The Case for Predictive Database Systems: Opportunities and Challenges

Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, Stan Zdonik
Department of Computer Science
Brown University, Providence, RI, USA
{makdere, ugur, matteo, eli, sbz}@cs.brown.edu

ABSTRACT

This paper argues that next generation database management systems should incorporate a predictive model management component to effectively support both inward-facing applications, such as self management, and user-facing applications such as data-driven predictive analytics. We draw an analogy between model management and data management functionality and discuss how model management can leverage profiling, physical design and query optimization techniques, as well as the pertinent challenges. We then describe the early design and architecture of **Longview**, a predictive DBMS prototype that we are building at Brown, along with a case study of how models can be used to predict query execution performance.

1. INTRODUCTION

Predictive modeling has been used with varying degrees of success for many years [GH05]. As models grow more sophisticated, and data collection and storage become increasingly more extensive and accurate, the quality of predictions improves. As such, model-based, data-driven prediction is fast emerging as an essential ingredient of both user-facing applications, such as predictive analytics, and system-facing applications such as autonomic computing and self management.

At present, predictive applications are not well supported by database systems, despite their growing prevalence and importance. Most prediction functionality is provided outside the database system by specialized prediction software, which uses the DBMS primarily as a backend data server. Some commercial database systems (e.g., the data mining tools for Oracle [Ora], SQL Server [SS08], and DB2 [DB2]) provide basic extensions that facilitate the execution of predictive models on database tables in a manner similar to stored procedures. As we discuss below, and also noted by others (e.g., [DB07, AM06]), this loose coupling misses significant opportunities for improved performance and usability. There has also been recent work on custom integration of specific models (e.g., [JXW08, HR07, ACU10, AU07, APC08]).

This paper argues that next generation database systems should natively support and manage predictive models, tightly integrating them in the process of data management and query processing. We make the case that such a **Predictive Database Management System (PDBMS)** is the natural progression beyond the current afterthought or specialized approaches. We outline the potential performance and usability advantages that PDBMSs offer, along with the research challenges that need to be tackled when realizing them.

A PDBMS enables **declarative predictive queries**—by providing predictive capability in the context of a declarative language like SQL; users will not need to concern themselves with the details of tasks like model training and selection. Such tasks will be performed by the optimizer behind the scenes, optionally using hints from the user. Much as SQL has made programmers more productive in the context of data processing, this approach will have a similar effect for predictive analytics tasks. While there will no doubt be some predictive applications that can benefit from custom, manually optimized prediction logic, we expect that many users will be satisfied with “commodity” predictive functionality. The success of the recent Google Prediction API [GP] is early evidence in this direction. This service allows users to upload their historical data to the service, which automatically and transparently performs model training and selection to produce predicted results.

Predictive queries have a broad range of uses. First, they can support predictive analytics to answer complex questions involving missing or future values, correlations, and trends, which can be used to identify opportunities or threats (e.g., forecasting stock-price trends, identifying promising sponsor candidates, predicting future sales, monitoring intrusions and performance anomalies).

Second, predictive functionality can help build introspective services that assist in various data and resource management and optimization tasks. Today, many systems either use very simple, mostly static predictive techniques or do not use any prediction at all. This is primarily due to the difficulty of acquiring the appropriate statistics and efficiently and confidently predicting over them. For example, pre-fetching algorithms are often based on simple linear correlations to decide future data or query requests. Most admission control schemes are based on static estimations (thresholds) of the maximum number of tasks that the system can cope with. Load distribution algorithms typically detect-and-react instead of predict-and prevent problems. Query optimizers commonly use simplistic analytical models to reason about query costs. There is major recent interest and success in applying sophisticated statistical and learning models to such problems [GKD09, BBD09, SBC06]. An integrated, readily available predictive functionality would make it easy to not only consolidate and replace existing solutions but also build new ones. As such, an integrated predictive functionality would be an

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2011.

5th *Biennial Conference on Innovative Data Systems Research (CIDR ‘11)*
January 9-12, 2011, Asilomar, California, USA.

important step towards building the truly autonomic database systems of the future.

A PDBMS integrates predictive models as first-class entities, managing them in much the same way as data. Thus, we consider **model management** as the key underlying component of a PDBMS. Model management may greatly benefit from analogues of many well-established data management techniques:

- **Profiling and modeling:** Cost and accuracy characteristics of models need to be modeled, and fed to the optimizer so that the proper model(s) can be chosen for a given predictive task.
- **Physical design and specialized data structures:** Data can be structured to facilitate efficient model building and predictive query execution (e.g., I/O-aware skip-lists [GZ08]).
- **Pre-computation and materialization:** Model building is often prohibitively expensive for ad hoc or interactive queries. In such cases, models can be pre-built and materialized for use by the optimizer and executor. Furthermore, this process can be automated in many cases.
- **Query optimization:** The optimizer considers the alternative ways of model building, selection, and execution, as well as the inherent cost-accuracy tradeoffs when selecting an execution plan.

In the rest of the paper, we discuss these model management techniques as well as the technical challenges that arise when building a PDBMS. Our discussion is centered on **Longview,** a prototype predictive DBMS that we have been building at Brown University. Longview is being designed to efficiently support declarative predictive analytics through novel integrated model management techniques. Users can plug new model types into the system along with a modest amount of meta-data, and the system uses these models to efficiently evaluate queries involving predictions.

We sketch the basic architecture of Longview and its early implementation on top of PostgreSQL. We also discuss an internal predictive application, query performance prediction, which exercises some of the model management issues we raise. Finally, we discuss prior work and finish with concluding remarks.

2. BACKGROUND: PREDICTION WITH MODELS

We use the term model to refer to any predictive function such as Multiple Regression, Bayesian Nets, and Support Vector Machines. Training a model involves using one or more data sets to determine the best model instance that explains the data. For example, fitting a function to a time series may yield a specific polynomial instance that can be used to predict future values.

In general, **model training** (or **building**) involves selecting (i) the feature attributes, a subset of all attributes in the data set, and (ii) a training data set. In some cases, a domain expert can manually specify the feature attributes. In other cases, this step is trivial as the prediction attribute(s) directly determine the feature attribute(s), e.g., as in the case of auto-regressive models. Alternatively, feature attributes can be learned automatically. Most solutions for automatic learning are based on heuristics, since given a set of n attributes, trying the power set is prohibitively expensive if n is not small or training is costly [GH05, MWH98]. A common approach is to rank the candidate attributes (often based on their correlation to the prediction attribute using metrics such as information gain or correlation coefficients [CT06]) and use this ranking to guide a heuristic search [GH05] to identify the most predictive attributes tested over a disjoint test data set. The training data set may be sampled to speed up the process.

Prediction **accuracy** is a function of the quality of the estimated models. The quality of the model (and the resulting predictions) can be measured by metrics such as the variation distance [MU05] or the mean square error between the predictions and the true values. With assumptions about the underlying stochastic process, one may be able to bound these measures analytically, using large deviation theory, appropriate versions of the central limit theorem and martingale convergence bounds [MU05]. Alternatively, one can use multiple tests on available data to compute the empirical values for these measures. However, using empirical values to estimate the model or prediction error adds another layer of error to the estimate, namely the gap between the empirical statistics and the true value they estimate. While the empirical statistic is an unbiased estimate, the variance of the estimate can be large, depending on the size and variance of the test set.

Hypothesis testing and confidence interval estimations are two common techniques for determining predictive accuracy [MWH98]. In general, it is not possible to estimate a priori what model would be most predictive for a given data set without training and testing it. One form of hypothesis testing that is commonly used is K-Fold Cross Validation (K-CV). K-CV divides up the training data into k non-overlapping partitions. One of the partitions is used as validation data while the other k-1 partitions are used to train the model.

3. LONGVIEW: A PREDICTIVE DBMS

3.1 Design and Architecture Overview

3.1.1 Data and Query Model

Longview provides two interfaces for access to its predictive functionality. The first access method is the **direct interface**, which consists of a collection of SQL functions that offers direct access to the functionality of the integrated prediction models. The direct interface does not provide the user with automated model management tools and is thus targeted towards advanced users who want to exert hands-on control on the prediction models and their operations. For example, using this interface a user can ask the system to build a linear regression model with specific configuration parameters or perform prediction with a pre-built support vector machine instance. We summarize the details of the direct interface in Section 3.2.1.

The second access method is the **declarative interface**, which offers additional, high-level predictive functionality on top of the low-level direct interface.

This declarative interface extends SQL in a few simple ways to accommodate the extra specifications needed for expressing predictive queries. In particular, queries may refer to **predictors** and **predictor relations** (**p-relations**) to access predictive functionality. Predictors are essentially SQL functions that provide declarative predictive functionality using system-managed prediction models. P-relations are essentially views produced by the application of predictors on select subsets of input features. Both p-relations and predictors can be used in conjunction with regular relations within standard SQL queries. A

p-relation is virtual by default; however, it can also be materialized to enable further optimizations.

We give a simple example that illustrates some of the key concepts of the query language that we are developing. Consider the following schema:

```
Customer(cid, name, city),
Orders(oid, cid, total),
TrainData(cid, status)
```

In addition to the Customer and Orders relations, which store the records for customers and their orders, we define the TrainData relation that stores the status (either "preferred" or "regular") of a subset of the customers. We first show how to build a predictor for predicting the status of any customer based on the training data supplied for a subset of the customers. Next, we discuss p-relations and their use through an example p-relation representing the status predictions of a select subset of customers based on a predictor.

The first step in creating a predictor is to define a schema describing the set of involved features and target attributes. For this purpose, we define a schema, named StatusSchema, with the target attribute customer status and features name, city and total using the CREATE P_SCHEMA statement:

```
CREATE P_SCHEMA StatusSchema (
        name text,
        city text,
        total int,
        TARGET status text)
```

To create a predictor, we use the CREATE PREDICTOR statement that can be used to automatically build prediction model(s) using the given training data set:

```
CREATE PREDICTOR StatusPredictor
ON StatusSchema(name, city, total, status)
WITH DATA
        SELECT name, city, sum(total) as total, status
        FROM Customer C, Orders O, TrainData T
        WHERE T.cid = C.cid and T.cid = O.cid
        GROUPBY cid, name, city, status
WITH ERROR CVERROR(10, "relative_error", 0.1)
```

With the statement shown above, we instruct the system to create a predictor named StatusPredictor by training a set of prediction models using the training data specified through a query. The last part, WITH ERROR, defines the error estimation process. In this example, we want to use 10-fold cross-validation and the relative_error accuracy metric with a target average error of 0.1. Notice that the decoupling between the schema and predictor definitions allows us to create multiple predictors with different data sets or accuracy requirements over a single schema.

The example query below illustrates the use of the StatusPredictor for estimating the status of all customers:

```
SELECT C.cid, StatusPredictor(C.name, C.city, O.total)
FROM Customer C,
      (select cid, sum(total) as total from Orders
       Group By Cid) as O
WHERE C.cid = O.cid
```

The output schema of a predictor is defined by the associated p_schema. In addition, one can add special ERROR attributes to a p_schema to access the estimated errors for each predicted value. For instance, adding the attribute "ERROR relerr real" to p_schema would extend the output schema of a predictor with the *relerr* attribute, which represents the estimated prediction error.

Now, we describe how to define p-relations with the following example:

```
CREATE VIEW StatusPRelation AS
SELECT cid, StatusPredictor(name,city,total)
FROM (
    SELECT cid, name, city, sum(total) as total
    FROM Customer C, Orders O
    WHERE C.cid = O.cid
    GROUP BY cid, name, city, status
    HAVING sum(total) > 1000)
```

With the above statement, we create a p-relation named StatusPRelation, which is basically a view consisting of status predictions from StatusPredictor for the set of features specified with the provided query (i.e., customers with order totals greater than 1000) and the features themselves.

When a view definition that accesses a predictor function is submitted to the system, Longview registers the given data set as a specific target feature set for that predictor. In turn, the model generation process for the predictor works to generate more efficient and accurate prediction models based on the properties of the given feature set.

The use of declarative queries for the specification of data sets in model building and prediction offers an easy and flexible method of expressing predictive operations over complex data sets. Users can easily specify complex queries (e.g., computing aggregates over groups) to supply input data sets for prediction models. Moreover, it is also possible to use database views as data providers. For instance, a database view can be used to perform standard pre-processing tasks such as cleaning, normalization, and discretization [DB07], and can cook the raw data into a form that is more amenable for effective learning.

3.1.2 Basic Architecture

We illustrate the high-level architecture of Longview in Figure 1, which shows the primary functional units of interest, along with the data they require. The architecture reflects the notion of models as first-class citizens by depicting the data manager and model manager as co-equal modules.

The Data Manager is very similar to a typical data manager in a conventional DBMS. The Model Manager is responsible for creating materialized models a priori (materialized) or in an on-demand fashion when an adequate materialized model does not exist. The role of materialized models in the model world is similar to that of indices and materialized views in the data world: they are derived products that can be used to quickly generate data of interest. Indices and materialized views improve query speed while materialized models improve prediction speed.

The Model Manager trains appropriate models (based on the available model templates) for each predictor in the database. The Model Manager can run as a background process, constantly instantiating models for improved accuracy and efficiency. In order to build and maintain prediction models, the Model

Manager can utilize many different strategies. For example, it can choose to sample data at different amounts and times and it can build different types of prediction models over different subsets of available features. In addition, the Model Manager also determines the best model for the query at hand: if an appropriate model has already been pre-computed and materialized, it will identify and use that model; if not, it will create a new instantiation on the fly.

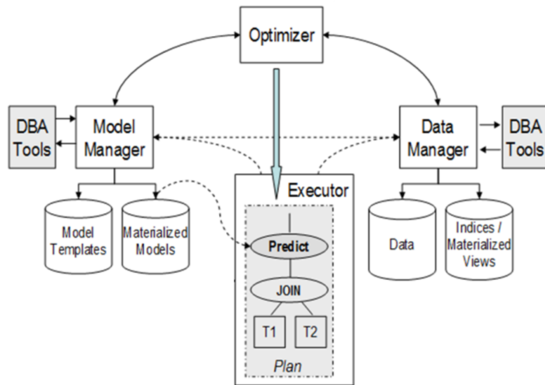

Figure 1: High-level Longview architecture. The system provides full-fledged support for models; model and data management are tightly integrated.

The Model Manager and the Data Manager must cooperate in their decision making. As we discuss later, special data structures can assist the process of model training. This is consistent with the fact that DBMSs, in general, get much of their performance gains from supporting specialized data structures like indices.

Model meta-data entered through the model interface as well as those derived during run-time such as the list of materialized models and their parameters, training results, error values for various data sets, are all stored in a model catalog. The model manager is responsible for updating the catalog.

Longview will try to produce a good model whenever possible by trying various parameter assignments (e.g., history length, sampling density, etc.) and using hypothesis testing to find the best fit. While Longview aggressively tries to optimize this model search process, in some cases, this is either not possible or would require testing too many alternatives. In these cases, Longview will provide a set of tools with which the DBA can inspect the data and add additional information about the datasets which might indicate, for example, that the data is seasonal, or that the data might best be modeled using exponential smoothing. Traditional DBMSs provide such tuning tools for DBAs as well.

P-relation queries are written against views that include predicted attributes. When a p-relation query is received by the system, the optimizer might generate a query plan that contains prediction operators. These operators are selected from a collection of instantiated models that are managed by the Model Manager, or created on the fly. Alternatively, tuples in the predicted view can be computed eagerly and materialized as resources become available, in which case p-relation queries can be executed as scans over the materialized tuples.

3.2 Model Management

As a design philosophy towards a generic model manager, we strive to build on existing database extension mechanisms such as views, triggers, rules and user defined functions to simplify our implementation and produce highly portable functionality.

3.2.1 Database Integration of Prediction Models

Prediction Model API. Longview currently supports a black-box-style integration approach that allows existing model implementations (available from a plethora of standalone applications and libraries such as libsvm [CC01]) to be used by the system as database functions. This approach offers an easy and effective way of utilizing pre-tested and optimized prediction logic within a SQL execution framework. New prediction models are registered into the system by providing implementations of a simple model interface (the prediction model API) describing function templates for training and application of prediction methods. This interface decouples implementation and predictive functionality, while allowing multiple predictive models to be used for the same task. Table 1 summarizes the basic interface methods.

Function	Arguments	Description
Build	*training data* *model parameters*	feature and target values model-specific training parameters
Predict	*model pointer* *feature list*	pointer to previously built model feature values for use in prediction
Serialize	*model pointer*	
Deserialize	*byte array*	serialized model

Table 1 - Prediction Model API

The build function is used to train a prediction model based on the given features and target values, as well as model-specific training parameters. The predict function uses a previously built model to predict a target attribute based on the input feature values. Finally, Longview uses the serialize and de-serialize functions to store and retrieve prediction models. Most third-party model libraries include built-in model (de)serialization methods for this purpose.

Prediction Model Direct Interface. The prediction model API is used internally by the Longview system to access the functionality of prediction models and is not visible to the user. However, as mentioned earlier, Longview also provides an interface for direct access to the prediction models by the user. The main functions included in this interface are given in Table 2. These functions have dynamic implementations in Longview, as wrappers around the prediction model API, and provide a unified method of access to all the available prediction model types within SQL statements.

The create function is used to create a prediction model entry in the model catalogs for the given model type and attribute schema. The Longview model catalog stores all model data and associated meta-data. Each model instance built is recorded in a relation that contains a unique (auto-generated) instance id, model type, and a serialization field storing the type-specific representation of a prediction model (e.g., the coefficients of a regression model). We also store model attributes; each is represented with a name, id, a type (e.g., double) and a role (i.e., feature, target).

The build and predict SQL-functions are similar to the corresponding functions in the prediction model API. The build

function trains the prediction model specified by the model id, and stores its serialized representation in the model catalog. The predict function performs prediction with the given model id over the provided feature data set. We also provide a test function that can be used to apply the model on a feature data set and compute its accuracy over the true values of the target attributes. We provide an argument to specify the accuracy function for use (e.g., absolute error, squared error). The outputs of the test and prediction functions are represented as relations and can be used as data sources in other queries.

Function	Arguments	Description
Create	*model schema* *model type*	description of features and target attributes prediction model type
Build	*model id* *training query* *model parameters*	specifies the model instance query computing the feature and target values model-specific training parameters
Predict	*model id* *feature list / query*	feature values for use in prediction
Test	*model id* *training query* *accuracy options*	parameters for the accuracy function

Table 2 - Prediction Model Direct User Interface

3.2.2 Model Building and Maintenance

Model Materialization. Longview builds and materializes model instances much as a conventional DBMS pre-computes indices or materialized views. For each predictor and associated p-relations, there can be multiple materialized prediction models built using different model types and different feature subsets. As a result, model building and maintenance may easily become a bottleneck as the number of pre-built models increases. Therefore, methods for decreasing the cost of building and maintaining models are an essential part of Longview.

The quality of a model is primarily a function of its training data and model-specific configuration parameters. In the limit, we would like to produce one materialized model for each prediction query. This approach will likely be infeasible for two reasons: (1) the time required to build a model per query is larger than some target threshold, e.g., in applications involving interactive queries; and (2) the estimated time required to update these models in the face of newly arriving data is greater than some maintenance threshold.

In many ways, this problem is very similar to the problem of automatic index or materialized view selection. We require (1) a reasonably good cost and accuracy model that can be used to compare the utility of the materialized models, and (2) a way to heuristically prune the large space of possible models.

A good solution to this problem involves covering the underlying "feature space" well such that a prediction with acceptable accuracy can be made for a large set of queries subject to a limit on model maintenance costs. In prior work, we proposed a solution along these lines for time-series-based forecasting using multi-variate regression [GZ08].

In addition to the techniques mentioned earlier such as sampling, feature selection and materialized models, there are further opportunities to reduce the execution costs of these tasks. First, these operations can be done in parallel for multiple models on the same data. In this **multi-model building** process (akin to multi-query optimization), data can be read once and all relevant models can be updated at the same time. Moreover, we can build and update models in an **opportunistic** manner based on memory-resident data.

Auto Design. The auto-design problem is a related problem in which the goal is to choose and build a set of prediction models based on a given workload that contains a set of predictive queries that are most likely to be submitted, i.e., queries that we would like to execute quickly and with good predictive accuracy. For this purpose, the database system would need to identify the most common prediction attributes in the workload and then the set of features that are highly predictive of those attributes.

Specialized Data Structures. There are opportunities for a PDBMS to leverage data representations that are tuned to the process of prediction. In particular, structures that can enhance model training have the most potential to yield major performance improvements with the idea being accessing "just enough" data to build a model of acceptable accuracy.

Data-driven training commonly involves accessing select regions in the underlying feature space, combined with sampling techniques that can be used to further reduce I/O requirements. This process is often iterative: more data is systematically included to check if the resulting model is better. In general, multi-dimensional index structures defined over the feature space can be effectively used here, but care must be taken that index-based sampling does not introduce any biases. Multi-dimensional clustering, when performed in a manner that facilitates efficient sampling, can provide further benefits. As an alternative to the index-based sampling of disk-resident data, we can also opt to replicate the data (or materialize the results of a training query) using disk organizations tuned for efficient sampling, e.g., horizontally partition the data into uniform samples so that sampling can be done with sequential I/O.

As a concrete example for time-series prediction, we introduced a variant of skip lists to efficiently access arbitrary ranges of the underlying time dimension with different sampling granularities.

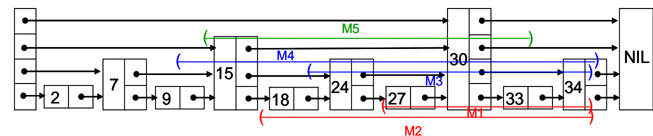

Figure 2: I/O conscious skip-lists. Each node indicates a block of tuples sampled from the original relation. Unlike in standard skip lists, nodes (blocks) are not shared across levels. M's indicate different time ranges and sampling density.

The original skip-list formulation is modified to make it I/O conscious by copying the relevant data from each lower level up to the higher-levels. Each level is essentially a materialized sample view [JJ08] stored in clustered form on disk, allowing us to access a particular time range with desired density with a small number of disk accesses (see Figure 2 for an illustration).

3.2.3 Query Execution and Optimization

Predictor optimization. Declarative predictive queries specify what to predict but not how. For a given prediction task, it is the responsibility of the predictor to build and use an appropriate prediction model satisfying the desired accuracy. For this purpose, each Longview predictor continuously tries to build accurate prediction models for as much of its input feature space as possible, while keeping resource consumption under a configurable threshold to avoid negatively impacting the other database tasks. In the case of p-relations, predictors can build more targeted prediction models using select parts of the training data (i.e., **model segmentation**) based on the target data of p-relations. We discuss an application of the model segmentation idea and demonstrate its potential in Section 4.

In addition, Longview automatically keeps track of model cost-accuracy characteristics. For each model instance, the run-time cost and quality of the predictions during build and test operations are recorded. Using this information, Longview can monitor the evolution of models, track the used training data sets and the performance values on test data sets. These **model profiles** guide query optimization decisions. We may also expect expert users (or model developers) to supply simple cost functions, akin to those for the relational operators, for training and prediction costs, which can also be stored and leveraged as part of model profiles.

Finally, we observed the need for a more formal, expressive tool when working with sophisticated prediction models. To this end, we believe that a **model algebra** that captures common model operations such as choice (selection), composition, and merge is warranted. Properties of these operations could introduce further functionality as well as optimization opportunities. A model ensemble, which uses a set of prediction models collectively to perform a prediction task, is an example for this complex model case. Model ensembles rely on the collective power of multiple prediction models to smooth their predictions and mitigate the potential errors from a single prediction model.

Online execution. Online execution of predictive queries (along the lines of online aggregation), in which predictions, and thus the query results, get progressively better over time, is an important usage model for interactive, exploratory tasks. Predictive accuracy can be improved over time using more data, more features, or more models. The challenge is to effectively orchestrate this process and perform efficient revision of query results.

4. CASE STUDY: PREDICTING QUERY EXECUTION LATENCIES

We now describe our ongoing work on an inward-looking predictive task, **query performance prediction** (QPP), which involves the estimation of the execution latency of query plans on a given hardware platform. Modern database systems can greatly benefit from accurate QPP. For example, resource managers can utilize QPP to allocate workload such that interactive behavior is achieved or specific quality of service targets are met. Optimizers can choose among alternative plans based on expected execution latency instead of total work incurred.

While accurate QPP is important, it is also challenging: database systems are becoming increasingly complex, with several database and operating system components interacting in sophisticated and often unexpected ways. Analytical cost models are not designed to capture these interactions and complexity. As such, while they do a good job of comparing the costs of alternative query plans, they are poor predictors of plan execution [GKD09].

As an alternative, we express the QPP task using the declarative prediction interface in Longview. In addition to describing the query specification and execution, we also show different modeling approaches to achieve accurate QPP under various workload scenarios. If a representative workload is available, for example, we can build good models using coarse-grained, plan-level models. Such models, however, do not generalize well, and perform poorly for unseen or changing workloads. In these cases, fine-grained, operator-level modeling performs much better due to its ability to capture the behavior of arbitrary plans, although they do not perform as well as plan-level models for fixed workloads. We then build hybrid models that combine plan- and operator-level models to provide the best of both worlds by striking a good balance between generality and accuracy.

Plan-level Prediction. We first consider a basic approach that extracts features from query plans and then couples them with sample plan executions to build models using supervised learning (as also explored in [GKD09]). Once built, these models can perform predictions using only static plan information. The following features are extracted from each query plan for modeling purposes: optimizer estimates for query plan costs, number of output tuples and their average sizes (in bytes), and instance (i.e., occurrence) and cardinality counts for each operator type included in the query plan.

We integrated two prediction models, Support Vector Machines and Linear Regression, into the PostgreSQL database system (version 8.4.1) through the use of machine learning libraries LIBSVM [CC01] and Shark [IMT08]. We used the TPC-H decision support benchmark to generate our database and query workload. The database size is set to 10GB and experiments were run on a 2.4 GHz machine with 4GB memory. Our query workload consists of 500 TPC-H queries, which are generated from 18 TPC-H query templates and executed one after another with clean start (i.e., file system and database buffers are cleared).

Fixed Workload Experiment: In the first experiment, we defined a plan-level predictor using the described query plan features and the execution time target attribute as our p_schema (named *PlanSchema*). For this purpose, we first inserted the runtime query plan features and the execution times of all queries in our TPC-H workload to database tables (*runtimefeats* and *qexec*). Then, we defined our predictor to use 90% of the workload for building prediction models to estimate the execution times of the remaining 10% of the queries. We provide the definition of the plan-level predictor below; however at this point we do not have a SQL parser for the extensions proposed in the declarative predictor interface and thus performed our operations using the direct interface along with a few additional SQL functions that provide functionality similar to the declarative predictor interface.

```
CREATE PREDICTOR PlanLvlPredictor
ON PlanSchema(…)
WITH DATA
    SELECT R.*, Q.exec_time
    FROM runtimefeats R, qexec Q
    WHERE R.qid = Q.qid and R.qid <= 450
```

The *qid* attribute is a key in both tables that uniquely defines a query in the TPC-H workload. Next, we used the pre-runtime

estimations of the query plan features from the query optimizer (stored in table *estimatedfeats*) for performance prediction of the remaining 10% of the queries. The following query is used to express this operation:

SELECT qid, PlanLvlPredictor(…).exec_time
FROM estimatedfeats E
WHERE E.qid > 450

In this experiment, we used support vector machines (SVMs) as our prediction model. In addition, our current predictor optimizer uses a standard feature selection algorithm for choosing the set of features to use in prediction models. The set of features (7 of the total 29 features) used in the resulting model are: number of Group Aggregate, Hash Aggregate, and Materialization operators, estimated total plan cost, cardinality of Hash Aggregate and Hash Join operators and the estimated total number of output rows from all operators in the query plan. The error value (defined as |true value – estimate| / true value) for each TPC-H template is shown in Figure 3 (The average prediction accuracy is 90%).

We observed that queries from the 9th template (which has the unusual high errors) run close to the 1 hour time limit (after which we killed and discarded queries) and therefore execute longer than most other queries. We then performed manual model-segmentation by building a separate prediction model for the queries of the 9th template, which achieved 93% accuracy.

This example illustrates the potential efficiency of using segmented models built from different data partitions. As discussed before, intelligent model-building algorithms that automatically identify such partitions in the feature space are essential for improved accuracy.

Finally, when we added the additional feature used by that model (cardinality of the Nested Loop operator) to the general prediction model and retrained it, we increased its accuracy to 93% (shown with the bars for 2-step feature selection in the figure).

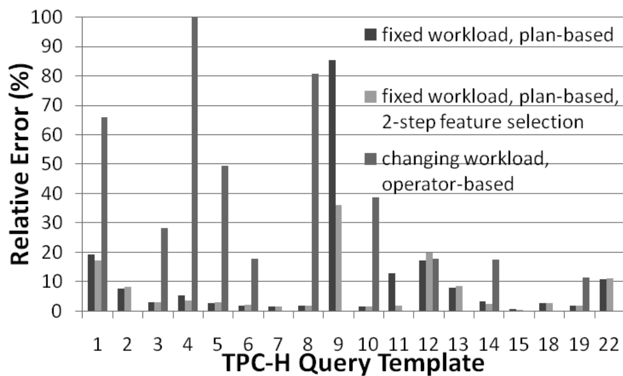

Figure 3: Query Prediction Performance for TPC-H Queries.

Changing Workload Experiment: In this experiment, we built separate prediction models for each TPC-H template using only the queries from the other TPC-H templates for training. In this case the average prediction error increased to 232%. In addition, the error values were highly dependent on the target query template and were distributed in a large range (2% to 1692%).

Operator-level Prediction. We also studied an operator-level modeling approach with the goal of building better models for the Changing Workload scenario. In this case, we build separate Linear Regression models to estimate the execution time for each *operator type* and compose them in a bottom-up manner up the query tree to predict the total execution time.

Each operator is modeled using a generic set of features such as the number of input/output tuples and estimated execution times for child operators (runtime and estimated values for these features are stored in *opruntimefeats* and *opestimatedfeats* tables). Bottom-up prediction requires a nested use of predictors. Moreover, the connections between predictors are dynamic as they depend on the plan of the query at hand. Currently, we perform this nested prediction operation within a user-defined database function that uses the operator predictors as required by the plan of each query. We think that such complex models can be built and used more effectively with a model algebra as mentioned in Section 3.

The results for the Changing Workload experiment using the operator-level prediction methods are shown in Figure 3 for 10 TPC-H templates. The average error rate is 56%, which represents a major improvement over query-level prediction for this workload.

Hybrid Prediction. Looking closer, we observe that the error (of 233%) for the operator-level prediction of template 4 queries is much higher than those for other templates. To gain more insight, we provide the error values for each operator in the execution plan for an example template-4 query (Figure 4). Observe that the errors originate from the highlighted sub-plan and propagate to the upper levels. Here, the error is due to the inability of the models to capture the per-tuple processing time of the Hash Aggregate operator, which in this case is computation-bound. Thus the I/O cost of the Sequence Scan operation that normally determines the overall execution latency is dominated by Hash Aggregate's high computational cost in this case. The fundamental problem is that operator-level training inherently fails to capture the "context" of the operator behavior.

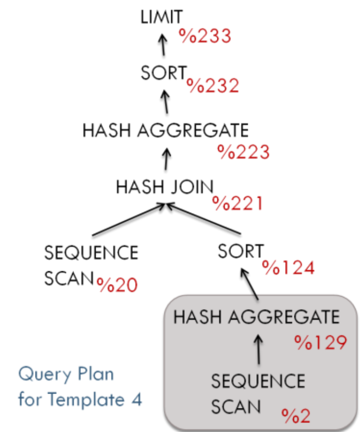

Figure 4: Query Tree and Prediction Errors for Template 4.

To solve this problem, we combined the plan- and operator-level prediction methods for template 4 by modeling the highlighted sub-plan with a plan-level model and using operator-level prediction for the remainder of the query plan. With this approach, we reduced the template error to 53% and the overall average error across templates to 38% for the Changing Workload scenario.

5. RELATED WORK

We draw from a large number of subject areas, which we summarize below. Other closely related work was cited inline as appropriate.

Major commercial DBMSs support predictive modeling tools (e.g., Oracle Data Mining tools, SQL Server Data Mining and DB2 Intelligent Miner). Such tools commonly allow users to invoke model instances, typically implemented as stored procedures, using extended SQL (e.g., the "FORECAST" clause [Ora]). In SQL Server Analysis Services, users are provided with a graphical interface within Visual Studio in which they can interactively build and use a number of prediction models such as decision trees and naïve Bayes networks. While we utilize similar prediction models and techniques, our goal is to create a more automated and integrated system in which predictive functionality is mostly managed by the system with help from the user (akin to existing data management functionality).

On the academic side, MauveDB [AM06] was an early system to support model-based views defined using statistical models. Such views can be used for a variety of purposes including cleaning, interpolation and prediction (with a focus on sensor network applications). The PDBMS functionality we sketch in this paper goes significantly beyond the scope of MauveDB. Our direct prediction interface and MauveDB views have similar functionality and purpose. However, we believe that automated model building and maintenance services, such as our declarative predictor interface, are essential for commoditization of predictive functionality.

Another closely related system is Fa [DB07], which was designed to support forecasting queries over time-series data. Fa offers efficient strategies for model building and selection, making a solid contribution towards model management and predictive query processing. Longview can leverage many of Fa's techniques but also aims for deeper, more comprehensive model management, by treating models as native entities and addressing the entire predictive model life cycle.

Recently, there have been successful applications of machine learning techniques to DBMS self-management problems. Query-plan-level predictions have been studied in [GKD09]. NIMO proposed techniques for accelerating the learning of cost models for scientific workflows [SBC06]. Performance prediction for concurrent query workloads was investigated in [AAB08].

6. CONCLUDING REMARKS

We argue that it is high time for the database community to start building predictive database systems. We discussed how predictive queries could meaningfully leverage and, at the same time, contribute to next generation data management. We presented our vision for a predictive DBMS called Longview, outlined the main architectural and algorithmic challenges in building it, and reported experimental results from an early case study of applying the predictive functionality for query performance prediction.

ACKNOWLEDGEMENTS

This work is supported in part by the NSF under grant IIS-0905553.

7. REFERENCES

[AAB08] Ahmad, M., Aboulnaga, A., Babu, S., and Munagala, K. Modeling and exploiting query interactions in database systems. CIKM 2008.

[ACU10] M. Akdere, U. Cetintemel, E. Upfal: Database-support for Continuous Prediction Queries over Streaming Data. PVLDB 3(1), 2010.

[AM06] A. Deshpande and S. Madden, MauveDB: Supporting Model-based User Views in Database Systems. SIGMOD 2006.

[AU07] Declarative temporal data models for sensor-driven query processing. Y. Ahmad and U. Cetintemel. DMSN 2007.

[APC08] Simultaneous Equation Systems for Query Processing on Continuous-Time Data Streams. Y. Ahmad, O. Papaemmanouil, U. Cetintemel, J. Rogers. ICDE 2008.

[BBD09] S. Babu, N. Borisov, S. Duan, H. Herodotou, and V. Thummala. Automated Experiment-Driven Management of (Database) Systems. HotOS 2009.

[CC01] Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2006.

[DB2] DB2 Intelligent Miner Web Site. http://www01.ibm.com/software/data/iminer/

[DB07] S. Duan and S. Babu, Processing Forecasting Queries. VLDB'07.

[GH05] J. G. De Gooijer and R. J. Hyndman. 25 Years of IIF Time Series Forecasting: A Selective Review. June 2005. Tinbergen Institute Discussion Papers No. TI 05-068/4.

[GKD09] A. Ganapathi, H. Kuno, U. Dayal, J. Wiener, A. Fox, M. Jordan, D. Patterson: Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. ICDE 2009.

[GP] Google Prediction API, http://code.google.com/apis/predict/

[GZ08] Tingjian Ge, Stan Zdonik. A Skip-list Approach for Efficiently Processing Forecasting Queries. VLDB 2008.

[HR07] H. Bravo, R. Ramakrishnan. Optimizing mpf queries: decision support and probabilistic inference. SIGMOD 2007.

[IMT08] Christian Igel, Verena H., and Tobias G. Shark. *Journal of Machine Learning Research*, 2008.

[JJ08] Shantanu Joshi and Chris Jermaine. Materialized Sample Views for Database Approximation. IEEE Trans. Knowl. Data Eng. 20(3): 337-351 (2008)

[JXW08] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, P. Haas: MCDB: a monte carlo approach to managing uncertain data. SIGMOD 2008.

[MU05] Mitzenmacher, M., Upfal, E. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*.

[MWH98] Makridakis, S., Wheelwright S., and Hyndman, R. Forecasting Methods and Applications. Third Edition. John Wiley & Sons, Inc. 1998.

[Ora] Oracle Data Mining Web Site. http://www.oracle.com/technology/products/bi/odm/index.html

[SBC06] P. Shivam, S. Babu, and J. Chase. Active and Accelerated Learning of Cost Models for Optimizing Scientific Applications. VLDB 2006.

[SS08] Microsoft SQL Server 2008. www.microsoft.com/sqlserver/2008/en/us/datamining.aspx