Automated Workload Management for Enterprise Data Warehouses

Abhay Mehta Chetan Gupta Song Wang Umeshwar Dayal

Hewlett Packard Labs
firstname.lastname@hp.com

Abstract

Modern enterprise data warehouses have complex workloads that are notoriously difficult to manage. Additionally, RDBMSs have many "knobs" for managing workloads efficiently. These knobs affect the performance of query workloads in complex interrelated ways and require expert manual attention to change. It often takes a long time for a performance expert to get enough experience with a large warehouse to be able to set the knobs optimally. Typically the warehouse and its workload change significantly within that time. This makes the task of manually optimizing the knob settings on a warehouse an impossible one. In this context, our goal is to create self managing Enterprise Data Warehouses. In this paper we describe some recent advances in building an automatic workload management system. We test this system against real workloads against real enterprise data warehouses.

1 Introduction

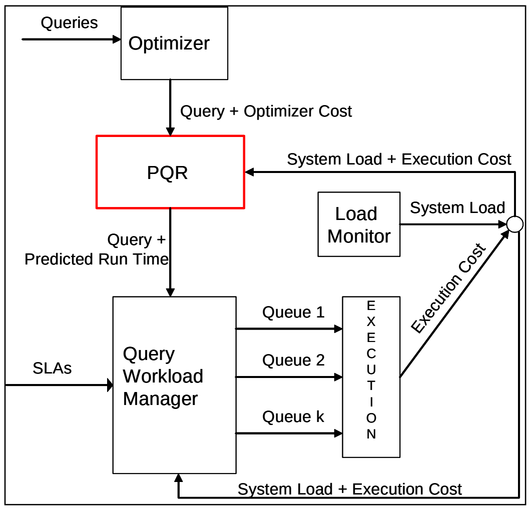
Many organizations are creating and deploying Enterprise Data Warehouses (EDW) to serve as the single source of corporate data for business intelligence. Not only are these enterprise data warehouses expected to scale to enormous data volumes (hundreds of terabytes), but they are also expected to perform well under increasingly complex workloads, consisting of batch and incremental data loads, batch reports and complex ad hoc queries.

The problem of database workload management aimed at self-tuning database systems has been studied in the literature (see Weikum [28] for a review of the advances in this area). We have borrowed from this work the idea of using multiprogramming level (MPL) to model the load on the system. However, the previous work was done in the context of OLTP workloads, not the complex query workloads typical of Business Intelligence (BI) data warehouses, which is the focus of our work.

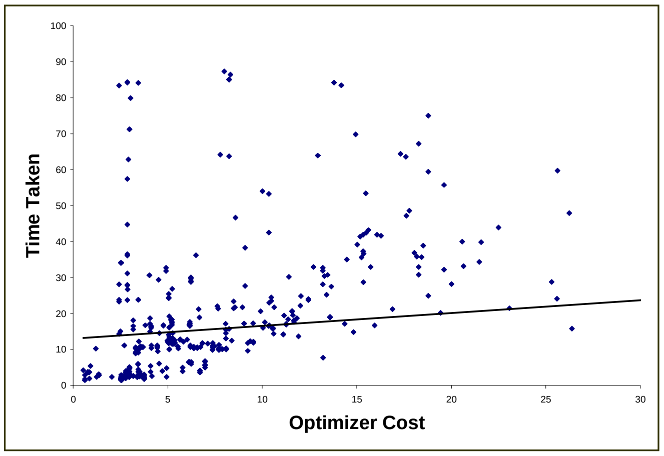
In this work we deal with two important challenges towards achieving Automatic Workload Management: *Predictability* and *Manageability*. In the next few sections, for each of these challenges we present a sketch of our solution. The detailed discussions and results can be found in [16] and [17].

Figure 1(a) depicts the architecture of an automatic workload management system. The *optimizer* outputs an execution plan for a query and an estimate of the query cost, which are input to the *Prediction of Query Runtime (PQR)* block. In addition, a *Load Monitor* extracts a load feature vector, which is also input to the *PQR* block.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



(a)



(b)

Figure 1: (a) System Diagram for an Autonomic EDW (b) Relationship between Optimizer Cost and Actual Execution Time

Whenever a new query comes in, the *PQR* block estimates the execution time of the query under current load conditions. This estimate is passed on to the *Workload Manager*, which schedules the queries. Other components in the system (not shown in the figure) keep track of the query's progress relative to its predicted execution time, and use this information to detect problem queries (such as runaway queries). All of this information is fed back to the *Workload Manager*, which can then apply the appropriate control actions to rectify the problems.

2 Related Work

There has been a tremendous amount of work on cost models for query optimization (see for example Graefe [9] for a survey). However, while these cost models are useful to the optimizer for selecting low cost execution plans, their cost estimates are very often not good predictors of actual query execution times (See Figure 1(b)).

Analytical approaches have been used for estimating query response times [23, 27] and there are a few commercial products that use analytical and simulation models to predict query execution times [1,14,20,21,25]. The analytical approaches depend on the creation of resource models which are notoriously complex and difficult to create and hence the results may not be relevant in practice.

Certain machine learning techniques have been used in the context of databases. The LEO learning optimizer uses a feedback loop of query statistics to improve the optimizer during run time [15, 26]. Raatikainen [22] summarizes some of the early work in using clustering for workload classification. In PLASTIC [8] queries are clustered to increase the possibility of plan reuse. Although these are interesting applications of machine learning techniques, none of these apply machine learning to our problem at hand.

Furthermore, statistical techniques, analytical techniques, and machine learning techniques have been used previously to predict execution times of tasks and resource consumption in fields other than database systems [10]. However, to the best of our knowledge there is no prior work in using machine learning techniques to build models for predicting query execution time ranges.

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

The problem of over subscription of memory, the primary cause of thrashing has been studied extensively since the 1960s. Several heuristics have been proposed for doing admission control either by explicitly controlling the MPL or otherwise. These include the Knee Criterion, the L=S criterion, the Page Fault Frequency algorithm, and the 50% rule [6]. However, thrashing is still an unsolved problem in operating systems and work continues in this area [11].

Another area of related work is in the design of memory managers for DBMSs. Several proposals have been made for memory managers in DBMS: [2–4]. The drawback of these methods is that the internal workings of the database memory manager have to be changed.

One more area of related work is in the feedback control of workloads. 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. Several of these methods have been summarized in [18]. Another good demonstration of this approach is provided by [19] that deals with real-time database systems, and by [24]. More recently, Web servers have employed a feedback loop approach [5, 7, 12, 13].

To our knowledge, the 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 and an appropriate MPL computed. 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: it is expensive, it results in a very approximate and inaccurate setting and since it is static it is not suitable for heterogeneous nature of a BI workload.

3 Predictability

The execution time of BI queries that run on large EDWs can vary from microseconds for simple lookup queries all the way to multiple hours for complex data mining queries. An effective workload management system depends heavily on an estimate of the execution times of queries in the workload, prior to running the queries. Estimating execution time accurately is a hard problem, especially on a loaded EDW with a complex workload.

Previous researchers have focused on predicting precise execution times. In our experience, this is extremely difficult to do with high accuracy. Furthermore, for workload management, it is actually unnecessary to estimate a precise value for execution time - it is sufficient to produce an estimate of the query execution times in the form of time ranges (for instance, queries may be assigned to different queues based on their execution time ranges). This allows us to reformulate the problem and bring in the machinery of machine learning to address it. It is precisely this problem of estimating query execution time ranges that we address in this paper. We focus on the following issues:

1. Discovering, selecting, and computing query plan features and system load features as classification attributes.
2. Finding appropriate execution time ranges to be used for prediction.
3. Ensuring high prediction accuracy.
4. Efficient algorithms for model building and deployment.

Researchers and practitioners have built increasingly sophisticated cost models for query optimization. However, building an accurate analytical model is difficult especially under varying load conditions. Using an optimizer's analytical cost model to estimate the actual execution time of a query on a loaded system has met with limited success in the field and it is common knowledge that query cost estimates produced by query optimizers do not accurately reflect query run times. For example, in Figure 1(b), we have plotted the actual query execution time and cost estimates for a batch of queries run on a customer database. The scatter plot and the best fit line show that the optimizer cost is not an ideal predictor for query execution time.

We take a different approach: we "learn" from the execution histories of various queries under varying load conditions. In particular, from the execution histories, we extract query plan features provided by the optimizer and system load features from the environment on which the queries were run. Discovering features and selecting which features to use is itself a challenging problem. We then build a predictive model that can estimate the execution time range of a query.

We found that conventional machine learning approaches to building predictive models, such as regression and decision tree classifiers were not adequate. The challenges were several since we are interested in not only predicting the time ranges but also in discovering them:

1. The time ranges should be sufficient in number. It would be meaningless to predict that all queries belong to a single time range.

2. Their span should be meaningful. Very small or very large time buckets are not very useful.

3. As with all predictive models the accuracy of prediction should be high.

4. The model should be cheap to build and deploy.

To address these we came up with a novel hierarchical approach. We call the predictive models built using this approach, PQR (Predicting Query Run-time) Trees. A PQR Tree is a hierarchical classification tree such that for every node, there is a binary classifier that decides how best to divide the time range of the node into two sub ranges for the two children of the node. There is also an associated accuracy for each node. Every node and leaf of the tree corresponds to a time range. At every node, we not only find the two sub ranges for the time range of the node but also a classifier that can predict the two ranges.

As illustrated in Figure 2(a), the prediction model is first built and trained using a set of execution histories of various queries under varying load conditions. Then, for an incoming query in the workload, PQR Tree will return an estimated execution time range with an associated accuracy. There are two overall steps:

1. *Obtaining a PQR Tree based on historical data of queries run on the system*: This involves three steps:

   (a) From the historical data extract query plan features like optimizer cost, number of joins, join cardinality, and etc.

   (b) From the historical data extract the system load vector for each query. The system load vector consists of the number of queries and the number of processes running while the query was executing.

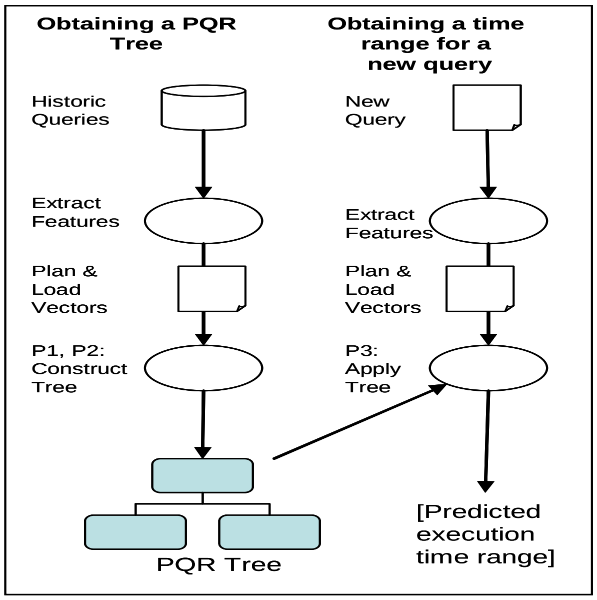
   (c) Build a PQR Tree with the feature vectors built above. This is done by choosing the combination of the classifier and the time interval that gives the highest accuracy.

2. *Obtaining a time range for a new query by applying the PQR Tree to a new query*: This is done by extracting a feature vector from the new query and applying the PQR Tree obtained in the previous step.

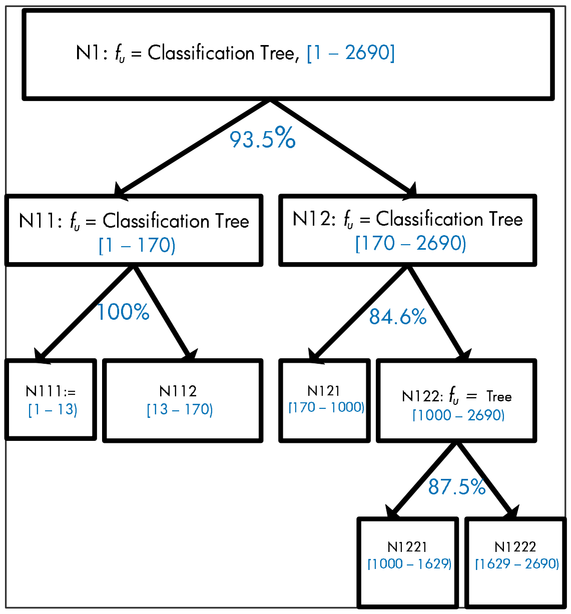
A sample PQR Tree is presented in Figure 2(b). The classifier, f_u associated with the root node divides the time range of [1, 2690] seconds into two: [1, 170) and [170, 2690] seconds with associated accuracy of 93.5%. The rest of the nodes can be interpreted similarly.

We did two series of experiments to verify our approach. They consisted of two different systems, both running a commercial, enterprise class DBMS and two different data sets. We ran a thousand queries against SF = 1, 50, 100, 200 TPCH databases. The queries were generated automatically and ran against all the tables. They include joins and order-bys. Sixteen of these queries were TPCH benchmark queries. We looked at four query MPL values: 4, 6, 8 and 10, i.e., we executed the queries in parallel with the number of parallel streams equal to the MPL value.

For the second set we took a set of actual BI (Business Intelligence) queries run in a day by one of HP's customers. They include both ad hoc queries and canned reports. There were a total of 500 queries in this data set. The database has more than 38G rows and total size is over 21TB. This experimental setup consisted of



(a)


N1: f_u = Classification Tree, [1 – 2690]
93.5%
N11: f_u = Classification Tree [1 – 170]
N12: f_u = Classification Tree [170 – 2690]
100%
84.6%
N111:= [1 – 13]
N112 [13 – 170]
N121 [170 – 1000]
N122: f_u = Tree [1000 – 2690]
87.5%
N1221 [1000 – 1629]
N1222 [1629 – 2690]

(b)

Figure 2: (a) Solution Components for Building PQR Tree (b) Sample PQR Tree where Time is in Seconds

hundred different tests. We took ten different MPLs: 8, 16, 32, 48, 64, 96, 128, 192, 224, 256 and ran each experiment ten times. In another series of hundred different tests we used a subset of smaller queries from the original five hundred queries.

In consultations with DBAs who manage workloads, we created a quality metric: a model that can predict at least four "reasonable" ranges[1] of query execution times with an accuracy of greater than 80% is considered acceptable. This metric captures all the key attributes we discussed earlier. Over 90% of the PQR Tree models were found to be acceptable. The results from various experiments were found to be extremely encouraging [16].

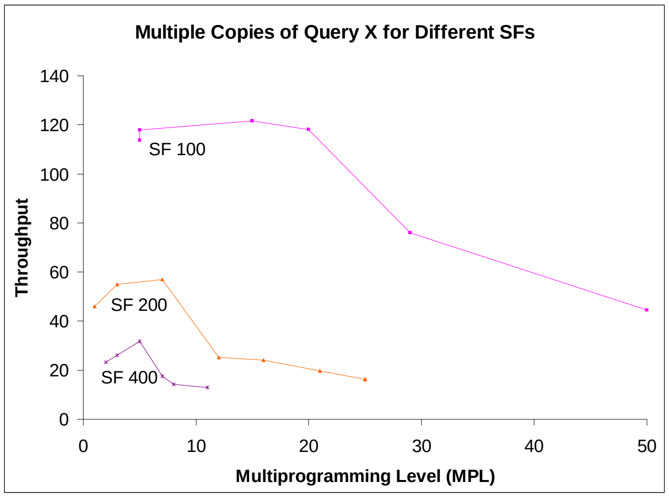
4 Manageability

Traditionally, the Multi Programming Level (MPL), which indicates the number of queries running concurrently on the system, has been used to control the load on the system. To avoid system underload or overload, MPL must be carefully set. Figure 3(a) shows the throughput curves for three different TPC-H workloads. Each workload has a range of MPL values for which there is no overload or underload. Clearly, different workloads require different optimal MPLs. However, a typical BI batch workload can fluctuate rapidly between long, resource intensive queries, and short, less intensive queries. This requirement makes it very challenging, if not impossible for a human (or a system) to keep the workload in an optimal region using the MPL setting.

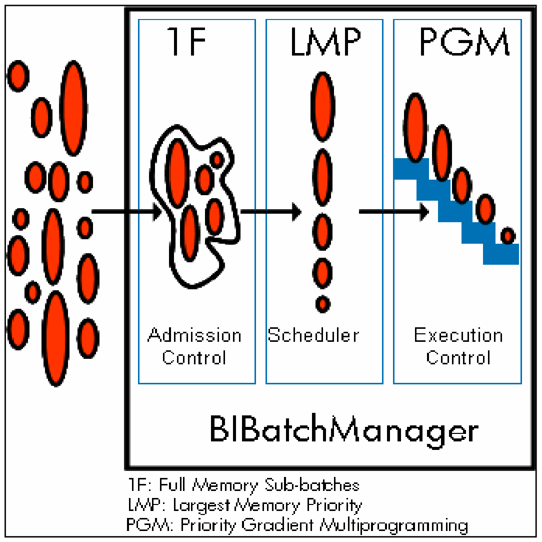
We have created a BI Batch Manager, which is a database workload management system for running batches of BI queries. The BI Batch Manager has the following salient points:

1. It employs new way of executing BI queries, Priority Gradient Multiprogramming (PGM), which automatically protects against overload.
2. The scheduling algorithm, Largest Memory Priority (LMP) further enhances the stability of execution.
3. Estimated memory is used as a basis for admission control in EDWs.

[1]We ensure "reasonableness" by stipulating that no child node has less than 25% examples of the parent node.



(a)


1F
LMP
PGM
Admission Control
Scheduler
Execution Control
BIBatchManager
1F: Full Memory Sub-batches
LMP: Largest Memory Priority
PGM: Priority Gradient Multiprogramming

(b)

Figure 3: (a) Throughput as a Function of MPL (b) Component Design for BI Batch Manager

A common way of looking at throughput is by means of throughput curves where the throughput is plotted against the MPL on the system. 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. 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, increasing the MPL by even one, can cause severe performance deterioration rather than a gradual decline in performance. This is because of thrashing, a problem that is inherent in all virtual memory, multiprogramming systems.

Our focus is on addressing the problem of automatically managing BI batch workloads, so that we are in the optimal region of the throughput curve, where there is no underload or overload. We focus on issues as follows:

1. Identify a manipulated variable whose predicted value is suitable for BI workload management.
2. Make the execution of the queries stable over a wide range of estimation errors of this variable.
3. Schedule the queries so that the system behaves without underload or overload for the admitted batch.
4. Use the manipulated variable for admission control, i.e. admit queries based on an estimated value of the manipulated variable.

Our solution is depicted in Figure 3(b). The BI Batch Manager has three primary components, that address the issues highlighted above: (1) *Admission Controller*, (2) *Scheduler* and (3) *Execution Manager*.

We use memory as the variable of choice to manipulate. Whenever there is an over-subscription of memory, there is a potential for serious degradation in performance due to thrashing. A workload thrashes whenever its cumulative peak memory requirement per CPU exceeds the available memory per CPU. Overload behavior can be predicted more accurately with memory rather than with MPL. Thus, in contrast to MPL, memory behaves much more predictably as a manipulated variable.

Current execution control technology centers around Equal Priority Multiprogramming (EPM), in which every query is executed at the same process priority. EPM is robust for a reasonable range of overestimates of the memory requirement. However, it is very unstable for underestimates, which will result in a sudden drop in throughput as the size of the workload memory increases beyond the available memory at runtime.

To overcome the sensitivity to thrashing for underestimates of memory[2], we introduce Priority Gradient Multiprogramming (PGM). In PGM, queries are executed at different process priorities such that a gradient of priorities is created. PGM requires that the operating system supports preemptive priority scheduling, which is standard on many commercial systems, including Linux and NSK. This results in queries asking for, and releasing resources at different rates. This solution has proven to be very effective in protecting against overload. Ironically, PGM is an effective execution control mechanism because it uses the priority gradient to distribute memory to queries in an unfair way. The priority gradient allows the operating system to automatically allocate resources to queries down the gradient without any over-allocation of resources and keeps the system in an optimal range of execution.

Since the throughput penalty for being in the overload region is much higher than being in the underload region, we designed a scheduler that stabilizes the system for memory underestimation errors. We call our ordering scheme: Largest Memory Priority (LMP). The query with the largest memory requirement is given the highest priority.

For admission control we create batches whose estimated memory requirement is equal to the available memory per CPU on the system. The whole batch is divided into these sub-batches using the standard technique of bin packing called First Fit Decreasing(FFD). Once a batch finishes, a new batch is admitted for execution. Here, we can have various definitions of what is meant by a batch being "done". For example, one definition could be that a batch is done when 90% of the queries in the batch are done or if all the memory is released. Estimation errors are compensated for, by our PGM execution control mechanism as described previously.

Our experiments have shown that the BI Batch Manager (BIBM) does not cause thrashing for memory estimates that underestimate the memory requirement to be a third of what it actually is. Similarly, it does not go into underload if we overestimate the memory to upto three times the actual memory requirement. Also, most DBMS put bounds on the memory available to BMOs. This reduces the extent to which memory can be underestimated. Finally, in our paper [17] we give a statistical proof that shows that errors in memory estimates of a batch are much less than the errors in the memory estimates of the individual queries. Thus, the main contribution of BIBM is that it makes the system tolerant of memory estimation errors. BIBM compensates upto a factor of 3, or 300% error in the estimate for how much memory a workload is going to need. This is seen as a sufficient margin of error for most practical workloads.

We have done a series of experiments to test various aspects of our BI Batch Manager. For the purpose of experimentation we used a TPC-H workload with three SF values: 50, 100, 200. It was installed on a two Segment (32 Nodes) commercial class Enterprise Data Warehouse, with 8GB physical memory per CPU. We created forty-eight mixed workloads of random sizes by uniform random sampling (with replacement).

We compared the throughput obtained with BIBM with the *EPM Throughput* (throughput achieved when all queries are running with the same priority) and the *Ideal Throughput* (theoretic maximum throughput achieved when CPU utilization is 100%). 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.

In our experiment result shown in Figure 4, BI Batch Manager generally achieved a system throughput of greater than 80% of ideal throughput. There was approximately 4 GB of memory available per CPU during the experimental runs. The workloads were created by first choosing a memory number between 1.33GB and 12GB (approximately 1/3 to 3 times of memory per CPU). Then queries were randomly chosen from TPC-H queries (with replacement) until the memory boundary was reached. More experiment results are available in [17].

[2]If memory is under-estimated, a batch that requires larger amount of memory might be submitted causing thrashing.

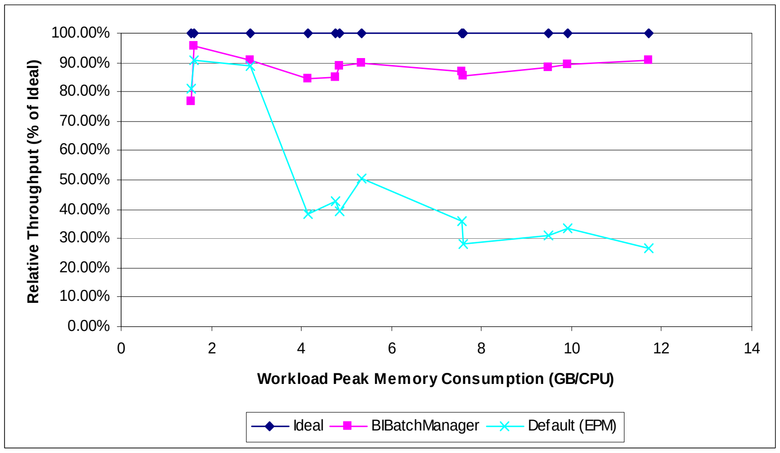


Figure 4: Throughput Results for Workloads from TPC-H SF200

5 Conclusion

In this work we have dealt with predictability and manageability in an effective way. Our results have been validated on real life commercial class EDWs. As a next step we plan to extend manageability to a continuous stream of queries and detect problem queries.

References

[1] BEZ Systems Inc. BEZPlus for NCR Teradata and Oracle environments on MPP machines. http://www.bez.com/software.htm, 1999.

[2] K. P. Brown, M. J. Carey and M. Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. In *VLDB*, pages 328–341, 1993.

[3] K. P. Brown, M. Mehta, M. J. Carey and M. Livny. Towards Automated Performance Tuning for Complex Workloads. In *VLDB*, pages 72–84, 1994.

[4] M. J. Carey, R. Jauhari and M. Livny. Priority in DBMS resource scheduling. In *VLDB*, pages 397–410, 1989.

[5] X. Chen, P. Mohapatra and H. Chen. An admission control scheme for predictable server response time for web accesses. In *WWW*, pages 545–554, 2001.

[6] P. J. Denning, K. C. Kahn, J. Leroudier, D. Potier and R. Suri. Optimal Multiprogramming. *Acta Inf.*, 7:197–216, 1976.

[7] S. Elnikety, E. M. Nahum, J. M. Tracey and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *WWW*, pages 276–286, 2004.

[8] A. Ghosh, J. Parikh, V. S. Sengar and J. R. Haritsa. Plan Selection Based on Query Clustering. In *VLDB*, pages 179–190, 2002.

[9] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[10] M. A. Iverson, F. Özgüner and G. J. Follen. Run-Time Statistical Estimation of Task Execution Times for Heterogeneous Distributed Computing. In *HPDC*, pages 263–270, 1996.

[11] S. Jiang and X. Zhang. TPF: a dynamic system thrashing protection facility. *Softw., Pract. Exper.*, 32(3):295–318, 2002.

[12] A. Kamra, V. Misra and E. M. Nahum. Yaksha: a self-tuning controller for managing the performance of 3-tiered Web sites. In *IWQoS*, pages 47–56, 2004.

[13] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein and S. S. Parekh. Online Response Time Optimization of Apache Web Server. In *IWQoS*, pages 461–478, 2003.

[14] M. Garth. Modelling parallel architectures. http://www.metron.co.uk/papers.htm, 1996. Metron Technology white paper.

[15] V. Markl, G. M. Lohman and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[16] A. Mehta, C. Gupta and U. Dayal. How to Predict the Running Time of Business Intelligence Queries on an Enterprise Data Warehouse. Technical Report HPL-2007-165, HP Labs, October 2007.

[17] A. Mehta, C. Gupta and U. Dayal. BI Batch Manager: A System for Managing Batch Workloads on Enterprise Data Warehouses. In *EDBT (To appear)*, 2008.

[18] A. Mönkeberg and G. Weikum. Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data-Contention Thrashing. In *VLDB*, pages 432–443, 1992.

[19] H. Pang, M. J. Carey and M. Livny. Managing Memory for Real-Time Queries. In *SIGMOD*, pages 221–232, 1994.

[20] Platinum Technology. Proactive performance engineering. http://www.softool.com/products/ppewhite.htm, 1999. Platinum Technology White paper.

[21] R. Eberhard. DB2 Estimator for Windows. http://www.software.ibm.com/data/db2/os390/estimate, 1999. IBM Corp.

[22] K. E. E. Raatikainen. Cluster Analysis and Workload Classification. *SIGMETRICS Performance Evaluation Review*, 20(4):24–30, 1993.

[23] S. Salza and M. Renzetti. Performance Modeling of Paralled Database System. *Informatica (Slovenia)*, 22(2), 1998.

[24] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. M. Nahum and A. Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. In *ICDE*, page 60, 2006.

[25] SES Inc. Solutions for information systems performance. http://www.ses.com/Solution/IS.html, 1999.

[26] M. Stillger, G. M. Lohman, V. Markl and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.

[27] N. Tomov, E. W. Dempster, M. H. Williams, A. Burger, H. Taylor, P. J. B. King and P. Broughton. Analytical response time estimation in parallel relational database systems. *Parallel Computing*, 30(2):249–283, 2004.

[28] G. Weikum, A. Mönkeberg, C. Hasse and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *VLDB*, pages 20–31, 2002.