloads were created by first choosing a memory number between 1.33GB (approximately 1/3F) and 12GB (approximately 3F). Then, queries were randomly chosen from the set of TPC-H queries (with replacement) until the memory size of the workload equaled the desired memory size. In Figure 11 we have plotted throughput results for these workloads. It can be clearly seen that for all the workloads, BI Batch Manager has a ratio of BT to IT of greater then 80 %. It can also be seen that EPM has a large drop in throughput for workload sizes greater than 1F.

Finally, we present results for a wide variety of workload sizes, measured as F values. These demonstrate that PGM behaves optimally for the span of 1/3F to 3F. This set had 10 mixed workloads, chosen from a broad set of workloads. We used queries from 3 different scale factor TPC-H databases (SF50, SF100, SF200) as the candidate set of queries. We randomly chose queries from this diverse candidate set (with replacement) and in this manner created workloads of various sizes. Before the execution of every workload, we computed the actual available memory on the system, and used it to compute the F factor. In Figure 12 we have plotted the throughput results for these workloads. It can be clearly seen from this figure that PGM behaves optimally for our experimental setup between 1/3F and 3F.

These experiments experimentally validate the usefulness of BI Batch Manager.

8. Conclusions

We have seen that running batch workloads is an important activity of BI systems. Batch workloads include the rollups that need to aggregate daily warehouse data to more usable information. Batch workloads also include Business Intelligence reports that are typically created daily (or nightly) on an EDW. The trend is to have larger and more complex reports being created daily. An EDW can spend a significant portion of its time running batch rollups and reports. BI workloads typically consist of a wide variety of small and large queries, and the workload mix can change in an instant. This makes the current static MPL techniques, or feedback based MPL control techniques insufficient to manage a BI workload. The BI Batch Manager is a feedforward, priority-based workload management system that complements the built-in operating system controls for load control.

The BI Batch Manager consists of three main components: a 1F Admission control policy, the PGM (Priority Gradient Multiprogramming) Execution control component, and an LMP (Largest Memory Priority) scheduling or packing policy. The PGM execution control is the key component. It is a novel execution control policy that runs queries on a priority gradient and stabilizes the memory range over which workloads will

experience optimal throughput. The other two components serve to make the BI Batch Manager a complete system that is practical for commercial systems.

The BI Batch Manager has been experimentally validated on a real commercial, enterprise class DBMS. We have shown that the BI Batch Manager can automatically manage real mixed-workloads and consistently attain high throughputs, with overload and underload avoidance, and with stability across a wide operating range. As a next step, we will be extending the BI Batch Manager to handle interactive, ad hoc queries as well.

9. References

[BLAK82] Blake, R. 1982. Optimal control of thrashing. In *Proceedings of the 1982 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Seattle, Washington, August 30 - September 01, 1982). SIGMETRICS '82. ACM Press, New York, NY, 1-10

[BOVE00] Daniel Bovet and Marco Cesati,"Understanding the Linux Kernel", O'Reilly, Oct 2000

[BROW93] Kurt Brown, Michael J. Carey, Miron Livny, "Managing Memory to Meet Multiclass Workload Response Time Goals", Proc of VLDB, p328-341, 1993

[BROW94] Kurt Brown et al, "Towards Automated Performance Tuning for Complex Workloads", Proc of VLDB, p 72-84, 1994

[CARE89] Cary, M, et al, "Priority in DBMS Resource Scheduling", Proc VLDB, Amsterdam, 1989

[CARE90] MJ Carey, et al. "Load control for locking: The 'half-and-half' approach", ACM Symposium on Principles of Database Systems, 1990.

[CARR81] Richard W. Carr and John Hennessey, "WSClock – A Simple and Effective Algorithm for Virtual Memory Management", Proc. 8th Sumposium of Operating Systems, 15 pt 5, 1981

[CHEN01] Chen, X., et al. "An admission control scheme for predictable server response time for web accesses". In *Proceedings of the 10th international Conference on World Wide Web* (Hong Kong, Hong Kong, May 01 - 05, 2001). WWW '01.

[DENN68-WS] Denning, PJ, "The working set model for program behavior", Comm. ACM 11, 5, May 1968, P. 323-333

[DENN68-TH] Denning, PJ, "Thrashing: Its Causes and Prevention", Proc AFIPS 1968 FJCC 33, p. 915-922

[DENN76] Peter J. Denning et al, „Optimal Multiprogramming", Acta Informatica, Springer-Verlag, 1976

[DENN80] Peter J Denning, "Working Sets Past and Present", IEEE Trans Softwar Engrg, 1980

[DENN95] P.J. Denning, "A short theory of multiprogramming," *mascots*, p. 2, Third IEEE International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '95), 1995

[ELNI04] Sameh Elnikety et al, "A method for transparent admission control and request scheduling in e-commerce web sites", WWW2004, May 2004.

[HEIS91] HU Heiss and R Wagner, "Adaptive load control in transaction processing systems", Proc. VLDB, pages 47-54, 1991

[JIAN02] Song Jiang, Xiaodong Zhang, "TPF: a dynamic system thrashing protection facility", Soft. Pract. Exper. 2002; 32:295-318

[JOHN74] Johnson, D., Demers, A., Ullman, J., Garey, M., Graham, R.: *Worst-case performance bounds for simple one-dimensional packaging algorithms*. SIAM Journal on Computing 3 (December 1974) 299--325

[KAMR04] Kamra, A.; Misra, V.; Nahum, E.M., "Yaksha: a self-tuning controller for managing the performance of 3-tiered Web sites," *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on* , vol., no., pp. 47-56, 7-9 June 2004

[LIU03] Xue Liu, et al, "Online Response Rime optimization of Apache Web Server », IWQoS 2003, LNCS 2707, pp 461-478, 2003

[MOEN92] A Moenkeberg and G Weikum, "Performance evaluation of an adaptive and robust load control method for the avoidance of data contention thrashing, Proc of VLDB, p 432-443, 1992

[PANG94] HweeHwa Pang et al, "Managing Memory for real-time queries", ACM SIGMOD, p 221-232, 1994

[PANG95] HweeHwa Pang, Michael J. Carey, „Multiclass Query Scheduling in Real-Time Database Systems", IEEE Trans on Knowl. And Data Engrg, Vol 7, No 4, Aug 1995

[RODR73] Juan Rodriquez-Rosell, Jean-Pierre Dupuy, "The Design, Implementation, and evaluation of a Working Set Dispatcher, Comm 16, 4, April 1973

[SACC86] G.M. Sacco and M. Schkolnick, "Buffer management in relational database systems", ACM Trans. Database Systems, Vol 11, No 4, Dec 1986, 473-498

[SCHR06] Bianca Schroeder, et al, "How to Determine a Good Multi-Programming Level for External Scheduling," *icde*, p. 60, 22nd International Conference on Data Engineering (ICDE'06), 2006

[SILB05] Silberschatz, Galvin, Gagne, "Operating System Concepts", 7th edition, John Wiley and Sons, pg 315-370

[SMIT80] Alan Jay Smith, "Multiprogramming and Memory Contention", Software-Practice and Experience, Vol 10, 531-552, 1980

[WEIK02] Gerhard Weikum et al, "Self-Tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering", Proc of VLDB, 2002.

10. Appendix A: Statistical Results for Addition of Memory Prediction Errors

For the purpose of analysis, we assume that the memory estimation errors are distributed normally. Let there be n queries in the workload. Let the memory prediction for the ith query, q_i denoted by $Q(m_i) \sim N(\mu_i, \sigma_i^2)$. It is well known that sum of n independently distributed normal random variables is Normal: More formally if $Q(m_i) \sim N(\mu_i, \sigma_i^2)$ then $Q(M) = \Sigma Q(m_i) \sim N(\mu, \sigma^2)$:

$\Sigma N(\mu_i, \sigma_i^2) = N(\mu, \sigma^2)$ where $\mu = \Sigma\mu_i$ and $\sigma^2 = \Sigma\sigma_i^2$. Then:

$N(\mu, \sigma^2) = \Sigma N(\mu_i, \sigma_i^2)$

$$=> N(\mu, \sigma^2) = N(\Sigma\mu_i, \Sigma\sigma_i^2)$$

$$=> N(\mu, \sigma^2) = N(\Sigma m_i, \Sigma\sigma_i^2) \qquad (1)$$

$$=> N(\mu, \sigma^2) = N(M, \Sigma\sigma_i^2) \qquad (2)$$

A point on the normal distribution such that 99% of the probability lies to the left of the point is given by $\mu_i + 3\sigma_i$. Let this point be k times the value of the mean. Then: $\mu_i + 3\sigma_i = k\mu_i$. This implies: $\sigma_i = (k-1)\mu_i/3$.

We consider two cases:

Case I: All queries are the same. Then $m_i = M/n => \sigma_i = ((k-1)M)/(3n)$.

$$Since\ N(\mu, \sigma^2) = N(M, \Sigma\sigma_i^2) \qquad From\ (2)$$

$$=> N(\mu, \sigma^2) = N(M, ((k-1)M/3)^2/n)$$

$$=> \sigma = ((k-1)M))/(3\sqrt{n})$$

We compute the z-value for a point that is F times the mean, This point is given by $M + FM$. We know $z = (x-\mu)/\sigma$. This implies:

$$=> z = (M+FM-M)/((k-1)M/(3\sqrt{n}))$$

$$=> z = 3F/(k-1)*\sqrt{n} \qquad (3)$$

Case II: All queries lie on a gradient. Then $m_i = m^* i$, where m^* is some factor such that: $\Sigma m^* i = M$. This implies $m_i = (2Mi)/(n(n+1))$ $=> \sigma_i = ((k-1)2Mi)/(3n(n+1))$. This implies:

$$Since\ N(\mu, \sigma^2) = N(M, \Sigma\sigma_i^2) \qquad From\ (2)$$

$$=> N(\mu, \sigma^2) = N(M, ((k-1)2M/3)^2*(2n+1)/(6n(n+1)))$$

$$=> \sigma = ((k-1)2M/3)*\sqrt{(2n+1)/(6n(n+1))}$$

We compute the z-value for a point that is F times the mean, This point is given by $M + FM$. We know $z = (x-\mu)/\sigma$. This implies:

$$=> z = (M+FM-M)/(((k-1)2M/3)*\sqrt{(2n+1)/(6n(n+1))})$$

$$=> z = 3\sqrt{6}F/(2(k-1))*\sqrt{(n(n+1))}/\sqrt{(2n+1)} \qquad (4)$$

We can state the result formally now for a workload W such that the memory requirement of a query $q_i \in W$ be predicted *with* $\sim N(\mu_i, \sigma_i)$ and $\Sigma m_i = M$ where m_i is the memory requirement of q_i and M is the memory requirement of the W and the number of queries in W is n.

Theorem 10: *For $n \geq 10$ and $m_1 = m_2 = ... = m_n$, if each predicted m_i is within 10 times the actual 99 % of the time then the predicted value for M will be within 3 times the actual M 99% of the time under the assumption of normality and the memories being independent.*

Proof: Substituting k=10, F=3 and n = 10 in Eq. 3, we get a z-value 3.16. And z-value increases as n increases.   ┆ □

Theorem 11: *For $n \geq 12$ and $m_i = m^* i$ for some m^*, if each predicted m_i is within 10 times the actual, 99 % of the times then the predicted value for M will be within 3 times the actual M, 99% of the time under the assumption of normality and the memories being independent.*

Proof: Substituting k=10, F=3 and n = 12 in Eq. 4, we get a z-value 3.06. And z-value increases as n increases.   □