PQR: Predicting Query Execution Times for Autonomous Workload Management

Chetan Gupta
HP Labs
chetan.gupta@hp.com

Abhay Mehta
HP Labs
abhay.mehta@hp.com

Umeshwar Dayal
HP Labs
umeshwar.dayal@hp.com

Abstract

Modern enterprise data warehouses have complex workloads that are notoriously difficult to manage. One of the key pieces to managing workloads is an estimate of how long a query will take to execute. An accurate estimate of this query execution time is critical to self managing Enterprise Class Data Warehouses.

In this paper we study the problem of predicting the execution time of a query on a loaded data warehouse with a dynamically changing workload. We use a machine learning approach that takes the query plan, combines it with the observed load vector of the system and uses the new vector to predict the execution time of the query.

The predictions are made as time ranges. We validate our solution using real databases and real workloads. We show experimentally that our machine learning approach works well. This technology is slated for incorporation into a commercial, enterprise class DBMS.

1. INTRODUCTION

Many organizations are creating and deploying “enterprise data warehouses” 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 is to manage the complex workload to meet stringent performance objectives. Workload management is the problem of scheduling queries and allocating resources to them so as to meet the performance objectives. Any autonomic data warehouse management system will need to have some estimate of the execution time of a query.

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.

Researchers and practitioners have built increasingly sophisticated analytical models for prediction. However, building an accurate analytical model is difficult especially under varying load conditions. In this paper, we have taken a different approach: the method “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 query optimizer and system load features from the environment on which the queries were run. 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 (we will explain why in Section 3).

The challenges were several since we are interested in not only predicting the time ranges but also in discovering them:

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

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

(iii) (As with all predictive models the accuracy of prediction should be high.

(iv) 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.

978-0-7695-3175-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ICAC.2008.12

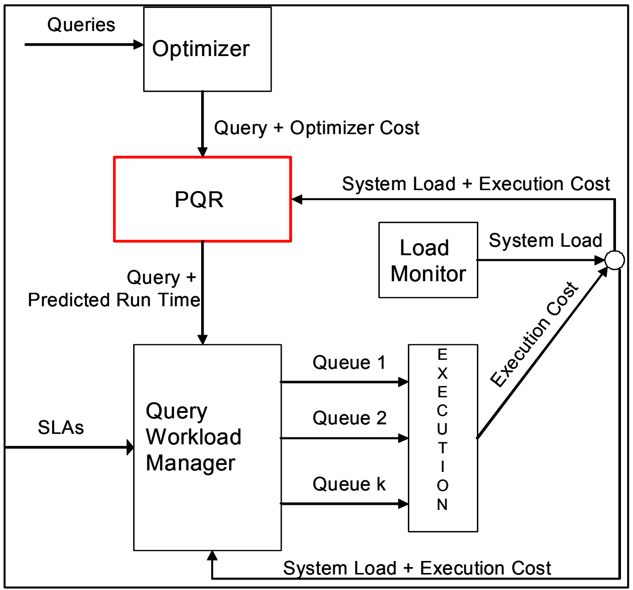

Figure 1: Workload Management System

Figure 1 depicts the architecture of the workload management system. Once the query optimizer outputs an execution plan for a query, the query plan and the optimizer's estimate of the query cost are input to the PQR prediction model. In addition, a load monitor extracts a load feature vector, which is also input to the PQR prediction model. Whenever a new query comes in, PQR 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. (In this paper we do not describe how the workload manager performs this task. Our focus here is to solve the problem of predicting query execution time ranges.)

We show in this paper that our approach works well in practice. In consultations with DBAs who manage workloads, we created a quality metric: a model that can predict at least four "reasonable" ranges of query execution times with an accuracy of greater than 80% is considered acceptable. This metric captures all the key attributes we discussed earlier. (In the following sections we precisely define the idea of reasonableness. In general we aim to create time ranges that are not "too small" or "too large" as a proportion of the overall time range). Over 90% of the PQR Tree models were found to be acceptable (the experimental section provides detailed results for these experiments). Our experiments were conducted under varying load conditions and for very different workloads, including a workload run against the TPC-H benchmark database and several workloads run against a real customer database.

The outline of the paper is as follows. In section 2, we present the related work. In section 3, we specify the problem and present an outline of our approach. In Section 4, we present the details of our approach and the various solution components involved. Section 5 shows the results of our experimental validation. Finally, we conclude in section 6.

Remark: *It is important to note that, the techniques and problems discussed in this paper are general enough to be applied to a variety of systems in which prediction of execution times is required and a history of previous execution times is available.*

2. RELATED WORK

Related work falls into three categories: workload management, query cost estimation, and the use of machine learning in database systems.

The problem of database workload management aimed at self-tuning database systems has been studied in the literature (see Weikum [17] 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. Since OLTP workloads consist of short transactions within a fairly predictable range of execution times, the problem of estimating query execution times was not important in the OLTP case. Similarly, Krompass [8] presents work on managing Quality of Service (QoS) using MPL.

There has been a tremendous amount of work on cost models for query optimization (see for example Graefe [5] 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, where we have plotted the optimizer cost against actual execution times).

Analytical approaches have been used for estimating query response times [13, 16] and there are a few commercial products that use analytical and simulation models to predict query execution times [1, 2, 3, 10, 14]. In [16], Tomov presents a three step process: first, a query is mapped to a collection of low level resource (CPUs, disks, etc.) usage specifications; then, the response times of the individual resources are estimated using queuing network techniques; finally, the overall execution time is estimated from the individual response time estimates. The analytical approaches depend on the creation of resource models which are notoriously complex and difficult to create. Assumptions have to be made (for example, exponential

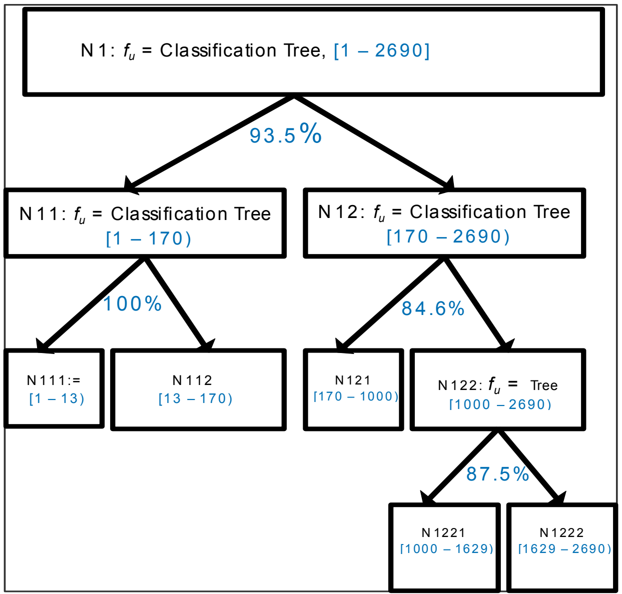
N1: f_u = Classification Tree, [1 – 2690]
93.5%
N11: f_u = Classification Tree [1 – 170)
N12: f_u = Classification Tree [170 – 2690)
100%
84.6%
N111:= [1 – 13)
N112 [13 – 170)
N121 [170 – 1000)
N122: f_u = Tree [1000 – 2690)
87.5%
N1221 [1000 – 1629)
N1222 [1629 – 2690)

Figure 2: Sample PQR Tree

service time distributions) to make the problem more tractable, and hence the results may not be relevant in practice. For instance in Tomov [16], 114 separate experiments were performed for one or two queries with varying arrival rates: of these, 48 have error less than 10%, another 48 have error between 10% and 20%, and 18 have error between 20% and 30%. In contrast, a PQR Tree model is simple and inexpensive to create (typically it takes a couple of minutes to build a PQR Tree), the technique can be used for a variety of realistic workloads (our experiments had up to 200 simultaneous running queries), and the accuracy of prediction is high.

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. It learns the statistics to avoid problems with cardinality and selectivity estimates for future query planning [9], [15]. Raatikainen [12] summarizes some of the early work in using clustering for workload classification. In PLASTIC [4] 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 – predicting the execution times of BI queries on an enterprise data warehouse.

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. For instance, there is work on using machine learning for resource allocation in Grids, or for estimating task completion times in computer systems. Iverson [7] uses a nonparametric regression technique, for estimating the execution time of a task to facilitate run time matching and scheduling in a distributed heterogeneous computing environment. Shivam[19] builds predictive models for performance prediction by combining limited *a-priori* structure with statistical learning.

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.

3. PROBLEM DESCRIPTION AND METHODOLOGY

In this paper we are interested in predicting a time range of the execution time of a query. More precisely: for the execution time (the time it takes for a query to run) of every query q we aim to predict a time range of the form:

$$t_a \le t_q \le t_b$$

Where t_q is the query execution time in seconds and t_a and t_b are the bounds of some predicted interval in which t_q lies.

This prediction is based on historical data. Using the historical data, a solution is obtained for this problem in the form of a binary tree, where every node of the tree represents a time range and the children's time ranges are non-overlapping subsets of the parent's time range.

When a new query q comes in, it traverses down this binary tree from the root node to a leaf l. The time range of the leaf l is the predicted time range for the query q.

We use a machine learning approach to obtaining such a tree. We call this a PQR Tree (Prediction of Query Runtime Tree). We now define a PQR Tree.

Definition 1: *A PQR Tree, denoted by T_s, is a binary tree such that:*

1. *For every node u of T_s, there is an associated 2-class classifier f_u.*
2. *The node u contains examples E_u, on which the classifier f_u is trained.*
3. *f_u is a classifier that decides for each new query q with execution time t_q in [t_{ua} , t_{ub}], if q should go to [t_{ua},t_{ua}+Δ) or [t_{ua}+Δ, t_{ub}], where Δ lies in (0, t_{ub}-t_{ua}).*
4. *For every node u of T_s, there is an associated accuracy, where accuracy is measured as the percentage of correct predictions made by f_u on the example set E_u.*
5. *Every node and leaf of the tree corresponds to a time range.*

Example 1: *In Figure 2, we present a sample PQR Tree. This was obtained in one of our preliminary runs. The classifier, f_u associated with the root node is a classification tree with a time range of [1, 2690] seconds The root node has two children that divide its range into two: [1, 170) and [170, 2690]. The associated accuracy of*

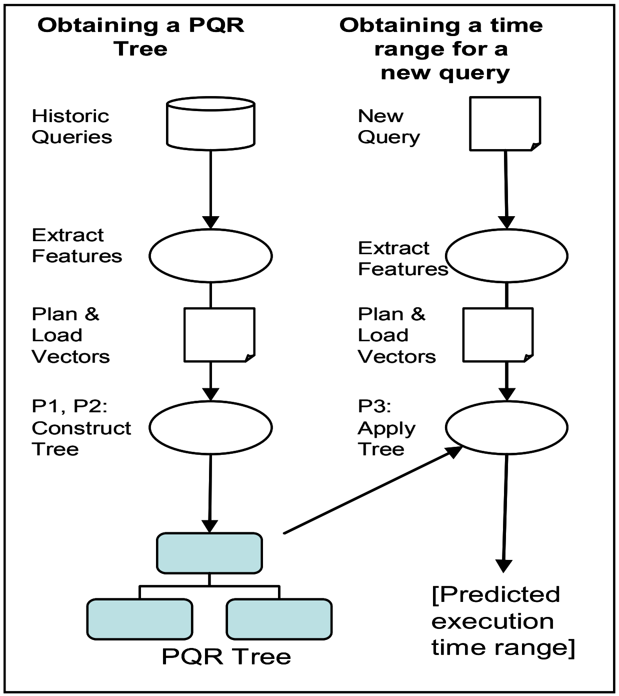

Figure 3: Overall System Diagram Showing the Solution Components

this classifier is 93.5%, i.e., for 93.5 % of the example queries in the root node the classifier was able to predict whether the query's time range was [1, 170) or [170, 2690]. The rest of the nodes can be understood similarly. It can be seen that the PQR Tree has five leaves. The final time ranges are (in seconds): {[1, 13), [13, 170), [170, 1000), [1000, 1629), [1629, 2690]}.

A PQR Tree is a natural solution to the problem of predicting time ranges. PQR Tree is a hierarchical structure and the time ranges are hierarchical, i.e., the children's time range is a subset of parent's time range.

PQR Trees are different from traditional decisions tree [11] which are used for multi-class classification problem. This is because the time ranges are not known prior to the creation of the tree. One plausible method is to divide the overall time range ([1, 2690], in the above example) into a fixed number of ranges or classes *a priori*. We experimented with this approach using both a multi-class classifier and a PQR like approach. In both cases the quality of prediction suffered. Moreover this suffers from an additional drawback of having to fix the number of time ranges *a priori*. Also, if multi-class classifiers considers all the classes at once (as opposed to combining several binary classifiers) the overall optimization problem becomes more expensive to solve [6].

At every node we not only need to find the two sub ranges for the time range of the node but also a classifier that can predict the two ranges. In a PQR Tree, at each node we find a combination of a classifier and two meaningful classes that give us the highest accuracy.

Our experiments have shown that for different time ranges different attributes are good predictors. Another advantage with a PQR Tree approach is that we are able to find different classifiers for different time ranges.

As we grow the PQR Tree, the classes are automatically obtained. With every increase in the depth of the tree, the number of classes increases and smaller ranges of time intervals are obtained.

Remark: *In certain applications the user wants a fixed number of time ranges. In such a scenario any number of pruning or other techniques can be used. We present one such case in our experimental section where we obtain balanced trees.*

4. SOLUTION COMPONENTS

We now present the methodology for constructing a PQR Tree.

As with all machine learning techniques we learn the past behavior of queries in order to predict the future. We have represented the overall schema of our approach in Figure 3.

There are two overall steps:

1. *Obtaining a PQR Tree*: We use historical data of queries run on the system to obtain a PQR Tree. The major sub-steps involved in building the PQR Tree are:
   a. Build a query plan vector for each historical query.
   b. Build a load vector corresponding to each historical query.
   c. Use the feature vectors to build a PQR Tree.
2. *Obtaining a time range for a new query*: Apply the PQR Tree to a new query *q*. By traversing the PQR Tree, predict the time range in which the query lies.

One intuitive way of predicting the execution time on a loaded system would have been to use a PQR tree to predict execution times on an unloaded system. Then build another PQR Tree that combines this unloaded estimate with a load vector to predict the execution time on a loaded system. We tried precisely this approach with very encouraging results. This approach suffers from a serious drawback in that an unloaded system is needed to gather data to build a PQR Tree that predicts execution times. On a real live system this would require isolating the system for testing and thus making it unavailable for normal business use. This makes this approach expensive and hence impractical.

We represent every query as a set of attributes or a feature vector. A feature vector for a PQR Tree is composed of a query plan vector, which describes the query plan, and a system load vector which comprises of the system load experienced by a query. This way the feature vector contains information both for the query itself and the load it will experience. This set of feature vectors is used to build a PQR Tree. We now discuss these in some detail.

4.1. Query Plan Vector

The success of any machine learning approach depends on choosing the right set of attributes. We followed an exhaustive approach. By looking at various query plans we enumerated all the common elements like cardinality, optimizer cost etc. Then with discussions with domain experts we computed some derived attributes. When building the PQR Tree these attributes are obtained from the query plan.

4.2. System Load Vector

A standard way of looking at the system load is to study CPU characteristics. There are two major drawbacks to this approach:

1. The CPU characteristics tend to be *erratic* over small time intervals. For prediction of query runtimes where there are several small queries, this is a major drawback
2. For a new query we need to predict how the CPU characteristics will be in the future. The CPU characteristics are *unpredictable,* which makes it difficult to use as a system load vector.

To overcome these drawbacks we use MPL (Multi Programming Level) to measure system load. We will restate one definition and then go into the details.

Definition 2: *The 'stretch' of a query is the ratio of the execution time of a query on a loaded system to the execution time on an unloaded system.*

The higher the query or process MPL (number of concurrent queries or processes running on the system), the greater the stretch and hence the longer the query takes to execute. Intuitively, if a query takes x seconds to run by itself, then when running on a fully loaded system with say k similar queries, it will take kx seconds. Using this intuition, we multiply query plan features with the query MPL the query q experiences as it enters the system to obtain some features of the load vector.

4.3. Constructing the Tree

For every node of the PQR Tree we not only need to obtain the classifier but we also need to compute the time ranges of the children. Recall that the time ranges of the children are non-overlapping subsets of the time range of the parent. We use the distribution of the execution time of queries on the system to determine the two time ranges at each node in the solution tree.

To obtain the partition of the time range at each node, we take each query in the training example set and arrange these execution times in ascending order. We then compute the time gaps (difference) between every two successive execution times: $\delta_i = (T_{i+1}-T_i)/T_i$, where, T_i is the execution (elapsed) time of the i^{th} query.

We take a constant n_{gap} largest time gaps as the points at which to partition the time range into two. We motivate this further with an example:

Example 2: *Let's say our training example set consists of 10 queries and their execution times, in ascending order are: Execution times = {2, 2, 3, 3, 5, 5, 9, 14, 16, 17}*
The differences as computed using the formula above between two successive execution times are Δ *= {0, 0.5, 0, 0.66, 0, 0.8, 0.55, 0.14, 0.06}. Lets say* n_{gap} *= 2. Then two largest differences are {0.8, 0.66}. They correspond to two partitions: [{2, 2, 3, 3, 5, 5}, {9, 14, 16, 17}] and[{2, 2, 3, 3}, {5, 5, 9, 14, 16, 17}].*

As mentioned earlier an important consideration in splitting is to obtain "reasonable" time ranges. We want to obtain time ranges that are not too small or too large. Very large ranges might not be useful. Numerous small ranges can lead to a loss of accuracy - If the time ranges are roughly equal in size, the depth of the tree at which time ranges are a reasonable fraction of the overall time range are obtained is less. This means a lesser loss of accuracy. Specifically:

Definition 3: *When partitioning a set S into subsets, the partition is called* n_{skip} *- reasonable, if no subset contains less than* n_{skip} *percentile of the number of points in S.*

From a machine learning perspective, if a children's time range is a small fraction of the parent's time range it can lead to over-fitting, i.e. the small class could exist only in the examples on which the classifier was created. To obtain reasonable time ranges and since our sets correspond to ranges, we skip n_{skip} percentage values from the beginning and end of the time ranges as candidate points for partition. For example, if we look at Example 2, and n_{skip} = 20. Then we will not partition at the first two or the last two values (0.2 * 10) , i.e., we do not partition the execution time range *{2, 2, 3, 3, 5, 5, 9, 14, 16, 17}* as*: [{2, 2}, {3, 3, 5, 5, 9, 14, 16, 17}] or [{2, 2, 3, 3, 5, 5, 9, 14}, { 16, 17}].* More specifically, the procedure for determining the best time ranges and the classifier for each node is as follows:

Procedure P1: Construct a node:

1. *Take all the queries in the training example set E and arrange them in list L of ascending order of execution time T, skipping* n_{skip} *percentage values from the beginning and end of the series of execution times,*
2. *Compute the deltas (time gaps) in this list L:* $\delta_i = (T_{i+1}-T_i)/T_i$*, where,* T_i *is the execution time of the* i^{th} *query.*

3. *Obtain a set Δ of n_{gap} largest δ_i .*
4. *For each $\delta_i \in \Delta$, partition the query set into two subsets at execution times that correspond to δ_i .*
5. *Fix a set of classifiers F. For every classifier $f \in F$, train f to predict the two classes of queries obtained from every $\delta_i \in \Delta$ as described in the previous step. Compute the accuracy for each such prediction.*
6. *Choose the classifier f and the time range as the combination that gives the highest value of accuracy in step 5.*

Now that we have a procedure for constructing a node, we use this procedure recursively to construct the overall PQR tree.

Once we have the feature vectors, it is easy to create the PQR Tree using P1 recursively - For every node (starting at the root node) apply P1 recursively (Recall that P1 creates new nodes) till a termination condition is reached. This procedure for creating the PQR Tree is called P2.

We use three stopping criterion and introduce three thresholds:

(i) With smaller ranges the accuracy of prediction goes down and we stop when the accuracy of prediction falls below a threshold - $min_{accuracy}$.

(ii) The quality of the predictor depends on the number of examples it is trained on. Hence, when the number of examples in a node falls below a threshold we do not further subdivide the nod - $min_{Example}$

(iii) If the time range is too small no additional information might be gained by further subdividing it, hence when that falls below a threshold we do not further subdivide the node- $min_{IntervalSize}$.

The recursive creation causes the number of ranges to increase while ensures a minimum threshold of accuracy.

Remark: *In our experiments it takes approximately four hundred examples to obtain a tree with desirable accuracy.*

Remark*: In our experiments we use $n_{skip} = 2$, i.e., we skip 25% of the points from either end of our time range as possible separation points.*

4.4. Applying a PQR Tree

Once we have constructed a PQR Tree using the procedures in the previous section, we can predict the time interval for the execution time of a new query. We decompose the query and the system load into the feature vector described earlier. We then apply the classifier at each node to this feature vector, to determine whether this new query belongs to the left or the right child. We do this recursively at each node until we reach a leaf of the PQR tree. The leaf has a time range associated with it and this becomes the predicted execution time of the new query. This procedure for applying a PQR Tree is called P3.

As we traverse down the tree the overall accuracy monotonically decreases since the overall accuracy of prediction of a query runtime at a node is not only dependent on the accuracy of the prediction at the node but the accuracy of all its predecessors.

Remark: *To build a PQR tree on a desktop with 2.00 GHz Pentium M processor with 1.00 GB of RAM it takes a couple of minutes for a query workload of approximately 1000 queries and this step needs to be done only periodically. For a new query it is a matter of applying the rules PQR generates and it is almost instantaneous.*

4.5. Updating the PQR Tree

Since a PQR Tree is built on historical data there might be a need to update the tree periodically. The accuracy of the predictions can be monitored and if they fall below a threshold a new PQR Tree should be created. For the period leading up to the creation of a new tree the query data and the load data should be stored.

4.6. Predicting the Execution Time of a Workload

Although the primary purpose of the work is to predict the execution time of a single query we can use the results to predict the total execution time of a workload. For that we first need the execution time of a single query. The execution time of a single query can be computed either by:

1. Taking the mean of the interval the query lies in as the execution time of the query.
2. Fit a regression line in each interval that predicts the execution time of a query.

	MPL = 10		MPL = 8		MPL = 6		MPL = 4	
	Accuracy	Buckets	Accuracy	Buckets	Accuracy	Buckets	Accuracy	Buckets
Test								
1	85.585586	6	89.344262	6	88.571429	5	91.176471	6
2	80	9	73.831776	9	83.157895	11	88.888889	9
3	84	5	84.210526	6	85.714286	8	87.610619	8
4	85.185185	9	87.5	3	80.869565	10	88.461538	10
5	75.490196	11	88.043478	8	82.653061	9	79.381443	8
6	88.77551	7	84.090909	10	87.5	7	91.304348	9
7	81.632653	6	92.66055	9	80.412371	10	84.782609	8
8	75.247525	6	67.567568	8	85.542169	10	90.425532	7
9	85.454545	8	81.132075	9	91.891892	5	89.07563	6
10	75.229358	8	87.628866	7	81.521739	5	82.978723	6
Avg	**81.6600558**	**7.5**	**83.601001**	**7.5**	**84.7834407**	**8**	**87.4085802**	**7.7**

Table 1: 40 Runs of Workload #1 Under 4 Different MPLs (10 test runs each) Against a TPCH Database

Both of the methods above have the disadvantage of being inaccurate. Now we present a statistical analysis that shows that the errors in the prediction of precise execution times is not additive but rather the prediction of the execution time of a workload will be more accurate than the prediction for an individual query.

For the purpose of analysis, we assume that the estimation errors are distributed normally. Let there be n queries in the workload. Let the execution time prediction for the i^{th} query, q_i in a workload W be 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* $Q(M) = \Sigma Q(m_i) \sim N(\mu, \sigma^2)$, where M can be understood as the total execution time of a workload:

$\Sigma N(\mu_i,\sigma_i^2) = N(\mu,\sigma^2)$ 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 execution time 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 execution time of q_i and M is the total execution time of the workload W and the number of queries in W is n.

Theorem 1: *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 2: *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.

5. EXPERIMENTAL RESULTS

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:

1. In the first set we ran queries against the TPCH benchmark database on a 32 node system for various MPLs.
2. In the second set we ran Customer X queries at various MPLs on a 256 node system.

We refer to each MPL setting as a run. For each run we randomly divided the queries into two sets, one containing 90% of the queries and the other containing the remaining 10% of the queries. We use the 90% queries to build a PQR Tree and for every query in the last 10% we test the PQR Tree by predicting its execution time range. The percentage of queries that were correctly classified in this last 10% is reported as the accuracy results in the following sections. For every run we do this ten times, each one being called a test. We introduce this randomness to mitigate the effects of any one combination of training and a test set.

We used the open source Java package, WEKA (Waikato Environment for Knowledge Analysis) [18] to create a PQR Tree generator.

We use n_{gap} *= 5 and* n_{skip} *= 25 percentile* (5 best splits with no less than 25% of the queries in a class). We used two basic classifiers: Nearest Neighbor and Classification Tree. For the Nearest Neighbor classifier, we used 2 flavors: the 1-nearest neighbor classifier and the 3-nearest neighbor classifier. For the Classification tree, we used the C4.5 algorithm [11] with six different combinations of parameters. Thus, in total, we use eight different flavors of classifiers. These eight classifiers in combination with five different partitions give us a total of 40 combinations of time range partitions and classifiers to try at each node.

We now describe in detail each experiment. Note that for company confidentiality reasons, we will not be disclosing customer names or any actual execution times.

5.1. TPCH

We ran a thousand queries (Workload #1) against a SF = 1 TPCH database. The database was installed on a machine with 32 Intel Itanium 2 1.6 GHz processors with 3 MB cache and 128 GB memory, 112 146GB Fiber Channel 15K RPM disk drives, and 3 HP Rack 10000 G2 Series. User data storage is 3.75 TB.

The queries were generated automatically and ran against all the tables. They include joins and order-bys. We looked at four query MPL values: 4, 6, 8 and 10. For each of these MPLs we ran our set of 1000 queries. Sixteen of these queries were TPCH benchmark queries.

We present the results in Table 1. For every MPL experiment, there are ten different tests. In each test, the 1000 queries were randomly divided into 900 of queries for building the tree and the remaining 100 were used to produce the accuracy results. For every test we have recorded two metrics, the accuracy of predicting the right time range and the number of time ranges.

It can be seen that for all four MPLs on an average we are able to accurately predict the correct time range for at least 84% of the queries. The number of time ranges varies from 5 to 11, with the average being greater than 7. These results were obtained *with* $min_{IntervalSize}$ *= 5 seconds,* $min_{Examples}$ *= 30,* $min_{accuracy}$ *= 0.82.*

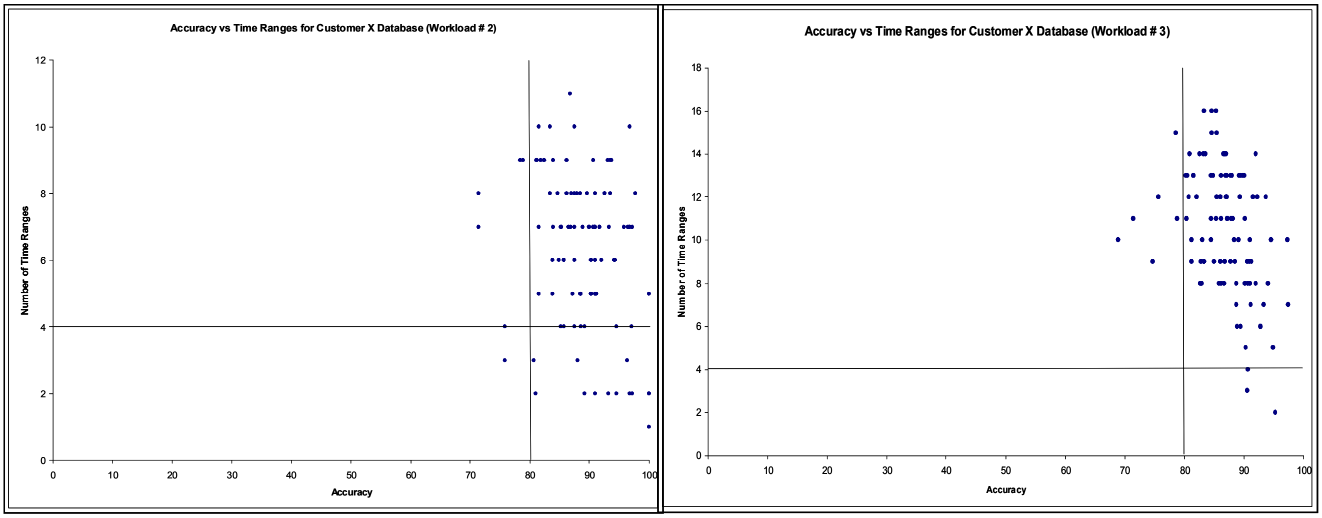

Figure 4: Accuracy and Time Range for Workload #2 and Workload #3. Each workload is run 100 times under 10 different MPLs

MPL	AVG
2	83.87
8	80.31
16	78.66
32	89.36
48	84.59
64	88.81
96	88.44
128	88.14
160	90.55
192	89.34
228	91.72
256	92.04

Table 2: A Table with Results for Average Accuracy, for Ten Test Runs with a Fixed Number of Time ranges on Various MPLs Against a Customer X Database

5.2. Customer X

This set of experiments for Customer X was run on a database that was installed on a machine with 256 Intel Itanium 2 1.6 GHz processors with 3 MB cache and 2 TB memory, 1344 146 GB Fibre Channel 15K RPM disk drives, and 23 HP Rack 10000 G2 Series. User data storage is 56 TB. We tested with two different workloads for Customer X.

5.2.1. Workload # 2

This time we took a set of actual BI (Business Intelligence) queries run in a day by one of our customers. They include both adhoc queries and canned reports. There were a total of 500 queries in this data set. This experiment consisted of 100 different tests (10 query MPL runs and 10 tests per MPL). The MPLs were: 8, 16, 32, 48, 64, 96, 128, 192, 224, 256.

The tests were conducted similarly to that described before. The results are presented in Figure 4 (left plot). Here we have plotted the accuracy of the prediction on the x-axis and the number of time ranges on the y-axis. These results were obtained *with* $min_{IntervalSize}$ *= 5 seconds,* $min_{Examples}$ *= 30*, $min_{accuracy}$ *= 0.82.*

It can be seen that for 94 of the tests we had higher than 80% accuracy and thirteen tests had less then four predicted time ranges. The average accuracies and average number of time ranges for the ten runs were (in order of MPL and in accuracy percentages): (84.41, 3.5), (90.76, 4.2), (89.65, 4.4), (85.96, 7.3), (88.81, 7.5), (91.87, 7.3), (88.25, 7.5), (88.11, 6.2), (91.32, 7.7) and (90.37, 8.0).

5.2.2. Workload # 3

We took another set of queries from Customer X which included a number of smaller queries. This test consisted of 100 different tests (10 query MPL runs and 10 tests per MPL). The MPLs were: 8, 16, 32, 48, 64, 96, 128, 192, 224, 256. These results were obtained *with* $min_{IntervalSize}$ *= 5 seconds,* $min_{Examples}$ *= 30*, $min_{accuracy}$ *= 0.82.*

Again the mode of experimentation was the same as described before. The results are shown in Figure 4 (right plot).

It can be seen that for 94 of the tests we had higher than 80% accuracy and only three tests had less than four time ranges. The average accuracies and average number of time ranges for the ten runs were (in order of MPL and in percentages): (87.57, 6.6), (90.22, 8.6), (84.41, 9.0), (85.71, 10.5), (84.30, 12.6), (85.45, 13.3), (87.72, 10.4), (85.96, 11.2), (90.25, 10.8) and (85.9, 12.5). It can be seen that we obtained high levels of accuracy and a high number of time ranges.

5.3. Quality of Results

By construction, we have ensured that our classes are n_{skip}-reasonable. Now, there are two metrics of interest, the number of time ranges and the accuracy of prediction. For the best results we want a high number of ranges and a high accuracy. We call our results of acceptable quality if we get more than four time ranges while maintaining an accuracy of at least 80 % for each node in the PQR Tree. The four time ranges can be understood as Small, Medium, Large and Huge, a distinction very often made by users and DBAs for workload management purposes.

In Workload # 1 with TPCH database 90% were of acceptable quality. In Workload # 2, 84% (84 out of the 100 experiments) were of acceptable quality. The unacceptable quality was mostly for the lower MPLs. For Workload # 3, 92% experiments were of acceptable quality. For Workload # 2 and Workload # 3 we have plotted the results in Figure 5. The desired results should be in the top-right quadrant.

In a variation we wanted to have four time ranges and a balanced tree. We grew the tree to a depth 2 ensuring that both the root's children have two children of their own.

We tested this approach on Customer X data with the second set of query values (Workload #3) as mentioned before. The results are presented in Table 4 where for each MPL we have taken an average of all the accuracies over the ten different tests. The results as can be seen in table are extremely encouraging, with the overall average accuracy of over 87%.

5.4. Discussion

To establish the robustness of our approach we have experimented with various workloads and two different hardware configurations on a commercial, enterprise class DBMS. We took sets of queries from two different domains, one from a customer and one set of queries against a TPCH database.

For each of these sets of queries we varied the MPLs to achieve different workloads. For the TPCH set of queries we had four different MPLs and for the two customer sets we had ten different MPLs for a total of 24 different workloads.

In addition to the above, we experimented with ten different workloads for the customer database for the experiment with a fixed number of time ranges.

The results were encouraging over all these different workloads and configurations and we believe that the approach would work under varying workloads and hardware configurations.

6. CONCLUSIONS

Being able to predict how long a query will take to run is key to autonomous workload management – a machine can design better admission control policies, better scheduling policies, perform better resource allocation and perform better load balancing. .

We have devised a novel machine learning technique of accurately predicting query ranges. This technique, called PQR, uses a machine learning approach to make its predictions. It first decomposes a query into a plan vector. It also decomposes the load on a system into a load vector. It combines the plan vector with the load vector and uses historical knowledge to estimate how long the query will take to run.

The techniques that have been outlined in this work are not limited to predicting query execution time. They can be easily used for predicting the execution times of in other scenarios such as web services.

We have validated the PQR technique on synthetic workloads as well as on actual workloads taken from real-life customers. We have run hundreds of experiments, where each experiment is a workload of hundreds of queries, running on large enterprise data warehouses. The PQR technique has consistently achieved an accuracy of prediction which is greater than 80%, with at least 4 time ranges and is slated for incorporation into HP's massively parallel, enterprise class DBMS product.

In our work, so far, we have looked for similar queries but have ignored any time of day considerations. In the future, we will be using time of day patterns as well. Another direction of future work is around learning data skew and around sudden changes in workloads.

7. REFERENCES

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

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

[3] M. Garth, "Modelling parallel architectures", http://www.metron.co.uk/papers.htm#PARA, 1996.

[4] A. Ghosh, J. Parikh, V. Sengar and J. Haritsa, "Plan Selection based on Query Clustering", *Proc. of 28th Intl. Conf. on Very Large Data Bases (VLDB)*, August 2002.

[5] G. Graefe, "Query Evaluation Techniques for Large Databases", ACM Computing Surveys, June 1993, ACM Press.

[6] C-W Hsu and C-J. Lin, "A comparison of methods for multi-class support vector machines", *IEEE Trans. on Neural Networks, 13:415{425}, 2002.*

[7] M. A. Iverson, F. Ozguner and G. J. Follen. "Run-time statistical estimation of task execution times for heterogeneous distributed computing", Proc. of 5th IEEE International Symposium on High Performance Distributed Computing, 1996. Volume , Issue , 6-9 Aug 1996 Page(s):263 – 270

[8] S. Krompass, D. Gmach, A. Scholz, S. Seltzsam, and A. Kemper "Quality of Service Enabled Database Applications", Service Oriented Computing - ICSOC 2006: Fourth International Conference on Service Oriented Computing, Dec. 4 - 7, 2006, Chicago, Illinois, USA

[9] V. Markl, G. M. Lohman, and V. Raman, "LEO: An autonomic query optimizer for DB2 ", IBM Systems Journal, Autonomic Computing, 42 (1), 2003.

[10] Platinum Technology, "Proactive performance engineering", , http://www.softool.com/products/ppewhite.htm, 1999.

[11] J. R. Quinlan, "C4.5: Programs for Machine Learning", Morgan Kauffman Publishers, San Mateo, California, 1993.

[12] E. E. K. Raatikainen, "Cluster Analysis and Workload Classification", Performance Evaluation Review, Vol. 20 (4), May 1993, 24-30.

[13] S. Salza and R. Tomasso, "Performance Modeling of parallel database systems", Informatica 22, 1998, 127-139.

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

[15] M. Stillger, G. Lohman, V. Markl, and M. Kandil, "LEO-DB2's Learning Optimizer", Proc. of the 27th International Conference on Very Large Databases (Sept. 2001), pp. 19–28.

[16] N. Tomov, E. Dempster, M. H. Williams, A. Burger, H. Taylor, P. J. King and P. Broughton, "Analytical response time estimation in parallel relational database systems", *Parallel Comput.* 30, 2 (Feb. 2004), 249-283

[17] G. Weikum, C. Hasse, A. Moenkeberg and P. Zabback: "Self-tuning Database Technology and Information Systems: from Wishful Thinking to Viable Engineering", Proc. *28th VLDB, Hong Kong, 2002.*

[18] I. H. Witten and E. Frank, "Data Mining: Practical machine learning tools and techniques", 2nd Edition, Morgan Kaufmann, San Francisco, 2005.

[19] P. Shivam, S. Babu and J. S. Chase, "Learning application models for utility resource planning", Proc. ICAC 06, June 06, 255-264.