Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning

Archana Ganapathi[†], Harumi Kuno[§], Umeshwar Dayal[§], Janet L. Wiener[§],
Armando Fox[†], Michael Jordan[†], David Patterson[†]

[†]*Computer Science Division, University of California at Berkeley*
Berkeley, CA USA
`{archanag, fox, jordan, pattrsn}@cs.berkeley.edu`

[§]*Hewlett-Packard Labs*
1501 Page Mill Road, Palo Alto, CA USA
`firstname.lastname@hp.com`

Abstract — **One of the most challenging aspects of managing a very large data warehouse is identifying how queries will behave *before they start executing*. Yet knowing their performance characteristics — their runtimes and resource usage — can solve two important problems. First, every database vendor struggles with managing unexpectedly long-running queries. When these long-running queries can be identified *before* they start, they can be rejected or scheduled when they will not cause extreme resource contention for the other queries in the system. Second, deciding whether a system can complete a given workload in a given time period (or a bigger system is necessary) depends on knowing the resource requirements of the queries in that workload.**

We have developed a system that uses machine learning to accurately predict the performance metrics of database queries whose execution times range from milliseconds to hours. For training and testing our system, we used both real customer queries and queries generated from an extended set of TPC-DS templates. The extensions mimic queries that caused customer problems. We used these queries to compare how accurately different techniques predict metrics such as elapsed time, records used, disk I/Os, and message bytes.

The most promising technique was not only the most accurate, but also predicted these metrics simultaneously and using only information available prior to query execution. We validated the accuracy of this machine learning technique on a number of HP Neoview configurations. We were able to predict individual query elapsed time within 20% of its actual time for 85% of the test queries. Most importantly, we were able to correctly identify both the short and long-running (up to two hour) queries to inform workload management and capacity planning.

I. Introduction

As data warehouses grow in size and queries become more and more complex, predicting how queries will behave — how long they will run, how much CPU time they will need, how many disk I/Os they will incur, how many messages they will send, etc. — becomes more and more difficult. Yet identifying these performance characteristics *before the query starts executing* is at the heart of several important decisions.

- *Workload management*: Should we run this query? If so, when? How long do we wait for it to complete before deciding that something went wrong (so we should kill it)?

- *System sizing*: How big a system (how many CPUs, how many disks, how much network bandwidth) is needed to execute this new customer workload with this time constraint?

- *Capacity planning*: Given an expected change to a workload, should we upgrade (or downgrade) the existing system?

For example, with accurate performance predictions, we would not even start to run queries that we would later kill because they caused too much resource contention or did not complete by a deadline. However, sources of uncertainty, such as skewed data distributions and erroneous cardinality estimates, combined with complex query plans and terabytes (or petabytes) of data make performance estimation hard. Understanding the progress of an executing query is notoriously difficult, even with complete visibility into the work done by each query operator [1].

Our goal was to build an accurate prediction tool for both short and long-running queries, that used only information available at compile-time as input. We ran a wide range of queries on an HP Neoview four processor system and multiple configurations of a Neoview thirty-two processor system (varying the number of processors used) to gather training and testing data. We then evaluated the ability of several techniques, including machine learning techniques, to predict multiple performance characteristics of these queries. The key feature of our chosen technique is that it finds multivariate correlations among the query properties and query performance metrics on a training set of queries and then uses these statistical relationships to predict the performance of new queries.

We show that we are able to predict elapsed time within 20% of the actual time for at least 85% of the test queries. Predictions for resource usage such as disk I/O, message counts, etc., are similarly accurate and are very useful for explaining the elapsed time predictions. These resource predictions enable solving the system sizing and capacity planning problems.

In the remainder of the paper, we first present a high-level overview of our approach and its objectives in Section II. We

next discuss related work in Section III, including a discussion of why we chose to investigate a machine learning based approach. We describe our experimental set-up in Section IV and present an overview of the prediction techniques that we considered in Section V. In Section VI we describe the heuristics we used to apply the most promising technique to our problem. Then in Section VII we present the results of experiments to validate our approach. We conclude with a discussion of open questions and future work in Section VIII.

II. APPROACH

Figure 1 illustrates our approach. Given a standard database system configuration, we want the vendor to use training workload runs to develop predictive models and tools. These tools can then be distributed to customer instances of that configuration and used to predict their query performance.
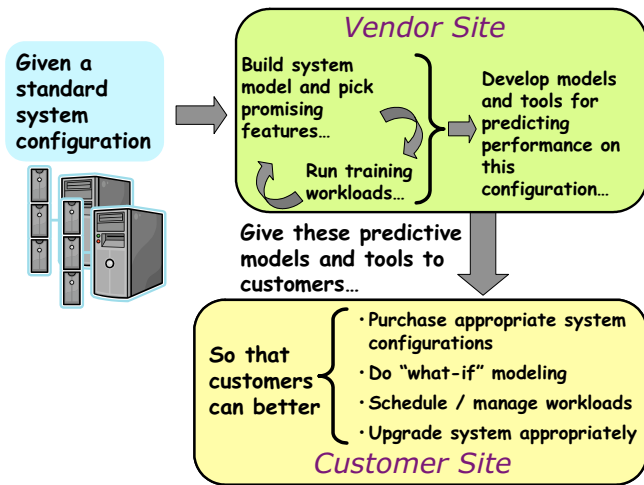

Fig. 1. Predictive models and tools can be built by the database vendor, then shipped to customer sites and used to predict their performance before execution (or even before buying a system).

A. Prediction goals

Our goals for prediction include the following:

1) Predict using only information available before query execution starts, such as the query's SQL statement or the query plan produced by the query optimizer.
2) Predict multiple resource usage characteristics, such as CPU usage, memory usage, disk usage, and message bytes sent, so that we can anticipate the degree of resource contention and ensure the system is adequately configured.
3) Predict accurately for both short and long-running queries. The queries that run in a data warehouse system can range from simple queries that complete in milliseconds to extremely complex decision support queries that take hours to run.
4) Predict performance for queries that access a different database with a different schema than the training queries. New customers will necessarily have different

data than existing customers and we must be able to predict during the sales period.

B. System Modeling vs. Machine Learning

With our approach, we need a model that incorporates important query and system features. We could have tried to build a complete cost-based model of the database system, but found that modeling the work performed by each operator implementation would be very complex and we would need to build a new model (with new cost functions) for each system configuration.

Instead, we chose to extract query features at a high level (as described in Section VI-D) and use them as input to machine learning techniques. The machine learning techniques then "learn" the appropriate cost functions for each system configuration. This approach was simpler, is easier to apply to new configurations and query types, and proved accurate.

III. RELATED WORK

We address the problem of predicting multiple performance characteristics of query performance *prior* to run-time. Prior attempts to predict database performance are all subject to one or more of the following limitations:

- They do not attempt to predict any actual performance metrics: they instead estimate the percentage of work done or produce an abstract number intended to represent relative "cost" (like the query optimizer's cost estimate) [2], [3], [4], [5], [6], [7], [8].
- They attempt to predict only a single performance metric, such as the elapsed time or actual cardinality of the underlying data [9], [10], [11], [1], [12], [13], [14].
- They require access to runtime performance statistics, most often the count of tuples processed. This requirement potentially introduces the significant overhead of needing to instrument the core engine to produce the required statistics [9], [10], [11], [1].

A. Query Optimizers

The primary goal of the database query optimizer is to choose a good query plan. To compare different plans, the optimizer uses cost models to produce rough cost estimates for each plan. However, the units used by most optimizers do not map easily onto time units, nor does the cost reflect the use of individual resources.

Unlike the optimizer, our model bases its predictions on the relative similarity of the cardinalities for different queries, rather than their absolute values. As a result, our model is not as sensitive to cardinality errors. We compare the query optimizer's cost estimates and our performance predictions in Section VII-C.1.

B. Query Progress Indicators

On the opposite end of the spectrum, query progress indicators use elaborate models of operator behavior and detailed runtime information to estimate a running query's degree of completion. They do not attempt to predict performance before

the query runs. Query progress indicators tend to assume complete visibility into the number of tuples already processed by a given query operator [9], [10], [11], [1]. Such operator-level information can be prohibitively expensive to obtain, especially when multiple queries are executing simultaneously. Luo et al. [15] leverage an existing progress indicator to estimate the remaining execution time for a running query (based on how long it has taken so far) in the presence of concurrent queries. They do not address the problem of predicting the performance of a query before it begins execution, nor do they predict metrics other than remaining query execution time.

C. Workload Characterization

Many papers attempt to characterize workloads from web page accesses [2], [3], data center machine performance and temperature [4], and energy and power consumption of the Java Virtual Machine [16], to name a few. In databases, El-naffar [5] observes performance measurements from a running database system and uses a classifier (developed using machine learning) to identify OLTP vs. DSS workloads, but does not attempt to predict specific performance characteristics. A number of papers [6], [7], [8] discuss how to characterize database workloads with an eye towards validating system configuration design decisions such as the number and speed of disks, the amount of memory, etc. These papers analyze features such as how often indexes were used, or the structure and complexity of their SQL statements, but they do not make actual performance predictions.

D. Machine Learning to Predict Database Performance

A few papers use machine learning to predict a relative cost estimate for use by the query optimizer. In their work on the COMET statistical learning approach to cost estimation, Zhang et al [12] use transform regression (a specific kind of regression) to produce a self-tuning cost model for XML queries. Because they can efficiently incorporate new training data into an existing model, their system can adapt to changing workloads, a very useful feature that we have not yet addressed. COMET, however, focuses on producing a single cost value intended to be used to compare query plans to each other as opposed to a metric that could be used to predict resource usage or runtime. Similarly, IBM's LEO learning optimizer compares the query optimizer's estimates with actuals at each step in a query execution plan, and uses these comparisons from previously executed queries to repair incorrect cardinality estimates and statistics [13], [14]. Like COMET, LEO focuses on producing a better cost estimate for use by the query optimizer, as opposed to attempting to predict actual resource usage or runtime. Although a query optimizer that has been enhanced with LEO can be used to produce relative cost estimates prior to executing a query, it does require instrumentation of the underlying database system to monitor actual cost values. Also, LEO itself does not produce any estimates; its value comes from repairing errors in the statistics underlying the query optimizer's estimates.

query type	number of instances	elapsed time (hh:mm:ss)		
		mean	minimum	maximum
feather	2807	30 secs	00:00:0.03	00:02:59
golf ball	247	10 mins	00:03:00	00:29:39
bowling ball	48	1 hour	00:30:04	01:54:50

Fig. 2. We created pools of candidate queries, then categorized them by the elapsed time needed to run each query on the HP Neoview 4 processor system.

The PQR [17] approach uses machine learning to predict ranges of query execution time, but it does not estimate any other performance metrics.

IV. EXPERIMENTAL SET-UP

After deciding to try a machine learning approach for performance prediction, our next task was to evaluate various techniques in order to identify the most appropriate for our purposes. In this section, we describe the database system configurations and the queries we used in these experiments.

A. Database configurations

All of our experiments were run on HP Neoview database systems. Our research system is a four processor machine. A fixed amount of memory was allotted per CPU. Each of the four CPUs has its own disk and data is partitioned roughly evenly across all four disks. Most query operators run on all four processors; the main exceptions are the operators that compose the final result. We used this system extensively for gathering most of our training and test query performance metrics.

We were also able to get limited access to a 32 processor production system. HP Neoview supports configuring one or more queries to use a subset of the processors. We therefore ran our queries on four different configurations of this system, those using 4, 8, 16, and all 32 processors. In all cases, the data remained partitioned across all 32 disks.

B. Training and test queries

Our experiments are intended to predict the performance of both short and long-running queries. We therefore decided to categorize queries by their runtimes, so that we could control their mix in our training and test query workloads. We defined three categories of queries, *feather*, *golf ball*, and *bowling ball*, as shown in Figure 2. (We also defined *wrecking ball* queries as queries that were too long to be bowling balls!) We then describe our training and test workloads in terms of the numbers of queries of each type.

While we defined boundaries between the query types, these boundaries are arbitrary because there were no clear clusters of queries. They simply correspond to our intuition of "short" and "long." Therefore, they can cause confusion in classifying queries near boundaries, as we show later. However, please note that our approach in no way depends on being able to classify queries into these three types.

To produce the queries summarized in Figure 2, we started with the standard decision support benchmark TPC-DS [18]. However, most of the queries we generated from TPC-DS templates (at scale factor 1) were feathers; there were only a few golf balls and no bowling balls. To produce longer queries, we wrote new templates against the TPC-DS database. We based these templates on real "problem" queries that ran for at least four hours on a production enterprise data warehouse before they completed or were killed. (System administrators gave us these queries as examples of queries whose performance they would have liked to predict.) We used these templates, in addition to the original TPC-DS query templates, to generate thousands of queries. We ran the queries in single query mode on the four processor system we were calibrating, then sorted them into query pools based on their elapsed times.

We note that it took significant effort to generate queries with appropriate performance characteristics, and that it was particularly difficult to tell a priori whether a given SQL statement would be a golf ball, bowling ball, or wrecking ball. For example, depending on which constants were chosen, the same template could produce queries that completed in three minutes or that ran for hours.

We also used a set of customer queries against a separate production database, mostly feathers, for our experiment with different schemas for the training and test queries.

V. EVALUATING MACHINE LEARNING APPROACHES

Statistical machine learning techniques first derive a model based on a training set of previously executed data points (queries) and their measured performance. They then predict performance for unknown ("test") data points based on this model. Our goal was to predict *multiple* performance metrics. We evaluated a number of alternative statistical machine learning techniques—some based on independent modeling of the performance metrics and some based on joint modeling of the performance metrics—for our performance prediction system.

A. Regression

A simple starting place is linear regression. We defined covariate vectors, $\mathbf{x} = (x_1, x_2, \ldots x_n)$, where the components x_i are query plan features such as join operator counts and cardinality sums. (We use the same features later in the example in Figure 9.) Each performance metric was considered as a separate dependent variable y. The goal of regression is to estimate the coefficients a_k in the equation $y = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n + \epsilon$, where ϵ is a noise term.

We compare the predicted and actual values for elapsed time and records used in Figures 3 and 4, respectively. The regression models do a poor job predicting these metrics of interest and results for predicting other metrics were equally poor. Many predictions are orders of magnitude off from the actual value of these metrics for each query. Furthermore, regression for elapsed time predicts that several queries will complete in a negative amount of time (e.g., -82 seconds)!
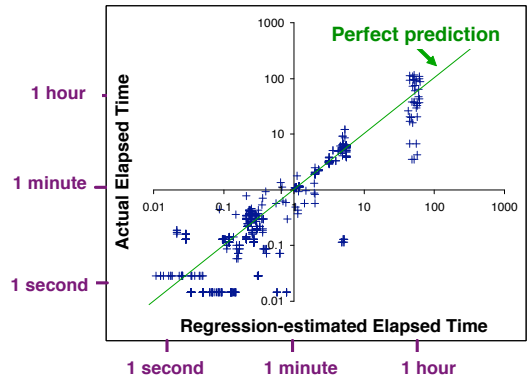

Fig. 3. Regression-predicted vs. actual elapsed times for 1027 training queries. The graph is plotted on a log-log scale to account for the wide range of query runtimes. 176 datapoints are not included in the plot as their predicted times were negative numbers.
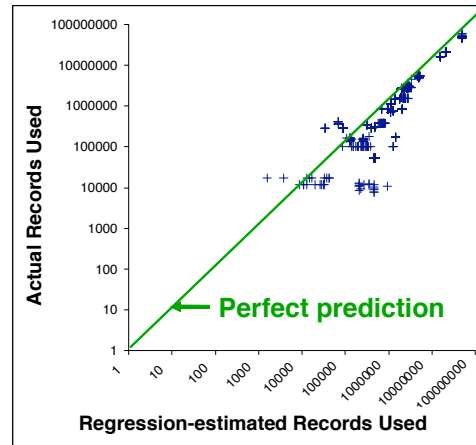

Fig. 4. Regression-predicted vs. actual records used for 1027 training test queries. Note that 105 datapoints had negative predicted values going as low as -1.18 million records.

One interesting fact we noticed is that the regression equations did not use all of the covariates. For example, even though the hashgroupby operator had actual cardinalities greater than zero for many queries, regression for elapsed time assigned zero to the coefficient $a_{hashgroupby}$. However, the features discarded were not consistent for the different dependent variables (such as elapsed time). Thus, it is important to keep all query features when building each model. Furthermore, since each dependent variable is predicted from a different set of chosen features, it is difficult to unify the various regression curves into a single prediction model.

B. Clustering techniques

We were not satisfied with the regression results and turned to other prediction methods. Clustering techniques partition a dataset based on the "similarity" of multiple features. Typically, clustering algorithms work on a single dataset by defining a distance measure between points in the dataset. While partition clustering algorithms such as K-means [19]

can be used to identify which set of points a test query is "nearest" to, it is difficult to leverage these algorithms for prediction as the clustering would have to be performed independently on the query features and the performance features. The points that cluster together with respect to query features do not reflect the points that cluster together with respect to performance.

In other words, we are looking for relationships for *two* datasets, where each dataset is multivariate. The first dataset represents the query features available before runtime, and the second dataset represents the measured performance features available after running each query. We wish to uncover statistical relationships between these two multivariate datasets so that we can predict performance features based on query features.

C. Principal Component Analysis (PCA)

The oldest technique for finding relationships in a single multivariate dataset is Principal Component Analysis (PCA) [20]. PCA identifies dimensions of maximal variance in a dataset and projects the raw data onto these dimensions. While one could use PCA separately within each of our two datasets, this would not identify correlations between the two datasets.

D. Canonical Correlation Analysis (CCA)

Canonical Correlation Analysis (CCA) [21] is a generalization of PCA that considers pairs of datasets and finds dimensions of maximal correlation between pairs of multivariate datasets. This meets our requirements at a high level; in detail, however, classical CCA has important limitations. In particular, CCA is based implicitly on the geometry of Euclidean vector spaces—queries are treated as similar if the Euclidean dot product between their feature vectors (either the query feature vectors or the performance metric vectors) is large. This is an overly restrictive notion of similarity in a domain such as ours, where queries may be similar textually as SQL strings but may have drastically different performance.

E. Kernel Canonical Correlation Analysis (KCCA)

Kernel Canonical Correlation Analysis (KCCA) [22] is a generalization of CCA that replaces Euclidean dot products with *kernel functions*. Kernel functions are at the heart of many recent developments in machine learning, as they provide expressive, computationally-tractable notions of similarity [23]. In the next section, we describe KCCA in more detail and show how we adapt the KCCA framework for database queries.

VI. USING KCCA TO PREDICT PERFORMANCE FEATURES

Since KCCA is a generic algorithm, the most challenging aspect of our approach was to formalize the problem of performance prediction and map it onto the data structures and functions used by KCCA. In particular, we needed to make the following three design decisions:

1) How to summarize the pre-execution information about each query into a vector of "query features," and how

to summarize the performance statistics from executing the query into a vector of "performance features?"
2) How to define a similarity measure between pairs of query vectors so that we can quantify how similar any two queries are, and how to define a similarity measure between pairs of performance vectors (i.e., define the kernel functions) so that we can quantify how similar the performance characteristics of any two queries are?
3) How to use the output of the KCCA algorithm to predict the performance of new queries?

A. Building a KCCA model

Figure 5 illustrates the steps in using KCCA to build a predictive model. The first step is to create feature vectors for all the points in the two data sets that we want to correlate. For query prediction, we need to build a vector capturing query features and a vector capturing performance characteristics for each query in the training data set. We describe choices for these feature vectors in Section VI-D.

We then combine these vectors into a query feature matrix with one row per query vector and a performance feature matrix with one row per performance vector. It is important that the corresponding rows in each matrix describe the same query.
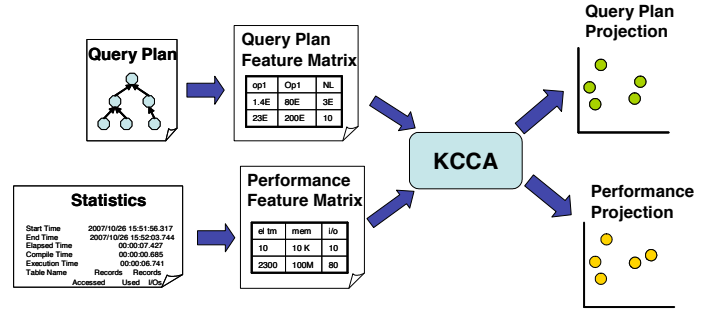

Fig. 5. Training: From vectors of query features and query performance features, KCCA projects the vectors onto dimensions of maximal correlation across the data sets. Furthermore, its clustering effect causes "similar" queries to be collocated.

KCCA then uses a kernel function to compute a "distance metric" between every pair of query vectors and performance vectors. In our work we made use of the commonly-used Gaussian kernel [23]. Denote the set of query vectors and corresponding performance vectors as $\{(\mathbf{x}_k, \mathbf{y}_k) : k = 1, \ldots, N\}$. For a pair of query vectors, $\mathbf{x}_i$ and $\mathbf{x}_j$, we define the Gaussian kernel as follows:

$$k_x(x_i, x_j) = \exp\{-\|x_i - x_j\|^2 / \tau_x\}, \tag{1}$$

where $\|x_i - x_j\|$ is the Euclidean distance and τ_x is a scale factor. We also define a Gaussian vector on pairs of performance vectors, $\mathbf{y}_i$ and $\mathbf{y}_j$, using a scale factor τ_y. While the scaling factors τ_x and τ_y can be set by cross-validation, in our experiments we simply set them to be a fixed fraction of the empirical variance of the norms of the data points.

Specifically, we used factors 0.1 and 0.2 for the query vectors and performance vectors, respectively.

Given N queries, we form an $N \times N$ matrix K_x whose (i,j)th entry is the kernel evaluation $k_x(\mathbf{x}_i, \mathbf{x}_j)$. We also form an $N \times N$ matrix K_y whose (i,j)th entry is the kernel evaluation $k_y(\mathbf{y}_i, \mathbf{y}_j)$. These kernel matrices are symmetric and their diagonals are equal to one.

The KCCA algorithm takes the matrices K_x and K_y and solves the following generalized eigenvector problem:

$$\begin{bmatrix} 0 & K_x K_y \\ K_y K_x & 0 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = \lambda \begin{bmatrix} K_x K_x & 0 \\ 0 & K_y K_y \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} .$$

This procedure finds subspaces in the linear space spanned by the eigenfunctions of the kernel functions such that projections onto these subspaces are maximally correlated. The specific kernel function that we use is the Gaussian kernel function. The linear space associated with the Gaussian kernel can be understood in terms of clusters in the original feature space [23]. Thus, intuitively, KCCA finds correlated pairs of clusters in the query vector space and the performance vector space.

Operationally, KCCA produces a matrix A consisting of the basis vectors of a subspace onto which K_x may be projected (giving $K_x \times A$), and a matrix B consisting of basis vectors of a subspace onto which K_y may be projected, such that $K_x \times A$ and $K_y \times B$ are maximally correlated. We call $K_x \times A$ and $K_y \times B$ the query projection and performance projection, respectively.
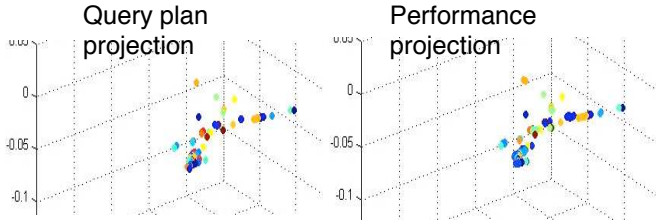

Fig. 6. Projections produced by KCCA: The same color point represents the same query in both projections. The similar locations of these points in each projection indicate that KCCA was able to cluster and correlate "similar" queries.

Figure 6 shows the query plan projection and performance projection produced by KCCA for a set of training queries.

B. Predicting performance from a KCCA model

Figure 7 shows how we predict the performance of a new query from the query projection and performance projection in the KCCA model. Prediction is done in three steps. First, we create a query feature vector and use the model to find its coordinates on the query projection $K_x \times A$. We then infer its coordinates on the performance projection: we use the k nearest neighbors in the query projection to do so (we evaluate choices for k in Section VI-E.2). Finally, we must map from the performance projection back to the metrics we want to predict. Finding a reverse-mapping from the feature space back to the input space is a known hard problem, both
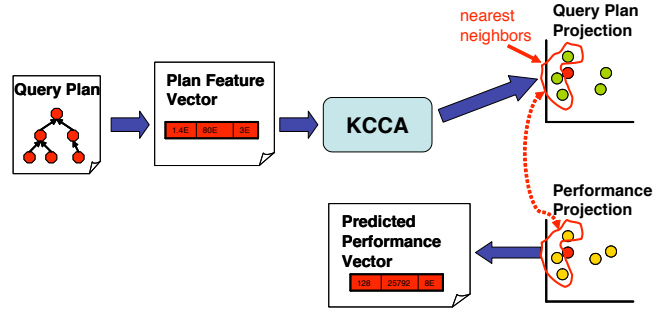

Fig. 7. Prediction: KCCA projects a new query's feature vector, then looks up its neighbors from the performance projection and uses their performance vectors to derive the new query's predicted performance vector.

because of the complexity of the mapping algorithm and also because the dimensionality of the feature space can be much higher or lower than the input space (based on the goal of the transformation function). We evaluate several heuristics for this mapping in Section VI-E.3.

C. Design evaluation

In the rest of this section, we evaluate different design decisions for implementing KCCA. We evaluate the choices for each decision by training a KCCA model with 1027 queries and testing the accuracy of performance predictions for a separate set of 61 queries. The queries contained a mix of feathers, golf balls, and bowling balls, as described in Section IV. All of them were run and performance statistics collected from our four processor HP Neoview system.

We use the a *predictive risk* metric to compare the accuracy of our predictions. Like the R-squared metric often used in machine learning, predictive risk is calculated using the equation:

$$PredictiveRisk = \\ 1 - \sum_{i=1}^{N}(predicted_i - actual_i)^2 / \sum_{i=1}^{N}(actual_i - actual_{mean})^2$$

However, while R-squared is usually computed on the *training* data points, we compute predictive risk on the *test* data points. A predictive risk value close to 1 indicates near-perfect prediction. Note that negative predictive risk values are possible in our experiments since the training set and test set are disjoint. This metric tends to be sensitive to outliers and in several cases, the predictive risk value improved significantly by removing the top one or two outliers.

D. Query and performance feature vectors

Many machine learning algorithms applied to systems problems require feature vectors, but there is no simple rule for defining them. Typically, features are chosen using a combination of domain knowledge and intuition.

Selecting features to represent each query's performance metrics was a fairly straightforward task; we gave KCCA all of the performance metrics that we could get from the HP Neoview database system when running the query. The metrics we use for the experiments in this paper are elapsed time,

disk I/Os, message count, message bytes, records accessed (the input cardinality of the file_scan operator) and records used (the output cardinality of the file_scan operator). The performance feature vector therefore has six elements. (We could easily add other metrics, such as memory used.)

Identifying features to describe each query was a less trivial task. One of our goals was to perform prediction using only data that is available prior to execution. We also wanted to identify features likely to be influential in the behavior of the system because such features are likely to produce more accurate predictions. We evaluated two potential feature vectors: one based on the SQL text of the query and one that condenses information from the query execution plan.
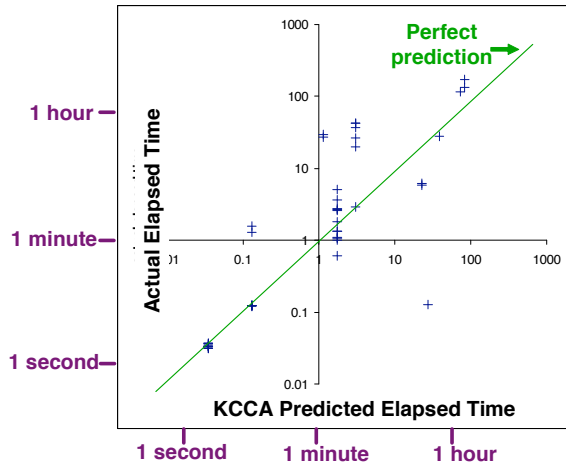

Fig. 8.   KCCA-predicted vs. actual elapsed times for 61 test set queries, using SQL text statistics to construct the query feature vector. We use a log-log scale to accommodate the wide range of query execution times. The predictive risk value for our prediction was -0.10, suggesting a very poor model.

1) SQL feature vector: The first feature vector we tried consisted of statistics on the SQL text of each query. The features for each query were: number of nested subqueries, total number of selection predicates, number of equality selection predicates, number of non-equality selection predicates, total number of join predicates, number of equijoin predicates, number of non-equijoin predicates, number of sort columns, and number of aggregation columns.

The prediction results for elapsed time are shown in Figure 8. While the SQL text is the most readily available description of a query and parsing it to create the query feature vector is simple, the prediction accuracy was fairly low for elapsed time as well as all other performance metrics. One reason for the poor accuracy is that often queries described using exactly the same feature vector were associated with different runtimes. This is because two textually similar queries may have dramatically different performance due simply to different selection predicate constants.

2) Query plan feature vector: Before running a query, the database query optimizer produces a query plan consisting of a tree of query operators with estimated cardinalities. We use this query plan to create a query feature vector. The
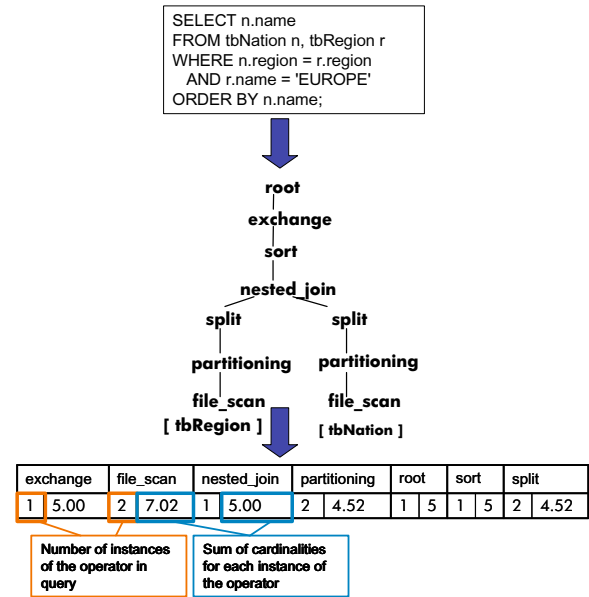

Fig. 9.   Using the query plan to create a query vector: vector elements are query operator instance counts and cardinality sums.

query optimizer produces this plan in milliseconds or seconds, it is not hard to obtain, and we hoped that the cardinality information, in particular, would make it more indicative of query performance. Furthermore, because poor query plan choices contribute to a substantial fraction of support calls, most commercial databases, such as Neoview, provide tools that can be configured to simulate a given system and obtain the same query plans as would be produced on the target system.

Our query plan feature vector contains an instance count and cardinality sum for each possible operator. For example, if a sort operator appears twice in a query plan with cardinalities 3000 and 45000, the query plan vector includes a "sort instance count" element containing the value 2 and a "sort cardinality sum" element containing the value 48000. Figure 9 shows the query plan and resulting feature vector for a simple query (although it omits operators whose count is 0 for simplicity). The intuition behind this representation is that each operator "bottlenecks" on some particular system resource (e.g. CPU or memory) and the cardinality information encapsulates roughly how much of the resource is expected to be consumed.

This feature vector proved to be a better choice than SQL text statistics for predicting the performance of each query (see results in Section VII). In all subsequent experiments, we used the query plan feature vector.

E. Prediction choices

As described earlier, accurate predictions of a previously unseen query's performance depend on identifying the query's nearest neighbors, or at least neighbors that are "similar enough." We address three important issues in using nearest neighbors to predict a test query's performance:

1) What is the metric to determine neighbors and their nearness?
2) How many neighbors do we need to consider when calculating our prediction?
3) How do we weigh the different neighbors in our prediction?

We consider each issue in turn.

Metric	Euclidian Distance	Cosine Distance
Elapsed Time	0.55	0.51
Records Accessed	0.70	-0.09
Records Used	0.98	0.19
Disk I/O	-0.06	-0.07
Message Count	0.35	0.24
Message Bytes	0.94	0.68

TABLE I

COMPARISON OF PREDICTIVE RISK VALUES FOR USING EUCLIDIAN DISTANCE AND COSINE DISTANCE TO IDENTIFY NEAREST NEIGHBORS. EUCLIDIAN DISTANCE HAS BETTER PREDICTION ACCURACY.

1) What makes a neighbor "nearest"?: We considered both Euclidian distance and cosine distance as the metric to determine the nearest neighbors. Since we are projecting our vectors onto a higher dimensional subspace, it seemed useful to consider the directional nearness. While Euclidian distance captures the magnitude-wise closest neighbors, cosine distance captures direction-wise nearest neighbors. Table I compares the cosine distance function with the Euclidian distance function for computing nearest neighbors in the query projection. Using Euclidian distance yielded better prediction accuracy than using cosine distance, as shown by the consistently higher predictive risk value.

Metric	3NN	4NN	5NN	6NN	7NN
Elapsed Time	0.55	0.59	0.57	0.61	0.56
Records Accessed	0.70	0.63	0.67	0.70	0.71
Records Used	0.98	0.98	0.98	0.98	0.98
Disk I/O	-0.06	-0.01	-0.003	-0.01	-0.01
Message Count	0.35	0.34	0.34	0.31	0.31
Message Bytes	0.94	0.94	0.94	0.94	0.94

TABLE II

COMPARISON OF PREDICTIVE RISK VALUES PRODUCED WHEN VARYING THE NUMBER OF NEIGHBORS. NEGLIGIBLE DIFFERENCE IS NOTED BETWEEN THE VARIOUS CHOICES.

2) How many neighbors?: Table II shows the result of varying the number of neighbors to use to predict query performance. $k = 3$ seems to work as well as $k = 4$ or 5. The difference between the predictive risk values is negligible for most of the metrics. Disk I/Os were 0 for most of the queries (as shown in Table 2, we had thousands of small queries whose data fit in memory). Thus, the predictive risk value appears to be very poor, perhaps because the number of disk I/Os is a function of the amount of memory and whether the tables can reside in-memory (which is more of a configuration issue than a query feature). $k = 3$ performs better for predicting

records accessed relative to $k = 4$ and $k = 4$ performs better for predicting elapsed time compared to $k = 3$. We therefore chose $k = 3$ with the intuition that for queries with few close neighbors, a smaller value of k would be better.

Metric	Equal	3:2:1 Ratio	Distance Ratio
Elapsed Time	0.55	0.53	0.49
Records Accessed	0.70	0.83	0.35
Records Used	0.98	0.98	0.98
Disk I/O	-0.06	-0.09	-0.04
Message Count	0.35	0.37	0.35
Message Bytes	0.94	0.94	0.94

TABLE III

COMPARISON OF PREDICTIVE RISK VALUES PRODUCED WHEN USING VARIOUS RELATIVE WEIGHTS FOR NEIGHBORS

3) How do we map from the neighbors to performance metrics?: Another consideration in our methodology was how to map from the set of k neighbors to a vector of predicted performance metrics. Ideally, we would combine the neighbors' points in the performance projection to get a new point and then use a reverse mapping function to derive a performance feature vector. However, finding a reverse mapping is an open problem (called the *pre-image* problem). Instead, we decided to combine the performance feature vectors of the neighbors. We tried different ways to weight the neighbors' values: equal weight for all three neighbors, a 3:2:1 ratio for the weight of the three neighbors in order of nearness, and weight proportional to the magnitude of distance from the test query feature vector. Results for these functions are summarized in Table III. None of the weighting functions yielded better predictions consistently for all of the metrics. We therefore choose the simplest function: weight all neighbors equally. Our prediction for each performance metric is the average of the three neighbors' metrics.

VII. PREDICTION RESULTS

After finding a good adaptation of KCCA for query prediction, we evaluated its performance prediction accuracy for multiple query sets and system configurations. We first present results from training our model on the four node HP Neoview system using queries to TPC-DS tables and predict performance for a different set of queries to TPC-DS as well as queries to a customer database. We then validated our approach by training a model for and predicting performance of queries on various configurations of a 32 node Neoview system. Lastly, we discuss how our predictions compare to the query optimizer's cost estimates.

A. Performance prediction using KCCA on a 4-node system

1) Experiment 1: Train model with realistic mix of query types: The first experiment shows our results from using 1027 queries for our training set, including 30 bowling balls, 230 golf balls and the 767 feathers. The test set includes 61 queries (45 feathers, 7 golf balls and 9 bowling balls) that were not included in our training set.
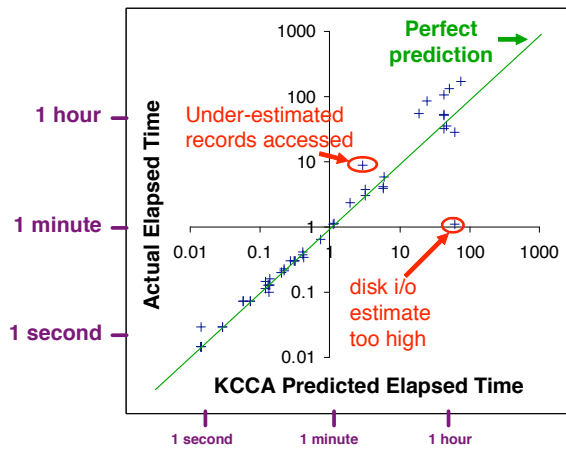
Fig. 10. Experiment 1: KCCA-predicted vs. actual elapsed times for 61 test queries. We use a log-log scale to accommodate the wide range of query execution times from milliseconds to hours. The predictive risk value for our prediction was 0.55 due to the presence of a few outliers (as marked in the graph). Removing the furthest outlier increased the predictive risk value to 0.61.
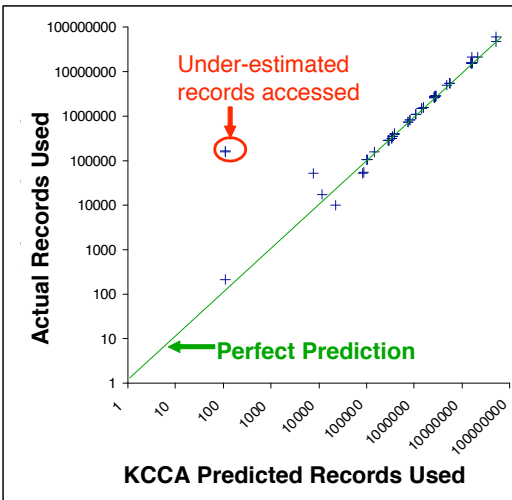

Fig. 11. Experiment 1: KCCA-predicted vs. actual records used. The predictive risk value for our prediction was 0.98. (Predictive risk value close to 1 implies near-perfect prediction).
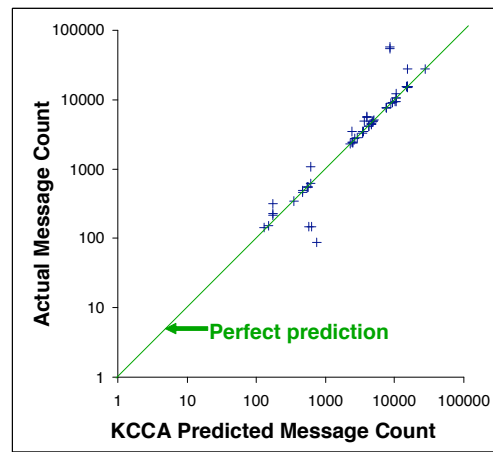

Fig. 12. Experiment 1: KCCA-predicted vs. actual message count. The predictive risk value for our prediction was 0.35 due to visible outliers.
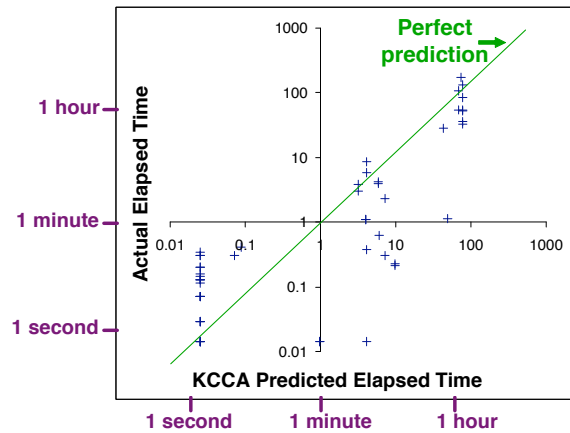

Fig. 13. Experiment 2: KCCA-predicted vs. actual elapsed times of 61 test queries, predicted using a training set of only 30 queries of each type.

Figure 10 compares our predictions to actual elapsed times for a range of queries. As illustrated by the closeness of nearly all of the points to the diagonal line (perfect prediction), our predictions were quite accurate. Our original test set had only three bowling balls. When we decided to add more, the operating system of our Neoview system had already been upgraded. Not surprisingly, the accuracy of our predictions for the six bowling balls we then ran and added was not as good. When the two circled outliers and the more recent bowling balls were eliminated, then the predictive risk value jumped to 0.95.

We show similar graphs for records used in Figure 11 and message counts in Figure 12 (other metrics are omitted for space reasons). The simultaneous predictability of multiple performance metrics using our approach enabled us to better understand inaccurate predictions. For example, for one prediction where elapsed time was much too high, we had greatly overpredicted the disk IOs. This error is likely due to our parallel database's methods of cardinality estimation. When we underpredicted elapsed time by a factor of two, it was due to under-predicting the number of records accessed by a factor of three.

2) Experiment 2: Train model with 30 queries of each type: For our second experiment, to balance the training set with equal numbers of feathers, golf balls and bowling balls, we randomly sampled 30 golf balls and 30 feathers to include in the training set and predicted performance of the same set of 61 test set queries as in Experiment 1. Figure 13 compares the predicted and actual elapsed times for this experiments. We see that our predictions were not as accurate as in our previous experiment (which included a larger number, 1027, of queries in the training set). As is the case with most machine learning
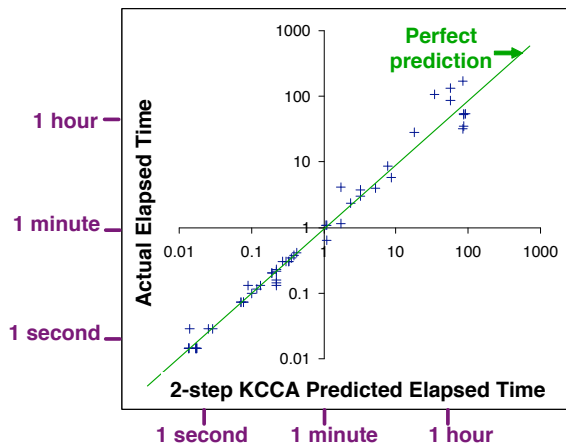
Fig. 14. Experiment 3: Two-step KCCA-predicted elapsed time vs. actual elapsed times for the 61 queries in our test set. The first step classifies the query as a feather, golf ball or bowling ball; the second step uses a query type-specific model for prediction. The predictive risk value was 0.82.

Fig. 15. Experiment 4: A comparison of one-model KCCA-prediction to two-step KCCA-prediction for a test set of customer queries.

Metric	4 nodes	8 nodes	16 nodes	32 nodes
Elapsed Time	0.89	0.94	0.83	0.68
Records Accessed	1.00	1.00	1.00	0.57
Records Used	0.99	1.00	1.00	0.99
Disk I/O	0.72	-0.23	Null	Null
Message Count	0.99	0.99	0.99	0.94
Message Bytes	0.99	0.99	1.00	0.97

Fig. 16. Comparison of predictive risk values produced for each metric on various configurations of the 32 node system (we show the number of nodes used for each configuration). Null values for Disk I/O reflect 0 disk I/Os required by the queries due to the large amount of memory available on the larger configuration.

algorithms, more data in the training set is always better.

3) Experiment 3: Two-step prediction with query type-specific models: So far, we have described a single model for predicting query performance. We also tried a two-step approach using multiple models based on the same feature vectors. In the first step, we use the neighbors from the first model to predict simply whether the query is a "feather," "golf ball," or "bowling ball." (E.g., if a test query's neighbors are two feathers and a golf ball, the query is classified as a feather). In the second step, we then predict its performance using a model that was trained *only* on "feather" (or "golf ball" or "bowling ball") queries. Since we had many fewer examples of golf ball and bowling ball queries in our training sets, we thought this approach might do better for predicting their performance.

Figure 14 shows our results for predicting elapsed time of queries using this two-step approach. Our predictions were more accurate than in Experiment 1, as evidenced by the fewer number of outliers with respect to the perfect prediction line. In comparing Figure 10 and Figure 14, we notice a few instances where the one model prediction was more accurate than the two-step prediction. For the most part, this result was due to the fact that the test query was too close to the temporal threshold that defines feathers and golf balls and forcing it into one category made the prediction marginally worse.

4) Experiment 4: Training and testing on queries to different data tables: We also evaluated how well we predict performance when the training set queries and test set queries use different schemas and databases. Figure 15 shows our one-model and two-step KCCA predictions for a set of 45 queries to a customer's database where the training was done on queries to the TPCDS tables. When compared to actual elapsed time, the two-step KCCA prediction method was relatively more accurate. Most of the one-model KCCA-predictions were one to three orders of magnitude longer than the actual elapsed
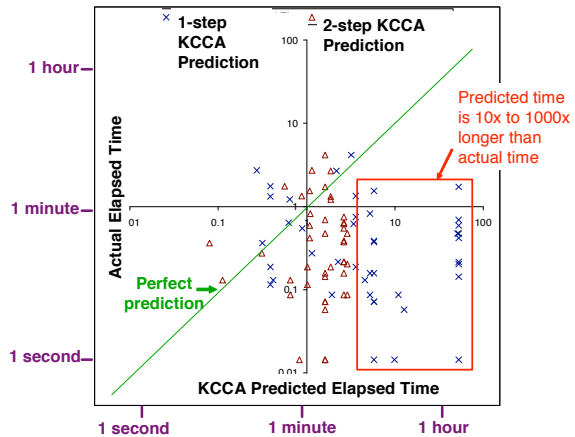
times. One caveat of this experiment is that the customer queries we had access to were all extremely short-running (mini-feathers). As a result, even when our predicted elapsed times were within seconds of the actual elapsed times, the prediction error metric is very high relative to the size of the query. In the near future, we are likely to gain access to longer-running queries for the same customer as well as a variety of queries from other customers.

B. Prediction on a 32-node system

We validated our prediction technique by training and testing on more and larger system configurations. We had limited access to a 32 node parallel database system, which we could configure to process queries using only subsets of nodes.

We reran the TPC-DS scale factor 1 queries on the 32 node system and used 917 of these queries for training the model and 183 queries for testing the model. Figure 16 shows the predictive risk values for each metric predicted for each of four configurations of the 32 node system. The configuration was varied by using only a subset of the nodes on the system, thus reducing the number of CPUs and consequent memory available for use. The training set of queries was rerun on each configuration.

Our results are very encouraging and demonstrate that we

can effectively predict the performance of TPC-DS queries regardless of system configuration. We made several observations in the course of re-running our queries on various configurations of the 32 node system.

First, while we had characterized our queries as feathers, golf balls and bowling balls on the 4 node system, when the same queries were run on the 32 node system, they all ran very quickly. The queries were short-running regardless of the number of nodes used on the 32 node system. We would have liked to run queries that took longer than 10 minutes on the 32 node system, but were unable due to restrictions on our access to this production machine.

Another noteworthy point is that the scenario where only 4 of the 32 nodes were used showed reasonable prediction accuracy of Disk I/Os whereas all other configurations of the 32 node system did not seem to include Disk I/Os in the model. We note that the likely cause for this special case is that the number of disk I/Os for each query was significantly higher for the 4 node configuration where the amount of available memory was not enough to hold many of the TPC-DS tables in memory. Since increasing the number of nodes used also proportionally increases the amount of memory, the 8, 16 and 32 node configurations had very few queries that required disk I/Os.

Lastly, the query execution plans for the 4 node system were different from the same queries' plans on the 32 node system. Also, plans for various configurations of the 32 node system differed from one another. This is because although we limited the number of CPUs used to process queries, we did not change the physical layout of the data on disk, so more than 4 nodes were involved in the data access operations. All plans produced on the 32 node system access and combine results from more disk drives than those on the 4 node system. Plans that limit the number of nodes used for processing on the 32 node system have fewer resources (especially memory) than those that run with all 32 nodes.

C. Discussion

1) Can we compare our predictions to query optimizer cost estimates?: Although we are often asked how our predictions compare to query optimizer's cost estimates, such a comparison is unfair, as described in Section III. For completeness, Figure 17 compares the 4 node system's query optimizer cost estimates to the actual elapsed times for running the 61 test queries. Since the optimizer cost units are not time units, we cannot draw a "perfect prediction" line. We instead draw a line of "best fit."

We generated similar graphs for other optimizer metrics, such as records used, which were consistent with the one shown: the optimizer's estimates do not correspond to actual resource usage for many queries, especially ones with elapsed times of over a minute. When compared to our results in Figure 14, it is apparent that our model's predictions are more accurate than the optimizer's query cost estimation.

However, our predictions are created in completely different circumstances than the query optimizer's. First, we have the
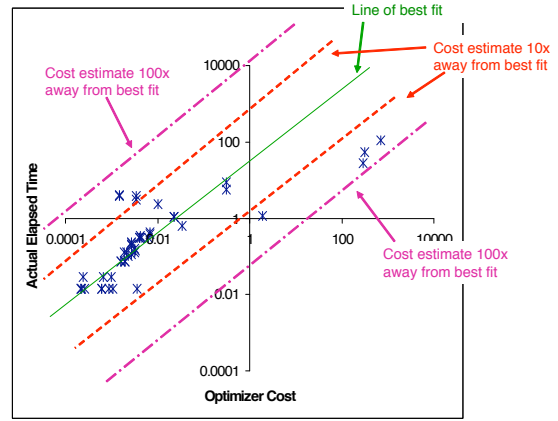

Fig. 17. Optimizer-predicted costs vs. actual elapsed times for the set of 61 test queries from Experiment 1. The optimizer costs and measured times came from Neoview's commercial query optimizer.

luxury of being able to train to a *specific configuration*, whereas the optimizer's cost models must produce estimates for all possible configurations. Second, we can spend orders of magnitude more time calculating our estimates than the optimizer. Third, we use historical information about the actual performance metrics for various plans. The optimizer typically does not use this information. In fact, our performance estimates are actually based upon the optimizer's cost estimates, since we use the optimizer's cost estimates plus historical performance data in our feature vectors.

2) Can our results inform database development?: We are very interested in using our prediction techniques to investigate open questions that might be of interest to a database engine development team. For example, we would like to identify which query operators, and in what conditions, have the greatest impact on query performance. However, in our model, when the raw query plan data is projected by KCCA, the dimensions of the projection do not necessarily correspond to features of the raw data; it is computationally difficult to reverse KCCA's projection to determine which features are considered for each dimension. As an alternate technique to estimating the role of each feature, we compared the similarity of each feature of a test query with the corresponding features of its nearest neighbors. At a cursory glance, it appears that the counts and cardinalities of the join operators (e.g., nested loops join) contribute the most to our performance prediction model.

3) Can we predict anomalous queries?: We currently suffer from the limitation that our predictions are limited to the range of performance metrics reflected by our training set. We are only just beginning to explore an approach to capturing and quantifying the certainty of our predictions. Initial results indicate that we can use Euclidian distance from the three neighbors as a measure of confidence – and that we can thus identify queries whose performance predictions may be less accurate. This approach could potentially identify anomalous queries. For example, the anomalous bowling balls in Fig-

ure 10 were not as close to their neighbors as the better-predicted ones.

4) How fast is KCCA?: Given a KCCA model, prediction of a single query can be done in under a second, which makes it practical for queries that take many minutes or hours to run, but not for all queries. Prediction can also occur simultaneously with query execution if its purpose is to identify the long-running queries. For a new customer, using KCCA to predict the time of a workload on one or more potential configurations is much faster than creating a database or running a set of queries.

Training, on the other hand, takes minutes to hours because each training set datapoint is compared to every other training set datapoint, and computing the dimensions of correlation takes cubic time with respect to the number of datapoints. We expect training to be done only periodically for each system configuration (and possibly only at the vendor). We also plan to investigate techniques to make KCCA more amenable to continuous retraining (e.g., to reflect recently executed queries). Such an enhancement would allow us to maintain a sliding training set of data with a larger emphasis on more recently executed queries.

VIII. Conclusions

We predict multiple performance metrics of business intelligence queries using a statistical machine learning approach. We are able to predict multiple resource usage characteristics, using only information available before the queries execute, and to predict accurately for both short and long-running queries. We believe our methodology can greatly improve the effectiveness of critical tasks that rely on accurate predictions, including system sizing, capacity planning and workload management.

Given operator cardinalities and performance metrics for queries in its input feature vectors, the KCCA algorithm we use interpolates on the relative differences between these cardinalities and metrics to build its model. This powerful technique allows us to make very accurate performance predictions for specific query plans. The predictions can be used to custom-calibrate optimizer cost estimates for a customer site at runtime and also give us a quantitative comparison of different query plans for the same query.

Our long-term vision is to use domain-specific prediction models, like the one we built for database queries, to answer what-if questions about workload performance on a variety of complex systems. Only the feature vectors need to be customized for each system. We are currently adapting our methodology to predict the performance of map-reduce jobs in various hardware and software environments. We also plan to use this methodology to predict application behavior in parallel computing environments.

References

[1] S. Chaudhuri, R. Kaushik, and R. Ramamurthy, "When Can We Trust Progress Estimators For SQL Queries?" in *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM Press, 2005, pp. 575–586.

[2] M. F. Arlitt, "Characterizing web user sessions." *SIGMETRICS Performance Evaluation Review*, vol. 28, no. 2, pp. 50–63, 2000.

[3] R. M. Yoo, H. Lee, K. Chow, and H.-H. S. Lee, "Constructing a non-linear model with neural networks for workload characterization," in *IISWC*, 2006, pp. 150–159.

[4] J.Moore, J.Chase, K.Farkas, and P.Ranganathan, "Data center workload monitoring, analysis, and emulation." 2005.

[5] S. Elnaffar, P. Martin, and R. Horman, "Automatically classifying database workloads," 2002. [Online]. Available: citeseer.ist.psu.edu/elnaffar02automatically.html

[6] P. S. Yu, M.-S. Chen, H.-U. Heiss, and S. Lee, "On workload characterization of relational database environments," *Software Engineering*, vol. 18, no. 4, pp. 347–355, 1992. [Online]. Available: citeseer.ist.psu.edu/yu92workload.html

[7] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh, "An analysis of database workload performance on simultaneous multithreaded processors," in *ISCA*, 1998, pp. 39–50.

[8] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker, "Performance characterization of a quad pentium pro smp using oltp workloads," in *ISCA*, 1998, pp. 15–26.

[9] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, "Estimating Progress Of Execution For SQL Queries," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM Press, 2004, pp. 803–814.

[10] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke, "Toward a Progress Indicator For Database Queries," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM Press, 2004, pp. 791–802.

[11] ——, "Increasing The Accuracy and Coverage of SQL Progress Indicators," in *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 853–864.

[12] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang, "Statistical learning techniques for costing xml queries," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 289–300.

[13] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil, "Leo - db2's learning optimizer," in *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 19–28.

[14] V. Markl and G. Lohman, "Learning table access cardinalities with leo," in *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2002, pp. 613–613.

[15] G. Luo, J. F. Naughton, and P. S. Yu, "Multi-query SQL Progress Indicators," in *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology*, 2006, pp. 921–941.

[16] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere, "How input data sets change program behaviour." [Online]. Available: citeseer.ist.psu.edu/529074.html

[17] C. Gupta and A. Mehta, "PQR: Predicting Query Execution Times for Autonomous Workload Management," in *Proc. Intl Conf on Autonomic Computing*, 2008.

[18] R. Othayoth and M. Poess, "The making of tpc-ds," in *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 1049–1058.

[19] J. Macqueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, ser. Berkeley: University of California Press, L. M. L. Cam and J. Neyman, Eds., vol. 1, 1967, pp. 281–297.

[20] H. Hotelling, "Analysis of a complex of statistical variables into principal components," *Journal of Educational Psychology*, vol. 24, pp. 417–441; 498–520, 1933.

[21] ——, "Relations between two sets of variates," *Biometrika*, vol. 28, pp. 321–377, 1936.

[22] F. R. Bach and M. I. Jordan, "Kernel independent component analysis," *Journal of Machine Learning Research*, vol. 3, pp. 1–48, 2003.

[23] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. New York, NY, USA: Cambridge University Press, 2004.