Abstract of "Supporting and Leveraging Prediction Models
for Database Applications", by Mert Akdere, Ph.D., Brown University, May 2012.

Model-based, data-driven prediction is emerging as an essential ingredient for both user-facing applications, such as predictive analytics, and system-facing applications such as autonomic computing. This thesis studies the two complementary research questions of how to effectively support and leverage predictive models for database applications.

We explore the performance and usability advantages of integrated predictive functionality within database systems and argue that next generation database systems should natively support and manage predictive models, tightly integrating them in the process of data management and query processing. This is in contrast to the current practice of implementing such functionality within the application space. We study how various types of predictive models can be efficiently supported by utilizing and extending database system mechanisms. Specifically, we discuss (i) white-box support, in which the knowledge of model semantics facilitates a tight integration, thus introducing rich optimization opportunities and (ii) black-box support, in which no such knowledge is assumed, thus leading to a general but less optimizable system.

We derive our results from two detailed case studies. In the first one, we describe white-box model support of Bayesian Networks to enable continuous predictive queries over streaming data. In the second, we describe black-box model support and its application on query performance prediction in database systems. We describe efficient implementations of both applications in open-source database systems. Our experimental studies provide quantitative evidence that predictive functionality can be achieved within databases with a level of performance and accuracy that is competitive with or favorable to that of specialized, custom solutions.

Supporting and Leveraging Prediction Models for Database Applications

by

Mert Akdere

B. S., Bilkent University, Turkey, 2005.

Sc. M., Brown University, USA, 2008.

A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2012

This dissertation by Mert Akdere is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date _____ _____
Uğur Çetintemel, Director

Recommended to the Graduate Council

Date _____ _____
Eli Upfal, Reader

Date _____ _____
Stanley B. Zdonik, Reader

Approved by the Graduate Council

Date _____ _____
Peter M. Weber
Dean of the Graduate School

Vita

Mert Akdere was born in İstanbul, Turkey in 1983. He completed his undergraduate studies in computer engineering at Bilkent University, Turkey in 2005. Then he started the doctoral program in the Computer Science Department at Brown University, USA in 2005. There he worked with Uğur Çetintemel and received his Sc.M. degree in Computer Science in 2008.

Acknowledgements

First and foremost, I want to thank my advisor, Uğur Çetintemel. I appreciate all his contributions of time and ideas to my research. He has been a model advisor; especially in times of difficulty, he has been consistently supportive and insightful.

I would like to thank my thesis committee members Stan Zdonik and Eli Upfal for their feedback and contributions to my dissertation. I am also grateful to John Jannotti, who has been involved in many of my studies at Brown. John always impressed me with his attention to detail and his deep and thorough understanding of systems research.

The Brown Database group is a small but productive community. I would like to thank its current and past members; Yanif Ahmad, Nathan Backman, Jennie Duggan, Hideaki Kimura, Alptekin Küpçü, Andrew Pavlo, Alex Rasin, Nesime Tatbul, Bradley Berg, JeHyok Ryu, Olga Papaemmanouil, Jeong-Hyon Hwang, and Tingjian Ge; for creating a nice environment for research.

I would also like to thank the following people, colleagues and friends, for their support and contributions to myself and my work: Çağatay Demiralp, Matteo Riondato, Serdar Kadıoğlu, Onur Keskin, Işın Çakır, Stuart Andrews, James Kelley, Jie Mao, Yong Zhao and Eric Koskinen.

Finally, I would like to thank my parents, Süheyla and Hüsnü, and my sisters, Yaprak and Yonca. They have been my greatest strength and support in life.

Contents

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Contributions 4

1.2 Outline 6

2 Background **8**

2.1 Model-based Prediction 8

2.2 Error Analysis 10

2.3 Stream Processing 11

3 DBMS Support for Predictive Models **12**

3.1 Prediction Queries 12

3.2 Integration of Prediction Models 14

4 The White-box Approach for Predictive Databases **16**

4.1 White-box Support of Prediction Models 16

4.2 Case Study: Database Representation and Support for Bayesian Networks 19

4.2.1 Background on Bayesian Networks 19

4.2.2 Inference with Bayesian Networks 21

4.3 Running Examples 24
4.3.1 The NIDS Application 24
4.3.2 The DyMon Application 25
4.3.3 Common Application Setup 26
4.4 White-box Support for CPQs using Bayesian Networks 27
4.4.1 CPQ Execution 27
4.4.2 Plan Selection 33
4.4.3 Model-Specific Optimizations 39
4.5 Experimental Evaluation 41
4.5.1 Setup 41
4.5.2 Network Intrusion Detection Results 41
4.5.3 Software Performance Monitoring Results 44
4.5.4 White-box Inference vs Off-Database Inference 47

5 The Black-box Approach for Predictive Databases 49
5.1 Black-box Support of Prediction Models 49
5.2 Prediction Interface 51
5.3 Model Management and Prediction 53
5.3.1 Model Registration 54
5.3.2 Model Representation, Building and Testing 55
5.3.3 Performance Tracking 57
5.3.4 Automated Model Building 58

6 Query Performance Prediction 60
6.1 Modeling Query Executions 61
6.1.1 Plan-level Modeling 63
6.1.2 Operator-level Modeling 64
6.1.3 Plan- versus Operator-level Modeling 66
6.1.4 Hybrid Modeling 69

6.2 Online Model Building . . . 73
6.3 Experiments . . . 76
6.3.1 Setup . . . 76
6.3.2 Prediction with Optimizer Cost Models . . . 78
6.3.3 Predicting for Static Workloads . . . 79
6.3.4 Predicting for Dynamic Workloads . . . 87
6.3.5 Platform Independence . . . 88

7 Related Work **91**
7.1 Machine Learning and Data Mining . . . 91
7.1.1 Computational Learning . . . 91
7.1.2 Learning Packages . . . 92
7.1.3 Database Support for Models . . . 92
7.2 Probabilistic Databases and Uncertainty . . . 93
7.3 Model-based Data Management and Query Processing . . . 93
7.4 Learning on Big Data . . . 95
7.5 Query Performance Prediction . . . 96

8 Conclusions **98**
8.1 Summary . . . 98
8.2 Open Challenges . . . 99
8.2.1 Prediction Query Optimizer . . . 99
8.2.2 Optimized Model Training . . . 101

Bibliography **103**

List of Tables

4.1 Computation and Storage Costs for Product-Join 34

5.1 List of Prediction Interface Arguments 51

5.2 Prediction Model SQL Interface 55

6.1 Features for Plan-Level Models 63

6.2 Features for Operator-Level Models 66

List of Figures

4.1 A Bayesian Network illustration 20
4.2 A Dynamic Bayesian Network illustration 21
4.3 Product-Join Operation 22
4.4 Marginalization in BNs 23
4.5 Intrusion Detection Bayesian Network 25
4.6 DyMon application setup 26
4.7 Execution Tree for NIDS query 27
4.8 A Dynamic Bayesian Network example 30
4.9 Execution tree generation for a point-based prediction query 31
4.10 Execution tree for a range-based prediction query 32
4.11 A two-slice DBN example 37
4.12 Execution plan for the range-based query on the two-slice DBN 38
4.13 Memory Usage vs. Computation tradeoff in the NIDS query 43
4.14 Query Execution Time vs. #Variables in the NIDS BN 43
4.15 Memory Usage vs. Computation tradeoff in DyMon 45
4.16 Query Execution Time vs Query Range in DyMon 46
4.17 Top-k Query execution times 47

5.1 Design schema for black-box model support 56

6.1 A Learning Approach to QPP 62
6.2 Operator-level QPP 67

6.3 Hybrid QPP example 70
6.4 TPC-H sub-plan analysis 75
6.5 QPP with optimizer cost estimates 79
6.6 Plan-level QPP in static workloads 81
6.7 Operator-level QPP in static workloads 83
6.8 Estimation Errors in QPP 84
6.9 Ordering Strategies in Hybrid QPP 86
6.10 QPP in dynamic workloads 87
6.11 QPP on a different hardware platform 89
6.12 Platform Independence: Hybrid QPP 89

Chapter 1

Introduction

Traditionally data management systems enabled users to efficiently query the past state of the world as is represented by the database. Many modern applications now rely on database systems that deliver up-to-date results with very low latency, hence enabling the user to query the present state of the world. Stream processing systems are designed to minimize this latency and produce near real-time answers [19, 1, 76]. Thus, we see a trend towards shrinking the "reality gap" to zero. But for some applications, even this is not good enough; there is often a desire to get out in front of the present by querying the predicted (i.e., forecasted) future state(s) of the database. Security applications are a good example for this, since they are typically interested in preventing a breach rather than simply reporting that one has happened. In a similar manner, some applications may leverage predictions for missing or unknown database values. Such *predictive* applications are increasingly deployed in order to identify and exploit opportunities, or avert calamities in a variety of IT or business monitoring, planning and decision-support scenarios.

Prediction is not new. Predictive modeling has been used with varying degrees of success for many years [44, 68]. Today, predictive modeling is a large sub-area of science that brings together many statistical techniques and algorithms from data mining, modeling and statistics to analyze and extract information from current and historical data. With the advances in computing power and storage capabilities, models grow more sophisticated

and data collection becomes more extensive and accurate, and as a result the quality of the predicted results are continuously improving. Modern weather forecasting is a good example of this. Twenty years ago, it would not have been possible to predict weather seven days ahead, nor would it have been possible to predict thunderstorms to the street level.

As a case in point, consider network intrusion detection. We can observe various characteristics of a network connection (such as the protocol used, duration, number of bytes sent, etc.) in real time and predict, based on historical traffic patterns accumulated a priori, whether the connection is likely to be an attack, either because it exhibits a similar pattern as a previous attack or it deviates from a typical connection pattern. In either case, we would like to flag suspicious connections as early as possible to trigger quick preventative action to avoid or mitigate potential damages (e.g., close the port and sandbox the corresponding server thread). As another example, consider the load management algorithms (e.g., job admission control and balancing) used in a data center for tasks such as achieving optimal resource utilization, minimizing response times and maximizing throughput. Standard algorithms typically "detect and react" to overload and imbalances, whereas predictive algorithms would facilitate a "predict and prevent" solution that reacts before the problem arises.

At present, predictive applications are not well supported by database systems, despite their growing prevalence and importance. Currently, most predictive applications follow an off-database approach in which the prediction functionality is provided outside the database system by specialized prediction software [38], which uses the database system primarily as a backend data server. Thus, the DBMS acts simply as a backend data source to feed the predictive application. Some database systems provide basic extensions [80, 54, 73], that facilitate the execution of predictive models on database tables (in a manner similar to stored procedures). As some others have also noticed (e.g., [34, 35]), we will argue that this loose coupling misses significant opportunities.

In this thesis, we demonstrate that there are many advantages to integrating predictive

mechanisms into the data management engine. As such, the broad goal of our work is to push this key predictive capability to the database layer in an attempt to leverage existing data modeling, query execution and optimization frameworks while providing generic, highly-efficient predictive functionality. Most of this functionality, which are available in database systems (already well debugged/optimized), needs to be duplicated to perform similar optimizations when prediction is done outside the database engine.

This thesis 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. In this study, we demonstrate techniques and a system architecture, as part of a PDBMS project called Longview [84], that allows us to efficiently support predictive functionality within database systems. We present the performance and usability advantages that PDBMSs offer over the traditional off-database prediction approach. We believe that the tight integration of model and data management, as demonstrated in Longview, will significantly broaden the applicability and appeal of database systems for new classes of data-intensive applications that leverage statistical models. We will see later that this dichotomy between data and model will show up as major components in our system design and architecture.

By providing predictive capability through declarative queries in the context of a database system, the programmers no longer need to concern themselves with the implementation and management of the prediction models or the low-level details of predictive tasks like model selection and training. The database system performs these tasks behind the scenes. Much as SQL has made programmers more productive in the context of data processing, we believe that our approach will have a similar effect for predictive programming tasks.

Also, integrating prediction with other more standard database operations allows the system to construct specialized structures (e.g., materialized models, data structures to organize those models) and the optimizer to build plans that make use of those structures.

Further, it allows for an execution model in which predictions are generated only as needed.

1.1 Contributions

In this thesis, we describe our work on supporting and leveraging prediction models for database applications within the context of a predictive database system being developed as part of the Longview project [84]. In particular, we describe how to provide predictive functionality in a database through prediction models, and discuss techniques to execute and optimize "prediction queries" on top of the prediction models by utilizing and extending the existing functionalities of a database system.

Prediction queries refer to the class of queries which use the existing data in a database system to build prediction models and estimate unknown data values. Prediction queries have a broad range of uses. 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).

We consider both one-shot and continuous prediction queries as part of this study. One-shot prediction queries, simply referred to as prediction queries, are ad hoc prediction queries which are only executed once. On the other hand, continuous prediction queries are standing queries that are executed multiple times to produce continuous results with updated "evidence". Evidence refers to the newly observed data values that affect the prediction results. In this study, techniques to efficiently support both types of prediction queries are discussed. For one-shot prediction queries, model building is usually the most time consuming operation and is therefore the main target of the proposed optimization techniques. For continuous prediction queries, incremental query optimization techniques and methods similar to multi-query optimization techniques are employed.

Furthermore, we investigate the use of prediction in the context of both continuous

stream-processing engines and standard database systems. While each of these settings raises interesting questions in their own right, we see the end-game as a system that combines both computing paradigms in an integrated and seamless way. Briefly, even in a real-time context, history is needed to do prediction.

An important premise of this work is that prediction models should be treated as first-class objects; meaning that models need to be managed in much the same way as data. For example, model instances will be materialized and stored. The period with which this pre-computation happens would be controlled by the system.

In this thesis, we discuss two fundamental approaches for integrating prediction methods into databases. The first method is to directly use the available implementations of prediction methods as *black-box* components in the database systems. Using the black-box approach, the database can immediately provide the predictive applications with already tested and optimized prediction methods without the otherwise required programming effort. The second approach, called the *white-box* method, is to implement the prediction methods within the database engine by extending the existing functionality of the database system. The white-box approach could require much more effort together with the expertise and the ability to manipulate the code for the database engine. However, in this way the inference operations are tightly integrated with query execution and additional query optimization opportunities involving the prediction operations, unavailable in the black-box approach, are possible.

The vision of this study involves dual uses of predictive query capabilities. The first use is *outward-facing*. Prediction queries can be used to answer application-specific questions in a variety of domains to identify opportunities, challenges, and trends that could potentially alert the application or the user accordingly (before the fact). For example, predictive analytics is common in financial services to support trading decisions, and in enterprise inventory management to predict future sales trends. As discussed before, currently database systems provide very little support for such predictive applications. Modeling and prediction are performed outside the database, which serves primarily as a dumb data store.

This approach is not only hard to implement but is also inefficient; as such, the goal of our study is to push as much of this functionality to the database as possible. In this study, we demonstrate the applicability and efficiency of our in-database prediction techniques for outward-facing applications with two case studies: a network intrusion detection system and a software performance monitoring application.

The second use of prediction is *inward-facing*: our efforts aim at the use of prediction queries as an introspective tool to assist various components of existing systems and facilitate intelligent data/resource management decisions. 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 in acquiring the appropriate statistics and efficiently and confidently predicting over them.

To demonstrate how the predictive functionality provided by a predictive database system can be leveraged to build such introspective services in existing systems, we consider the problem of query performance prediction (QPP) in database systems. Accurate QPP is central to effective resource management, query optimization and user experience management in a database. However, it is also a challenging task as database systems are becoming increasingly complex, with several database and operating system components interacting in sophisticated and often unexpected ways. As such, analytical cost models, which are commonly used by optimizers to compare candidate query execution plan costs, are poor predictors of execution latency. As a more promising approach to QPP, we demonstrate and evaluate predictive modeling techniques that accurately learn query execution behavior from historical data sets.

1.2 Outline

The rest of this thesis is structured as follows. We first provide background information in Chapter 2. Next, in Chapter 3 we describe prediction queries and introduce the white-box and black-box approaches to integrating prediction models into the database system.

In Chapter 4, we present our white-box model support techniques within the context of a streaming database system. Our discussion includes case studies to explain our optimization and execution methods for supporting predictive functionality. In Chapter 5, black-box model support techniques and the Longview system are described. Then, we demonstrate the usability and performance of the Longview system using a system-facing predictive application, Query Performance Prediction, as an in-depth case study in Chapter 6. Related work is discussed in Chapter 7. Finally, in Chapter 8, we present our final remarks and ideas for future work.

Chapter 2

Background

2.1 Model-based Prediction

We use the term model to refer to any predictive function such as Linear and Multiple Regression [68], Support Vector Machines [29], and Bayesian Networks [57]. The main property of a predictive function is learning: to generalize from the given examples in order to produce useful outputs for new inputs [74].

Prediction models serve a variety of purposes and have diverse characteristics. For instance, a prediction model is often designed only for a particular operation such as regression, clustering or classification. In addition, a model usually supports only a few restricted types of data (e.g., integers but not real numbers). Finally, with a few exceptions, different modeling and prediction tools correspond to different assumptions about the underlying stochastic process, and the prediction goals. The characteristics of the data being modeled is the most important feature in choosing the correct type of model. The assumptions made by the model should be in agreement with the data for accurate modeling and prediction. Hence, prediction models are often useful only for specific scenarios and a single type of model is not general enough for supporting a wide range of predictive applications.

In the machine learning approach, an input data set, called the training data, is initially used to tune the parameters of a prediction model. This process is called training (or

learning). The goal of training is to determine the best model instance that explains the available data set(s) while generalizing to new inputs. 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 involves selecting (i) a training data set and (ii) the training features (i.e., attributes or variables), a subset of all attributes in the data set, in addition to the learning operation. The training data contains both the prediction (i.e., target) attributes and the explanatory attributes which are the predictive features providing information on the target attributes. In practice, the original input data is usually *preprocessed* to transform the data into a new space of variables where solving the learning problem is expected to be easier. Normalizing the magnitude of numerical attributes, filling in missing data values, and removing outliers are examples of common preprocessing operations.

In some cases, a domain expert can manually specify the training features. In other cases, this step is trivial as the prediction attribute(s) directly determine the explanatory feature attribute(s) (e.g., in autoregressive models). Alternatively, training features can be learned automatically via *feature selection*; however, given a set of n attributes, trying the power set is prohibitively expensive if n is not small or training is expensive [68, 49] thereby requiring heuristic solutions. Most approaches rank the candidate attributes (often based on their correlation to the prediction attribute using metrics such as information gain or correlation coefficients [95]) and use this ranking to guide a heuristic search [44, 98] to identify the most predictive attributes tested over a disjoint test data set. The training data set can be sampled to speed up the process.

Once a prediction model is trained, it can then be used for predicting the unknown values of the target attributes given the values of explanatory attributes. The data set used in this prediction phase is usually called the test data, and only contains the data values for the explanatory attributes.

2.2 Error Analysis

Prediction accuracy is a function of the quality of the estimated models and the training data. The quality of the model (and the predictions) can be measured by distance and divergence metrics such as the variation distance or the mean square error between the predictions and the true values. With assumptions about the stochastic process, one may able to be bound these measures analytically, using large deviation theory, appropriate versions of the central limit theorem and martingale convergence bounds. 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 it estimates. While the empirical statistic is an unbiased estimate, the variance of the estimate can be large; it depends on the size and variance of the test set.

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 using various model types and feature sets. The process of selecting a statistical model among a set of candidate models is called *model selection* and is one of the fundamental tasks in machine learning and data mining. The performance of a model on the training data is not a good indicator of its predictive capabilities due to the well-known problem of *over-fitting*. The characteristics of the over-fitting problem is that such statistical models generally describe noise or error in the training data set instead of the underlying relationships between variables and therefore do not generalize to new test inputs.

A common practice is to use an independent/separate test data set for evaluating the performance of a prediction model. Yet in cases where the data is scarce, we would like to use as much of the data as possible for learning. In such cases, we can use a form of hypothesis testing: k-Fold Cross Validation (K-CV). K-CV divides the observed data up 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 and to predict that data in the validation

interval. To estimate the error with K-CV, in each iteration we can compute, for instance, the root mean square error for the pairs (a_i, e_i) where a_i is the actual value and e_i is the predicted value for each of the n/k points in the validation partition. Other methods such as regularization, pruning, prior-based techniques also exist to overcome the problem of over-fitting [57]. These techniques usually rely on explicit terms which penalize complex models (e.g., models with many parameters) or on the use of independent validation data sets.

2.3 Stream Processing

Stream processing has been developed over the last decade as a response to the need for low latency query processing over an ordered and potentially infinite collection of tuples (e.g., [19, 1, 76, 20]). In stream processing, queries continue to execute as tuples arrive. The operators in a query mirror the standard relational operators with a few differences to account for the infinite nature of the input streams. The result of evaluating a query is typically one or more potentially infinite output streams. In order to achieve low latency, stream processing systems typically record their state in main memory. The most common form of transient state is the window which is a finite contiguous subset of the tuples in a stream [10]. Windows move along the stream in response to arriving tuples and a sophisticated set of rules for window definition.

Chapter 3

DBMS Support for Predictive Models

3.1 Prediction Queries

In this thesis, we describe our work on providing support for predictive applications in database systems through the use of declarative prediction queries built on top of database-integrated prediction models. Declarative queries provide an efficient and easy-to-use interface to predictive applications for using the database-supported predictive functionality. To this end, we also discuss SQL extensions to easily express predictive operations within SQL and demonstrate the usability and efficiency of the declarative prediction approach.

We define prediction queries as a new class of queries that performs inference based on the existing data in a database system to produce estimates of future or missing data values. The inference operation is performed as part of the query execution process using the prediction models supported by the database system. In this section, we describe the types of prediction queries considered in this thesis.

We consider both *one-shot* and *continuous* prediction queries:

One-shot prediction queries refer to the set of prediction queries which execute and produce results only once. They are typically ad-hoc, customized queries which may

perform any type of inference on any subset of the available data attributes.

Continuous prediction queries (CPQs) [7] are standing queries, which are repeatedly executed to produce continuous results. A CPQ may produce results whenever new data (i.e., new evidence) is observed or when its set of parameters (e.g., parameters for selection, predicates or prediction operations) are updated and new results are required. Alternatively, a CPQ might produce periodic results with given time intervals.

In general, a predictive application could opt to use one-shot prediction queries, CPQs or both depending on its requirements. However, CPQs are generally a better fit for streaming predictive applications whereas in traditional environments we expect the use of one-shot prediction queries to be more common.

There are also performance considerations when comparing the types of prediction queries. Observe that, according to our definition, CPQs can also be viewed as independently executed multiple one-shot queries. However, this proposed approach to CPQs is not favorable, as in this case, the database client is required to repeatedly submit one-shot prediction queries to the database.

On the client side, this might imply polling the database for updated evidence and new data (or using a database trigger as a notifier), modifying the one-shot query with the newly received data and resubmitting the modified query to the database. On the database side, the situation is even more problematic. The main issue is that the database is not aware of the continuous execution process for the prediction query. Therefore, each time the query is submitted, the database needs to re-parse, re-plan and re-execute the query from scratch. This process will also include re-building (i.e., re-training) the prediction models used by the query thereby causing significant overhead in each query submission.

Most importantly, the database system is prevented from applying additional query optimization techniques to optimize the continuous execution processes. For instance, materialization techniques could be used on prediction results across multiple executions of the

query to reduce the query execution times. In this thesis, we will describe such database-style integrated query optimization techniques both for one-shot and continuous prediction queries.

3.2 Integration of Prediction Models

We study the integration of prediction models as first-class entities in the database system. In our approach, prediction models are managed in much the same way as data is managed in the database system. As such, we consider model management as the key underlying component of our approach. In this thesis work, we demonstrate that model management can greatly benefit from analogues of many well-established data management techniques.

Below we highlight some of these optimization opportunities that we explore in thesis for model management:

Profiling and modeling: Computation/storage costs and accuracy characteristics of prediction models can be modeled, and fed to the query optimizer so that efficient execution plans and accurate model types can be chosen for performing a given predictive task.

Physical design and specialized data structures: Data can be structured to facilitate efficient model building and predictive query execution. Well-known data management and access methodologies such as (multi-dimensional) indices and disk organizations can be adapted and extended for this purpose.

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, similar to the way a DBMS pre-computes indices and materialized views, for use by the optimizer and executor. Furthermore, this process can be automated in many cases.

Query optimization: The optimizer can be altered to consider alternative ways of model

building, selection, and execution, as well as the inherent cost-accuracy tradeoffs when selecting an execution plan for a predictive query.

Prediction models can be supported by a database system at varying degrees of integration. As mentioned before, in this thesis we consider two alternative ways of model integration: black-box and white-box methods.

In the black-box method, existing implementations of prediction models are directly linked or embedded into the database as user defined functions (UDFs) or external libraries. On the other hand, in the white-box approach predictive methods are tightly integrated into the core of a database system, and usually supported as native database operations.

Depending on the integration method used for a prediction model, the kind and effectiveness of the data management style optimizations that we can utilize for improving its execution performance vary. In general, with the white-box approach we are able to adapt more of the existing functionality in a database system for efficient prediction due to the low-level integration into the database engine and the transparent functionality of the prediction operations. As such, optimization opportunities for black-box methods are limited. Hence, we mostly consider providing efficient higher-level predictive functionality and modeling capabilities within the context of the black-box approach. We demonstrate the usability and extensibility of our black-box techniques in Chapter 6 using a system-facing application, Query Performance Prediction, as our case study.

Chapter 4

The White-box Approach for Predictive Databases

4.1 White-box Support of Prediction Models

White-box model support refers to the approach of integrated implementation and support of predictive functionality within the database engine instead of the use of black-box implementations by the database system. In particular, the aim of white-box modeling is to implement the functionality of prediction models using the available query execution operators and other (possibly extended) components of a database system. Therefore, in this approach prediction models are natively supported by the database query processing mechanism rather than model-specific black-box implementations. The white-box approach also includes other available forms of database support to prediction models (e.g., state management for prediction models or implementation of the underlying mathematical entities) based on information about their inner workings.

As with the white-box approach the operations of prediction models are part of the database query processing mechanism, it follows that the query optimizer is able to optimize the execution of prediction models. Hence, we are able to benefit from the well-tested

and complex optimization rules and transformations of the query optimizer in running prediction queries. In addition, with the use of white-box prediction models, the database query optimizer could also be extended with new optimization rules and algorithms to obtain additional performance gains by aggressively optimizing the query plans for prediction model operations. Additional optimizations of the prediction operations are essential, especially for the support of near real-time predictive applications. Moreover, in complex queries having prediction queries as their sub-queries, the optimization and execution of the prediction models are intertwined with the rest of the query. This brings up new optimization opportunities to the query optimizer.

Observe that not all types of prediction models can be directly supported within the traditional relational-model of database systems. Certain models such as Bayesian Networks (BNs) [57, 82] and polynomial-based regression models can be naturally represented and used in the relational model [17, 4], while other model types require the use of custom implementations either through user defined functions or external hooks. The characteristics of the operations required by a prediction model determine if it can be directly supported within a database system. More specifically, the functionality of a prediction model is fully implementable as a regular database query if its operations are expressible in relational algebra terms. It is also possible to extend the relational algebra model supported by a database system to support more complex prediction models [96].

In Section 4.2, we provide background information on Bayesian Networks and describe how they can be supported by a database system based on the white-box approach. Previous work showed how BNs can be represented by a functional relational model and predictive queries be supported using extended relational operators [17, 96]. In Section 4.4, we build on these results and discuss how to execute and optimize continuous prediction queries (CPQs) on top of BN-based prediction models, specifically the discrete Bayesian Networks and their variant Dynamic Bayesian Networks (DBNs) [77], in a streaming database system. There is a large suite of predictive models, including regression models and classifiers. Yet, among these (D)BNs constitute an important class that is widely used in practice. Their

common use and that they can be naturally represented and used in the relational model make them a good candidate for native DB support.

BNs are not new; they have been extensively studied in a variety of domains including AI, machine learning and statistics [57, 82]. As such, our contribution is not to introduce a new BN technique but to demonstrate how existing BN approaches can be *natively* supported by a database system to perform highly-efficient CPQ over streaming data. We describe how to create a rich plan space for CPQs and perform cost-based optimization in this space. Our primary contribution is a cost-based prediction query optimization and execution framework that combines materialization, sharing and model-specific optimization techniques for CPQs using BN-based prediction models.

In Section 4.3, we describe two streaming database applications that we use to demonstrate our query execution and optimization methods for the BN-based prediction models. As mentioned, in the context of the white-box modeling approach, we focused our studies on the support of BN-based predictive functionality with continuous prediction queries in a streaming database environment. However, the representation and use of BN-based prediction models is studied within the context of a general relational model and is therefore applicable to both traditional and streaming scenarios. As such most of the algorithms and optimization methods presented in this chapter are also applicable (or easily adaptable) to the traditional disk-based scenarios with small changes to the optimization metrics (e.g, using disk I/O instead of computation cost). Discussion on the one-shot prediction queries, their execution and optimization, is provided in Chapter 5, the Black-box Approach for Predictive Databases.

4.2 Case Study: Database Representation and Support for Bayesian Networks

4.2.1 Background on Bayesian Networks

Bayesian Networks (BNs) [57, 82] are compact representations of joint distributions over sets of variables. The compactness is achieved by utilizing the conditional independences among the variables. A BN consists of a directed acyclic graph (DAG) that includes a node and a conditional probability distribution (CPD) for each variable. The CPD of a variable encodes its distribution given its parents in the graph. Thus, for a graph G with N nodes $x = \{x_1, x_2, \ldots, x_N\}$, the joint distribution is given by

$$p(x) = \prod_{i=1}^{N} p(x_i|pa(x_i))$$

where $pa(x_i)$ denotes the parents of x_i.

Previous work used functional relations to represent the conditional probabilities of a BN in a relational database [17, 97, 96]. A functional relation R has the schema $\{A_1, A_2, \ldots, A_n, f\}$ where f is called the measure attribute and the functional dependency $A_1, A_2, \ldots, A_n \rightarrow f$ holds. In this case, the measure attribute corresponds to the conditional probability for the configuration represented by a tuple. An example BN consisting of the binary attributes X, Y, Z and T is shown in Figure 4.1 together with the functional relations for each CPD.

Dynamic Bayesian Networks (DBNs) [77, 42] are a natural extension of the Bayesian Networks (BNs) for modeling dynamic systems, i.e., systems which are represented with variables that evolve with time. Some basic continuous prediction queries that the users would like to pose in such systems could include point-based and range-based temporal predictions:

Query 1: The expected CPU usage of process p in the next minute (i.e., forecasting at a future time point)

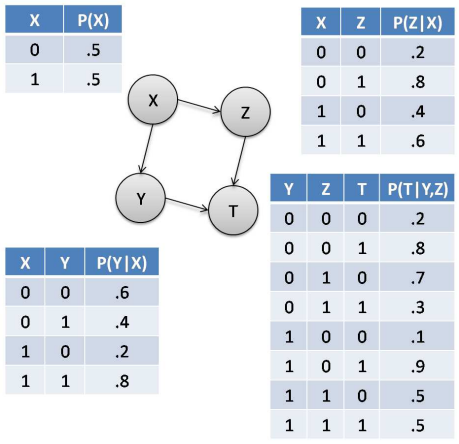


Figure 4.1: **An example Bayesian Network, defined over the binary attributes X, Y, Z and T, representing the joint distribution** $P(X, Y, Z, T) = P(X)\ P(Y|X)\ P(Z|X)\ P(T|Y, Z)$**.**

Query 2: The expected CPU usage of process p for the next 10 minutes (i.e., forecasting at a future time interval).

To answer Query 1, a BN can be constructed such that the value of the attribute CPU at different time steps is represented using different variables. In this case, the BN does not consider the temporal causality between the variables and does not utilize the fact that it is a single variable observed at different time steps.

A similar approach can be taken to answer Query 2. However, in this case we will need to introduce 10 variables, as we require a separate CPU variable for each time step. This approach is not practical since both representing the BN and executing inference will quickly become infeasible with the growing number of variables. Moreover, if we would like to predict a different range of CPU values, say the next 15 minutes instead of 10 minutes, we would have to extend the BN structure with additional nodes.

DBNs represent a similar modeling approach except that there are certain restrictions which help reduce the size and complexity of the network structure and thereby make inference more efficient. The restrictions involved in a DBN can be briefly stated as follows:

1. *Forward linking:* No backward links in time.

2. *Temporal consistency:* If there is a link from $x[i]$, the node representing variable x at time point i, to $x[j]$, then there is a link from $x[i+k]$ to $x[j+k]$ for all k.

3. *Identical distributions:* Conditional probability distributions for the same attribute at different time steps are the same.

The DBN shown in Figure 4.2 can be used to answer both queries `1` and `2` (inference with BNs and DBNs is discussed later in this section). This is a simple DBN in which the current CPU value depends only on the previous value. The actual representation of the DBN inside the database consists of two time slices as shown in the figure.

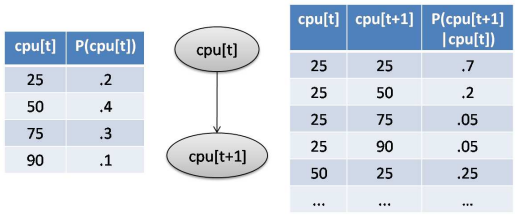


Figure 4.2: **An example DBN consisting of two time slices in which the current CPU value depends only on the previous value.**

4.2.2 Inference with Bayesian Networks

The inference problem in BNs is the problem of computing the posterior distribution of a set of target variables given the values of a set of observed/evidence variables. There are a variety of exact and inexact (or approximate) inference algorithms for BNs [57]. The exact algorithms compute the precise values for posterior distributions (i.e., marginals) and include variable elimination, clique tree propagation, cutset conditioning and other BN inference methods [57, 15, 67]. Examples of inexact BN inference algorithms are belief propagation, variational methods and MCMC algorithms [15, 59]. Almost all inference algorithms try to exploit the BN structure, which encodes the conditional independences, to efficiently compute the posterior distributions. In our work, we use the most common exact inference algorithm: variable elimination (VE) [100, 99].

The VE algorithm, like many other probabilistic inference algorithms, is based on a set of operations for manipulating probability distributions. These basic operations on probability distributions have been incorporated into relational systems by means of extended relational algebras [96, 17]. These extensions generally contain two main additional operations: `marginalization` and `product-join`, which we describe below using the BN shown in Figure 4.1.

- **The product-join operation** ($\overset{*}{\bowtie}$) is defined on two functional relations s and r as follows:

$$s \overset{*}{\bowtie} r = \pi_{s.a \cup r.a, s.f * r.f}\big(s \bowtie_{s.a \cap r.a} r\big)$$

  where $s.a$ and $r.a$ represent the non-measure attributes of relations s and r. For instance, in Figure 4.3 we apply the product-join operation on the variables X, Y and Z of the BN given in Figure 4.1 to get the joint distribution $P(X, Y, Z)$.

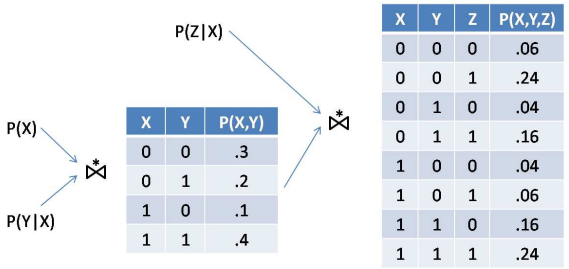


Figure 4.3: **Calculating $P(X, Y, Z)$ with the product-join operation.**

- **The marginalization operation** is defined over a joint distribution and is used to compute marginal distributions by eliminating variables. For instance, let r represent the distribution encoded by the BN in Figure 4.1 over all variables X, Y, Z and T, i.e.,

$$r = PX \overset{*}{\bowtie} PY \overset{*}{\bowtie} PZ \overset{*}{\bowtie} PT$$

where we refer to the relations of the variables X, Y, Z and T with PX, PY, PZ and PT respectively.

Then, to compute the marginal distribution over a subset A of all the variables in the joint distribution, we eliminate all variables not in A by applying:

$$\pi_{A,sum(r.f)}(GroupBy_A(r))$$

Observe that, the defined product-join operation is both associative and commutative.

In Figure 4.4, we calculate the marginal distribution $P(Y, Z)$ by eliminating the variable X from the joint distribution $P(X, Y, Z)$ (computed in Figure 4.3). Observe that in this case, we did not need to construct the full joint distribution r as the variable T does not contribute to the probability of Y and Z. Identifying such redundant variables has been discussed in literature before [57] and helps reduce the required computation.

$\Pi_{Y, Z, sum(P(X,Y,Z))}$ (GroupBy $_{Y, Z}$ (P(X,Y,Z)))

Y	Z	P(Y,Z)
0	0	.1
0	1	.3
1	0	.2
1	1	.4

Figure 4.4: **Calculating $P(Y, Z)$ using marginalization on $P(X, Y, Z)$.**

Now, we can express the variable elimination algorithm simply as a series of product-join and marginalization operations on the base relations. The problem of efficiently ordering these operators was recently addressed and the algorithm was integrated into the query optimizer in the database core [17].

While inference in BNs can be implemented based on the mentioned variable elimination algorithm, in the case of DBNs we may have to *unroll* the DBN before we apply the variable elimination algorithm. Consider query 2 described using the DBN given in Figure 4.2. To

predict the next 10 CPU values, we would have to unroll the network by adding new time slices consisting of the nodes $cpu[t+2], \ldots, cpu[t+10]$. Inference can then be executed similar to the BN case.

4.3 Running Examples

We describe our white-box modeling support techniques for continuous prediction queries and the corresponding optimization methods via two representative use cases: (i) a network intrusion detection system (NIDS) adapted from [53] and (ii) a dynamic software performance monitoring (DyMon) application [88, 89]. NIDS is primarily used to illustrate the BN-based prediction techniques, whereas DyMon is used to discuss the temporal CPQs supported by DBNs.

4.3.1 The NIDS Application

We base our network intrusion detection application on the 1999 KDD Intrusion Detection Contest [53]. In the NIDS application, the task is to detect intrusions in a computer network by monitoring the network connections. In particular, in the Intrusion Detection Contest each network connection is described with a set of 42 variables e.g., 'protocol type', 'duration' and 'number of failed logins'. Given these 42 variables for each connection, we would like to predict their types, represented with the additional variable 'access_type' being one of 'normal' or 'attack'. The KDD99 contest also involves identifying the attack type for bad connections, which we do not consider in this study.

We use the BN given in Figure 4.5 for predicting the access type of each connection. The given Bayesian Network provides highly accurate prediction results despite using only 6 of the given 42 variables. It is essential to produce such simple networks with high accuracy properties for efficient inference. We used a similar method to that described in [35] for forming the network structure. First, we ranked the given 42 variables according to their correlation with the `access_type` attribute based on the Linear Correlation Coefficient

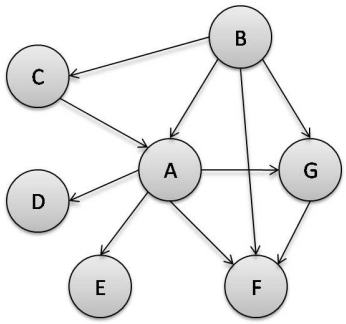


Figure 4.5: **Network Intrusion Detection System BN. Variable A is the `access_type` and others are the observable variables. The BN corresponds to the factorization:** $P(A, B, C, D, E, F, G) = P(B)P(C|B)P(A|B, C)$ $P(G|A, B)P(D|A)P(E|A)P(F|A, B, G)$.

metric [95]. Then, starting with a minimal subset of the 42 variables, we iteratively increased the size of the subset, using a best-first search technique, until we could create a network with high accuracy for identifying the connection type.

The generic prediction query we would like to run on this network is $P(access\_type \mid evidence)$, where evidence is used to represent the values of the observed variables. This is a continuous prediction query in the sense that every time a new connection information is received, it produces a prediction.

4.3.2 The DyMon Application

The second application we use for demonstrating our white-box execution and optimization methods for prediction models is a software performance monitoring system. The Dynamic Software Performance Monitoring (DyMon) [88, 89] application attaches profiling agents to local and remote Java processes and monitors runtime performance metrics such as number of threads, number of I/O calls, and CPU/memory usage.

In this study, we monitor the performance of a Java-based web server, named Jetty [58], under a load of web requests replayed from the 1998 FIFA web logs [37]. The high-level setup of our software performance monitoring application is shown in Figure 4.6. The system is also augmented with additional monitoring software to continually acquire additional

performance metrics not monitored by the DyMon agents (e.g., the number of received HTTP requests and the average time of processing a request). We will discuss both point- and range- based prediction queries using the DyMon application. We will use the DBN shown in Figure 4.8 for illustrating our query optimization techniques.

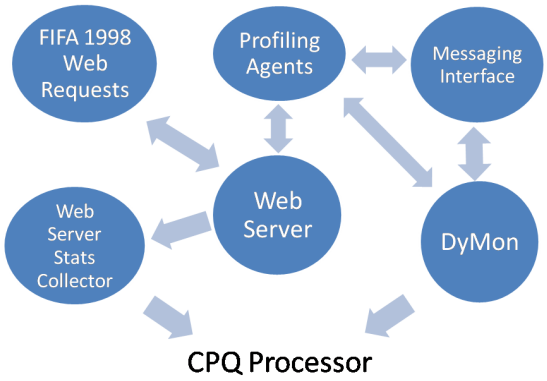


Figure 4.6: **The setup for the DyMon application. The web server is under a load of web requests replayed from the FIFA 1998 web logs. The Continuous Prediction Query Processing system receives its information both from DyMon and a custom monitoring software.**

4.3.3 Common Application Setup

The execution setup for both of the applications is a single-node streaming database system where the prediction models and the data are managed, the streaming data is received and the prediction is performed. Both applications require continuous prediction results with each received streaming tuple. The received tuples are formed by concatenating the values of the observed attributes and feed the prediction models with data to produce predictions on the target attributes. We assume that we are provided with training data beforehand using which prediction models are to be built. The training data includes values for the prediction attributes together with the observed attributes (i.e., supervised learning).

4.4 White-box Support for CPQs using Bayesian Networks

4.4.1 CPQ Execution

A. Query Execution with Bayesian Networks

We describe our continuous prediction query execution techniques for BN-based prediction models using the network intrusion detection example given in Section 4.3. In the NIDS connection type prediction query, we would like to compute the probability $P(access\_type|evidence)$ for each network connection. The *access_type* variable represents the type of a network connection with the values 'attack' and 'normal'. Using the Bayesian rule, we have:

$$P(access\_type|evidence) = \frac{P(access\_type,evidence)}{P(evidence)}$$

which is proportional to $P(access\_type, evidence)$ as $P(evidence)$ is constant. Hence, we only need to compute $P(access\_type, evidence)$ and normalize it to produce the query result. Substituting in the variable names used in the NIDS Bayesian Network given in Figure 4.5 and using lowercase letters for observed variables, our task is to compute $P(A, b, c, d, e, f, g)$. We use the example query evaluation tree given in Figure 4.7. The query tree contains product-join ($\overset{*}{\bowtie}$) and selection (σ) operations which were described in Section 4.2 as part of the BN-based inference with databases discussion.

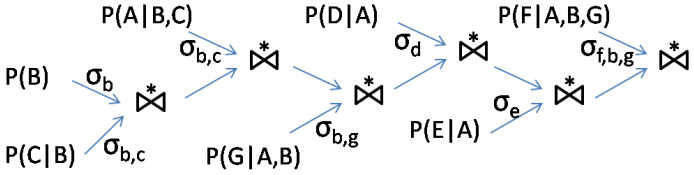


Figure 4.7: **An example query execution tree for** $P(A, b, c, d, e, f, g)$**.**

Selection constraints are used for propagating the evidence (i.e., observed values) in the query tree. They are pushed down to the probability distributions. In continuous query execution, every time a new tuple is received, the σ-constraints are going to be modified with the new observed values. This approach is similar to parameterized query execution

techniques with `prepared` statements [78].

Materialization Options: Consider the continuous evaluation of the product-join operation between $\sigma_b(P(B))$, denoted with $P(b)$ and $\sigma_{b,c}(P(C|B))$, denoted with $P(c|b)$, as new b and c values are received. We have the following execution options:

- **recompute**: compute $P(b) \overset{*}{\bowtie} P(c|b)$ every time b and c values are received.
- **materialize** $P(B) \overset{*}{\bowtie} P(C|B)$: precompute $P(B) \overset{*}{\bowtie} P(C|B)$ and store it in memory as $P(B, C)$. Then, to compute $P(b) \overset{*}{\bowtie} P(c|b)$ we need to execute a selection constraint, $\sigma_{b,c}$, on $P(B, C)$. Observe that, we no longer need $P(B)$ or $P(C|B)$. Hence, in some cases materialization may help us reduce both memory usage and computation at the same time.
- **partially materialize** $P(B) \overset{*}{\bowtie} P(C|B)$: precompute $P(B) \overset{*}{\bowtie} P(C|B)$ and store an α-factor of it in memory as $P_\alpha(B, C)$. Here, α is a probability value and $P_\alpha(B, C)$ is the subset of $P(B, C)$ consisting of a minimal number of highest probability tuples whose cumulative probability is greater than or equal to α. In this case, to compute $P(b) \overset{*}{\bowtie} P(c|b)$, we will first check $P_\alpha(B, C)$, and we will only do the product-join if the answer is not found. Observe that the answer will be found in $P_\alpha(B, C)$ with probability α.

Hence, in constructing query plans for each product-join operator, we have alternatives we can use to trade-off computation and memory requirements. Note that when α is 0, partial materialization is equivalent to recomputation, and when α is 1, it is equivalent to full materialization. Therefore, different α values enable us explore the space between these two extreme options.

The base conditional probability distributions, materialized and partially materialized distributions are all either stored in sorted order of the observed variables or there is an index defined on the observed variables of the distribution. Hence, selection on the distributions can be implemented as a fast-lookup operation in memory. For instance, $P(D|A)$

could be sorted on D and σ_d can then be implemented as an in-memory binary search operation.

CPQ Execution Flow: Consider CPQ execution with the query tree given in Figure 4.7. The execution starts with the product-join at the top level. If the product-join result is materialized, then the whole query execution becomes a simple selection on the joint distribution. However, this is unlikely to be the case even with a moderate number of variables, since the joint distribution may become quite large easily. For instance, if we have n variables each with a small domain size of 10, then the joint distribution could have as many as 10^n tuples.

If the product-join at the top level is partially materialized, we first look for our answer, with a selection operation, in the materialized part of the product-join. Depending on the answer, we may or may not need to execute the lower levels of the query tree. In the worst-case scenario, we need to traverse down to all the leaves of the query tree, in which case the execution is equivalent to full recomputation over all the tree. In the best case, however, a small fraction of a distribution will have significant probability mass which will enable our system to materialize a very small but most frequently accessed part of the distribution.

Marginalization: We do not need to eliminate any variables (i.e., perform marginalization) to compute the probability $P(A, b, c, d, e, f, g)$. Hence, there are no group-by or projection operations, which are used to implement the marginalization operation in a database as discussed in §4.2, in the query tree of Figure 4.7. However, we might have to eliminate variables in many other cases. This issue is more significant for inference in DBNs and is therefore revisited in the DBN-based Query Execution discussion in the next section.

B. Query Execution with Dynamic Bayesian Networks

We use the DBN shown in Figure 4.8 to illustrate the execution of both point- and range-based prediction queries. Point-based CPQs return a probability distribution on the values

of a given set of attributes at some time point in the future, given information about their current and past values. On the other hand, range-based CPQs return a set of point-based prediction results for a given time interval.

At a high level, the execution of both types of queries proceed similar to the execution of CPQs on BNs in the sense that (i) materialized results are utilized whenever possible to avoid recomputation, and (ii) selection constraints are used to propagate evidence throughout the variables. However, with CPQs on DBNs there is more opportunity for sharing computation and materialized results due to the structure and assumptions of a DBN, especially for the range-based queries. Below, we discuss query plans and their execution for both types of DBN-based CPQs.

Point-based CPQs: For the DBN shown in Figure 4.8, we consider point-based prediction queries of the form $P(cpu[t+k]|cpu[t], cpu[t-1])$ for $k > 0$. When $k = 1$, query execution is similar to that with BNs. However, when $k > 1$, we need to *unroll* the DBN as shown in the figure, until it includes the variable $cpu[t+k]$.

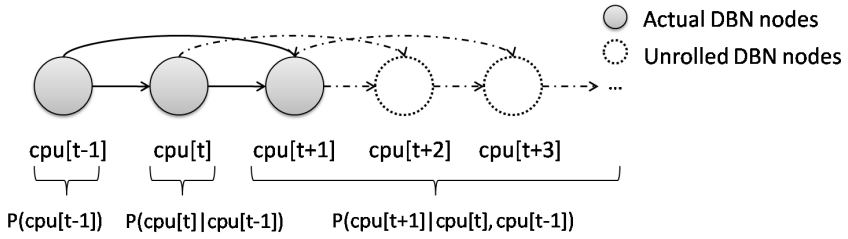


Figure 4.8: **A three time-slice DBN for the CPU variable is unrolled to include future time points. Newly added nodes share the same distribution with the variable cpu[t+1].**

An execution tree for the point-based prediction query with $k = 3$ is obtained through repeated multiplication of the distributions in sequential order. This process is shown in Figure 4.9. The projection nodes in the figure correspond to the marginalization operation, and therefore are preceded by "group by" nodes.

We can construct execution trees for prediction queries defined with arbitrary values of

k in a similar way: starting from the query execution tree given for the query where $k = 3$, we repeatedly "append" the query tree shown on the right side of the figure for increasing values of j $(j = 4, 5, \ldots, k)$ to the end of the tree.

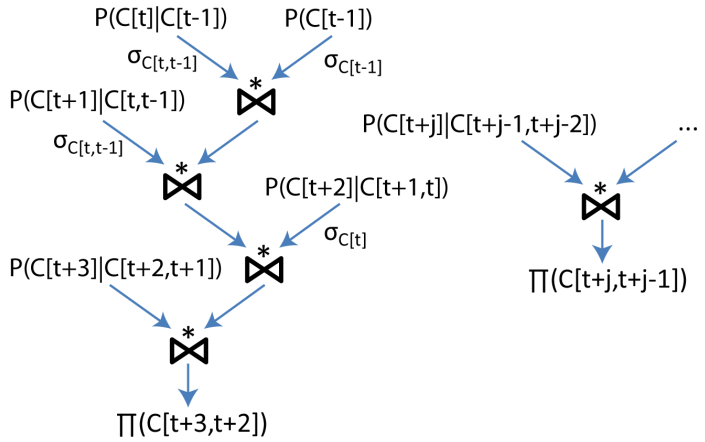


Figure 4.9: **An execution tree for calculating the probability** $P(cpu[t+3] \mid cpu[t], cpu[t-1])$ **is given on the left (**C **is used to denote CPU). A normalization operation, not shown in the figure, is to be added to the end of the tree. On the right is a template query tree for the described iterative operation in generating plans for arbitrary point-based prediction queries.**

Range-based CPQs: According to our definition, range-based queries return a set of independent point-based prediction results for a set of variables within a given time interval. For the DBN in Figure 4.8, the result of the range query at time t would be $P(cpu[t+k]|cpu[t], cpu[t-1])$, $\forall k \in \{1, \ldots, m\}$ where m is the length of the range. The alternative would be to return a joint distribution instead of the point-based results. That is, we would return $P(cpu[t+1], cpu[t+2], \ldots, cpu[t+m]|cpu[t], cpu[t-1])$. However, as m increases the size of this distribution would become very large, making its computation and storage impractical. In addition, the individual probabilities for each configuration of the distribution would diminish with increasing m. Hence, while our system can compute the joint distributions for relatively small ranges, we focus on supporting the first definition of the range query in the rest of this section. Later in this section, we discuss how to efficiently compute only the top-k most likely events of the joint distribution.

Observe that, according to our definition, range-based queries can also be considered as multiple point-based prediction queries. If we take this naive viewpoint, we can actually execute multiple point-based queries independently to compute the result of the range-based query. However, we use execution plans utilizing multi-query execution techniques to share the computation and storage across these seemingly independent (but in fact causally dependent) queries.

We first note that the execution tree for the point-based query $P(cpu[t+k] \mid cpu[t], cpu[t-1])$ is actually built on the execution tree of the query $P(cpu[t+k-1] \mid cpu[t], cpu[t-1])$. Hence, the most basic optimization is to share the computation across these queries by using a combined execution plan as shown in Figure 4.10.

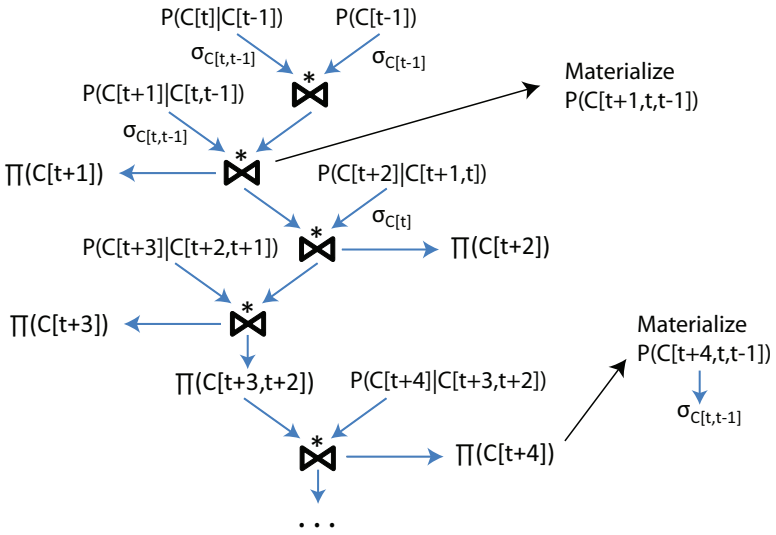


Figure 4.10: **An execution tree for the range-based query computing** $P(cpu[t+k] \mid cpu[t], cpu[t-1])$ **for a range of** k **values is given for the first 4 time points (**C **is used to denote CPU). The marginalization operations containing only the observed variables are not shown.**

The query execution tree in Figure 4.10 shows the shared computation of the first four time points of the range-based query. It can be extended in a similar way to include all the time points in the query range. There are multiple output points in the query tree, one for each projection node without a parent. Each such node corresponds to a single time point

in the range of the query.

Similar to the sharing of computation, any materialized results (except for the materialization at the output nodes) can also be shared across the time points. In Figure 4.10, the second product-join node and the output node for $k = 4$ have been materialized. Observe that the schema of the materialized relation for the output node contains the CPU variables from times t and $t-1$, whereas its counterpart in the query tree does not have these variables. This is because we cannot push the selections down, eliminate the observed variables and then do the materialization since the parameters of the selection operations are not fixed. However, during the computation of the range query for the given values of the CPU attribute at times t and $t-1$, the selections can be pushed down and the computation cost can therefore be reduced.

4.4.2 Plan Selection

A. Cost Modeling for CPQs

We estimate the computation and storage requirements of each execution plan using simple statistics with a cost model. In our system, the processing on each accessed tuple is light-weight. As such, we base our computational cost model on the number of memory accesses incurred during query execution. This is consistent with the cost models used by main-memory database systems [16]. For storage costs, which we need for estimating the memory requirements of an execution plan, we assume uniform space requirements for all tuples.

Consider an intermediate relation X, formed during query execution, consisting of the observed variables O_X and the unobserved (hidden) variables U_X. The typical selection operation used for evaluating the prediction queries is to find the tuples with the given values o_X for the observed variables O_X. We assume that there is either an index on the observed variables or the tuples of X are sorted on the observed variables. Then, the cost of the selection operation, σ_{o_X}, on relation X is the sum of the costs for finding the location of the tuples satisfying the selection constraint and retrieving the tuples:

$$comp\_cost(\sigma_{o_X}(X)) = \log(|\pi_{O_X}(X)|) + \frac{|X|}{|\pi_{O_X}(X)|}$$

Now consider another relation Y that consists of the observed variables O_Y and the unobserved variables U_Y. The computation and storage costs for the product-join of X and Y, $X \overset{*}{\bowtie} Y$, are given in Table 4.1. We denote the selection factor between X and Y with σ_{xy}, which is calculated based on the attribute independence assumption. Recall that α is the probability factor used in partial materialization. Finally, θ is the ratio of the relation that gets materialized with the partial materialization method. θ depends on both α and the joint distribution represented by the observed variables of the relation, particularly its entropy. For a fixed value of α, the best case is where the entropy of the distribution is low, since then most of the probability mass will be concentrated on a few tuples and θ will be small. As the entropy increases, θ will generally get larger for a fixed α value. The worst-case scenario is where we have a uniform distribution, since in this case the entropy is maximized. As a result, α is an upper bound for θ. We will assume this worst-case scenario in our experiments; if θ can be better estimated, better results can be obtained.

method	storage cost	computation cost
recompute	0	$comp\_cost(\sigma_{o_X}(X))+$ $comp\_cost(\sigma_{o_Y}(Y))$
materialize	$\|X\|\|Y\|\sigma_{xy}$	$comp\_cost(\sigma_{o_X \cup o_Y}(\|X\|\|Y\|\sigma_{xy}))$
partially materialize	$\theta\|X\|\|Y\|\sigma_{xy}$	$comp\_cost(\sigma_{o_X \cup o_Y}(\theta\|X\|\|Y\|\sigma_{xy}))$ $+(1-\alpha)((comp\_cost(\sigma_{o_X}(X))+$ $comp\_cost(\sigma_{o_Y}(Y))))$

Table 4.1: **Computation and storage costs for the product-join operation: $X \overset{*}{\bowtie} Y$.**

The computation and storage costs for the marginalization operation are simpler to derive than for the product-join operation. The reason is that marginalization works on a single input and eliminates one or more variables from the input distribution. Hence, dividing by the size of the eliminated variables will give an estimate of the storage cost. Moreover, the computation cost, which is linear in the input size in the case of recomputation, can be derived similarly to the product-join case.

B. Generating CPQ Execution Plans

Next, we first modify the Selinger-style Dynamic Programming (DP) algorithm given for query optimization in [17, 23, 24], to find the query execution plan with the minimum computation cost that satisfies a given memory constraint for the case of BN-based CPQs. Then, we modify the proposed DP algorithm to generate plans for DBN-based CPQs and demonstrate additional optimization techniques for range-based queries.

Plan generation for BN-based queries:

The DP algorithm used with BN-based CPQs to find the query execution plans is given in Algorithm 1. At a high level, the algorithm constructs plans for growing subsets of base relations in successive iterations. At each iteration, the plans from the lower levels are used for forming the new plans. As presented, the algorithm considers only linear execution plans. However, it can be modified in a straightforward manner to consider nonlinear plans as well. The value of k on line 6 determines the granularity of α values we consider. For instance, if k is 2 then $\alpha \in \{.5\}$ and if k is 4, then $\alpha \in \{.25, .5, .75\}$. The notation $p \prec q$ is used to denote that plan p *dominates* plan q by yielding lower computation and storage costs.

Algorithm 1 Dynamic Programming based plan selection algorithm for BN-based CPQs.

1. S: the set of all base relations
2. **for all** $l \in 1 \ldots |S|$ **do**
3. **for all** $S_j : S_j \subseteq S \wedge |S_j| = l$ **do**
4. $p_{S_j}.add(materialize(S_j))$
5. $p_{S_j}.add(materialize(GroupBy(S_j))$
6. **for all** $i \in 1, \ldots, k-1$ **do**
7. $\alpha = i/k$
8. $p_{S_j}.add(partmaterialize(\alpha, optplans(S_j)))$
9. $p_{S_j} = \{q \in p_{S_j} : \nexists q' \in p_{S_j}$ such that $q' \neq q \wedge q' \prec q\}$
10. **for all** $r_j, S_j : r_j \in S \setminus S_j, S_j \subseteq S \wedge |S_j| = l$ **do**
11. $Q' = S_j \cup r_j$
12. $p_{Q'}.add(product\_join(optplans(S_j), r_j))$
13. $p_{Q'}.add(product\_join(GroupBy(optplans(S_j)), r_j))$

The "GroupBy" used in the DP algorithm refers to marginalization. The "optplans(x)"

returns all the non-dominated plans generated for computing the argument x. Hence, in lines 8, 12 and 13, the algorithm creates multiple plans. In addition, in line 8 where the partial materialization plans are considered, the non-materialized part of S_j may be computed using any non-dominated plan generated this far for S_j. A separate plan is created for each such option.

Given a set of base relations S, there are $|S|!$ linear plans for computing their product-join. In addition, there are $k+1$ possible options of materialization for each of the product joins: partially materialize into one of $k-1$ fractions, fully materialize or no materialization. Hence, there are $O(|S|!k^{|S|})$ different plans for computing the product-join involving the materialization options. Finally, after each one of the product-join operations, there can be a marginalization operation for eliminating variables. Therefore, the total number of plans together with the group by operations is $O(2^{|S|}|S|!k^{|S|})$. Observe that, to construct the plans for a set of relations of size l, we only need the plans for the relations of size $l-1$. Hence, in the worst case, we would have approximately twice the number of plans for relations of size $|S|$ in memory.

Plan generation for DBN-based CPQs:

For the **point-based queries** on DBNs, the Dynamic Programming algorithm discussed for the BN case can be used with minor changes. First, the DBN has to be unrolled as shown in Figure 4.8, until the target time point has been reached. Also, as the distributions of a variable are identical across time points in a DBN, we only need to store a distribution once and share it between the relevant variables. Note that this idea can be applied to more general situations involving operations over identical distributions as well. Consider a product-join between $X[t+k]$ and $Y[t+k]$. The result of this operation is the same for all k values where the variables X and Y are not observed at time $t+k$. Hence, we only need to compute it once. Observe that, in this case we save both computation and storage.

For the **range-based queries**, we need to produce outputs at all the time points in the given range. Hence, we cannot simply apply the described DP algorithm. In addition, as

the range given in the query specification increases, the number of variables in the unrolled DBN increases as well. In such a case, the DP algorithm will quickly become impractical as its complexity is exponential in the number of the variables.

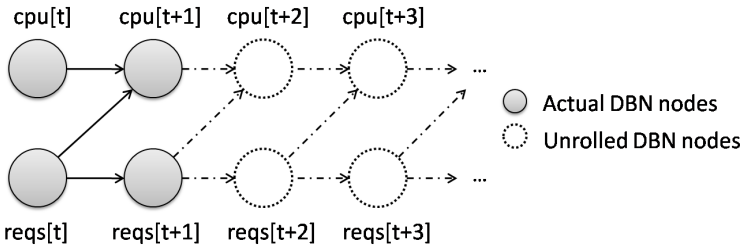


Figure 4.11: **A two-slice DBN consisting of the variables CPU and REQS. CPU represents the CPU usage and REQS is the number of requests received by the web server in the DyMon application.**

As discussed before, we use an alternative method that creates a template plan that can be iteratively applied to produce execution plans for the range queries. Consider the DBN given in Figure 4.11 which has two variables, CPU and REQS, in each time slice. We build an execution plan for the range-based query $P(cpu[t+k]|cpu[t], reqs[t]), \forall k \in \{1, \ldots, m\}$ for an arbitrary range value m in successive steps starting from the first time step. For all the initial time slices, which have at least one variable that directly depends on an observed variable, we call the described DP algorithm to incrementally generate the execution plan using the plans from the previous time slice. An example execution plan for this range query is given in Figure 4.12. In this example, the first call to the DP algorithm, for time step $t+1$, creates plans for the portion of the plan until the first project node. The second call, for time step $t+2$, then creates plans for the part of the plan till the second project node using the results of the previous run. Next, for time $t+3$, there is no variable that is either observed or depends on an observed variable, hence, in this step, we create the template plan that will be used to create the rest of the plan for this query.

The template plan is created using the DP algorithm as well. However, the computation and storage costs in the cost model are adjusted according to the number of times each operation needs to be executed for the rest of the time points in the query range. The

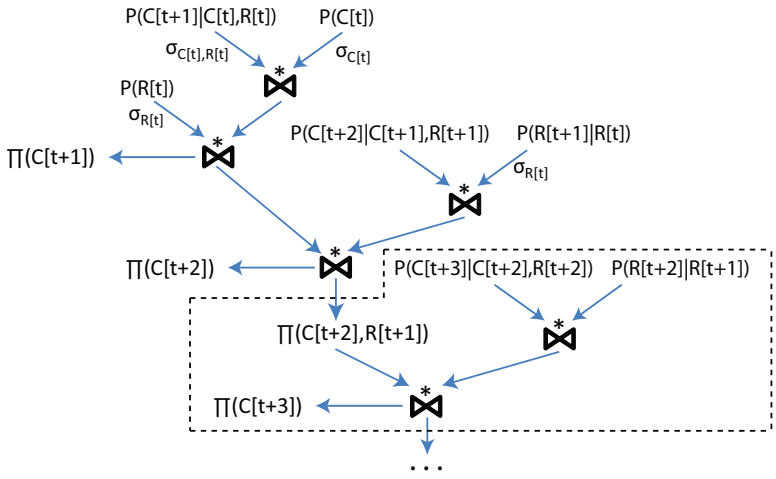


Figure 4.12: **A query execution plan for the range-based query predicting the CPU value based on the DBN shown in Figure 4.11. C is used to denote the CPU variable and R is used for the REQS variable. Finally, the highlighted area is the instantiation of the template plan for time 3.**

part of the plan highlighted with the dashed rectangle in Figure 4.12 is an instantiation of the template plan for time $t + 3$. The template plan has the same structure with the plan shown in the highlighted area, but represents the time values of the variables as adjustable parameters. Hence, we can similarly create the rest of the query plan by instantiating the template plan for increasing time values.

Observe that, within the template plan, if an operator depends on the results of a previous plan, the operator will have to be recomputed for each time point in the query range. For instance, the top-level product-join and both of the projection operations in the highlighted area will be computed (or materialized) separately for each time point. In such cases, the DP algorithm may choose to materialize the result for some of the time points and compute it for the rest of the time points. On the other hand, if an operation only depends on the relations introduced in this slice, then it can be computed or materialized only once and used in all the time points in the query range. For example, the product-join of $P(C[t + 3]|C[t + 2], R[t + 2])$ and $P(R[t + 2]|R[t + 1])$ needs only be computed (or materialized) once and then can be shared across multiple time points in the query range.

4.4.3 Model-Specific Optimizations

Prefiltering low probability events: In many cases, users are only interested in high probability events. For instance, a user could specify a probability threshold Θ, and then only ask for the events with probability values greater than Θ. In such cases, we can speed up the query execution by pushing down the probability constraints and eliminating low probability events early in query execution. Consider the product-join $P(X,Y) = P(X) \overset{*}{\bowtie} P(Y|X)$ and the constraint $P(X,Y) \geq \Theta$. Here, the constraint can be pushed down as $(P(X) \geq \Theta) \overset{*}{\bowtie} (P(Y|X) \geq \Theta)$.

In some cases, it is not easy or it just does not make sense to define such arbitrary thresholds but the user is still interested in high probability results. Consider the NIDS application where the task is to find the most likely type for a given network connection. If the connection is an 'attack', in many cases its probability value in the joint distribution is really low, but still higher than the probability of the connection being 'normal'. Hence, one cannot simply set a general probability threshold to eliminate all the low probability events. However, we can still find simple event elimination constraints for each of the operators. Consider the result, $P(A,b,c)$, of the second product-join operation in Figure 4.7, the product-join with $P(A|B,C)$. For any given values of b and c, there are at most two possible events: $A_1 = \{A =$'normal' $\}$ and $A_2 = \{A =$ 'attack' $\}$. For this scenario, the result of the inference query depends on the ratio:

$$r = \frac{P(A_1,b,c)P(g|A_1,b)P(d|A_1)P(e|A_1)P(f|A_1,b,g)}{P(A_2,b,c)P(g|A_2,b)P(d|A_2)P(e|A_2)P(f|A_2,b,g)}$$

Observe that for r we have the following bound:

$$\begin{aligned} r &\leq \frac{P(A_1,b,c)}{P(A_2,b,c)} max \frac{P(G|A_1,B)P(D|A_1)P(E|A_1)P(F|A_1,B,G)}{P(G|A_2,B)P(D|A_2)P(E|A_2)P(F|A_2,B,G)} \\ &\leq \frac{P(A_1,b,c)}{P(A_2,b,c)} max \frac{P(G|A_1,B)}{P(G|A_2,B)} max \frac{P(D|A_1)}{P(D|A_2)} max \frac{P(E|A_1)}{P(E|A_2)} max \frac{P(F|A_1,B,G)}{P(F|A_2,B,G)} \\ &= \frac{P(A_1,b,c)}{P(A_2,b,c)} p^{max}_{A_1/A_2} \end{aligned}$$

As a result if $\frac{P(A_2,b,c)}{P(A_1,b,c)} \geq p^{max}_{A_1/A_2}$ then we can eliminate the 'normal' event with b and c values (i.e., tuple A_1) from $P(A, b, c)$. Likewise, if $\frac{P(A_1,b,c)}{P(A_2,b,c)} \geq p^{max}_{A_2/A_1}$ then we can eliminate the tuple A_2.

We can derive bounds for all the product-join operators in Figure 4.7 using the same technique and reduce computation without introducing errors in the query results. in calculating the bounds for all the operators in the query tree. Alternatively, we can multiply the bound with a constant $1/\sigma$, where $\sigma \in [0, 1]$, to avoid eliminating the A_1 tuples with probability values greater than $\sigma P(A_2)$. This method can be used to produce the set of most likely events in which each event has a probability that is at least σ times the probability of the most likely event.

Top-k maximum probability events: In probabilistic databases, top-k queries are generally used to produce the k most likely results of a query [47, 91]. For instance, in the DyMon application one could specify a top-k query to generate only the top-k predictions of the number of web requests n time units in the future. A naive way to execute this query would be to produce the target distribution on the number of requests and then to output the top-k results. However, top-k queries are most useful when it is intractable to produce the target distribution. Consider the case when the user is interested in the most likely sequences of the number of requests in an interval of 10 time units. We mentioned before that it is impractical to produce the joint distribution of variables even for relatively small time intervals. Hence, the previous execution strategy is not viable in this case. However, a much more efficient approach that depends on the distributive properties of the top-k operator on the product-join operator exists. The top-k operator can be pushed down to eliminate the redundant events early in execution:

$$\underset{x,y,z}{\text{top-k}}(P(X,Y) \overset{*}{\bowtie} P(Y,Z))$$
$$= \underset{x,y,z}{\text{top-k}}(\underset{x}{\text{top-k}}(P(X,Y)) \overset{*}{\bowtie} \underset{z}{\text{top-k}}(P(Y,Z))).$$

Here, the top-k operator works on a list of free variables and an argument distribution. For instance, top-$k_x(P(X, Y))$ would return the top-k events for each value of Y.

4.5 Experimental Evaluation

4.5.1 Setup

Prototype Implementation: We implemented the described algorithms for continuous prediction queries in Java by modifying H2 [48], an open-source, embedded in-memory database engine. The database was used for storing the data in memory in an organized way and also enabled us to run regular SQL queries. In addition, we used the tree-based indexing functionality provided by the database for creating indices over the base and the materialized relations. The query optimizer and parts of the query executor were removed and replaced by our implementation. Our implementation follows the basic data-driven stream processing model, the queries are evaluated continuously as new inputs arrive.

Experimental Environment: Our experiments were done on standard desktop machines with AMD Athlon(tm) 64 X2 Dual Core 3800+ processors and 2GB of memory running Linux 2.6.26.

Experimental Metrics: Our primary performance metric is the average processing time per tuple. We report average processing latency of a tuple for various algorithms with different levels of available memory. As end-to-end processing latency of a tuple is our main metric, in our experiments each tuple was processed and consumed entirely before the next tuple. Thus, the data sets are replayed at a rate roughly inverse of average processing latency.

4.5.2 Network Intrusion Detection Results

The NIDS dataset was obtained from the 1999 KDD Intrusion Detection Contest [53]. Our dataset consisted of 500K tuples, 5K of which were used in testing the system and the rest for training (to learn the distributions and the network structure).

The resulting network structure used in the experiments consisting of seven variables is shown in Figure 4.5. While accuracy was not our immediate goal in this study, it is a useful metric to know, in order to have a sense of the eventual applicability of predictive

queries using BNs. Thus we report as a small side note that, for the NIDS experiment, we had approximately 99% accuracy in correctly identifying the type of connections in the test data.

Query Execution Time vs. Memory Usage: The average execution time versus memory usage (i.e. the #tuples stored in memory) results for the DP algorithm using the no materialization, full and partial materialization options are shown in Figure 4.13. The discussed tradeoff between materialization (memory usage) and computation time is clearly observed.

The results for DP with partial materialization option reflect the fact that only a fraction of the top-level joint distribution ($\approx 3\%$) have most of the probability mass ($> 90\%$). The reason is that most of the data actually consists of "normal" connections, as expected in all similar scenarios, and hence exhibit similar connection properties. As a result, if we only materialize this high probability portion of the distribution together with the base relations, we obtain a 3.50 ms query execution time with a memory use of 136 tuples. Moreover, if we materialize more than 312 tuples, then we actually get an execution performance better than materializing all the joint distribution represented by the BN, which is 870 tuples. The result is not suprising since selection on the full joint distribution takes longer than selection on the smaller high probability set even when using an index on the selection attributes.

In our cost model, we made a uniform distribution assumption for estimating the fraction of a distribution to materialize in the case of partial materialization. However, because the distributions in the NIDS dataset are actually highly skewed, our estimations for the plan costs and storage sizes are all overestimates. Hence, the DP with the partial materialization option chooses to materialize the whole distribution when there is sufficient memory. A better estimate would most likely produce better overall results as well as avoid the materialization of the joint distribution.

Number of Variables vs. Query Execution Time: In Figure 4.14, we present the query execution times as we increase the number of variables in the NIDS BN. The No Materialization option uses just enough memory to store the base relations. 2X and 4X

No Materialization		Full Materialization		
#tuples	Query Exec. Time	#tuples	Query Exec. Time	#tuples*
122	6.52 ms	146	5.28 ms	136
Partial Materialization		158	4.97 ms	158
#tuples	Query Exec. Time	203	5.01 ms	193
136	3.50 ms	272	4.83 ms	212
179	2.67 ms	533	3.91 ms	412
312	2.51 ms	870	2.66 ms	688
553	2.40 ms			
870	2.66 ms			

Figure 4.13: **Memory Usage (#tuples) vs. Computation tradeoff for the NIDS scenario. The #tuples* column shows the memory use after elimination of the low probability events for the full materialization case.**

Memory options are allowed memory usages up to 2 times and 4 times the size of the base relations respectively. Finally, the No Limit option is not given a memory constraint and therefore finds the minimum computation cost plans. When the number of variables is low, we can reduce the query execution time even with low memory budgets. However, as the number of variables increases, the size of the joint distributions quickly increases as well. At the same time, we observe a decrease in the relative performance gains with respect to the materialized size.

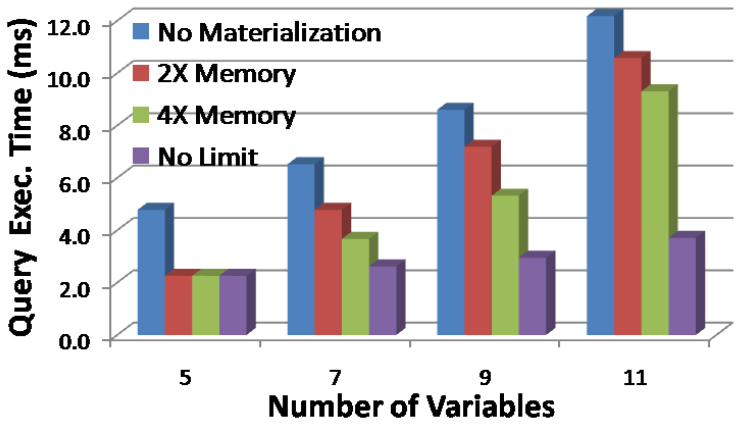


Figure 4.14: **The average query execution time vs. the number of variables in the NIDS BN presented for different levels of available memory.**

The results in Figure 4.14 are obtained using the DP algorithm with the full materialization option. We do not show the results for the partial materialization case, as they are similar to the results in Figure 4.13. Finally, it should be noted that the linear plan space is more favorable for the partial materialization option as it can partially materialize all the distributions and utilize all the available memory. However, with full materialization, there will be only a single materialized distribution (i.e. the highest level joint distribution that fits in available memory) for the BN-based CPQs (for DBN-based CPQs there can be multiple). When nonlinear plans are considered, we expect the full materialization strategy to perform better.

Eliminating Low Probability Events: We applied the techniques described in §4.4.3 for eliminating the low probability events from the materialized distributions. The results are shown in Figure 4.13, in the column labeled '#tuples*', for the full materialization option. As the sizes of the materialized distributions increase, generally we can eliminate more events. For instance, we have 20.92% memory savings (i.e. 182 tuples), from 870 to 688 tuples, when the whole joint distribution is materialized. In the NIDS application, the access_type attribute takes on two separate values, at most one of which we can eliminate for each configuration of other variables. Hence, greater savings would be possible with a larger domain size.

4.5.3 Software Performance Monitoring Results

Experiments on the DyMon application were performed using the setup described in §4.3. We collected 60,000 tuples using the described monitoring facilities from the monitored web server for training purposes. During the data collection, the web requests obtained from the FIFA web logs were replayed on the web server. We used the first web logs of the 50^{th} day of the FIFA Cup. We also scaled down the number of requests per second in the logs by a factor of two to avoid overloading our web server. Each collected data tuple is a summary of the performance of the process in a period of length approximately 500ms.

Partial Materialization vs. Full Materialization: For the range-based prediction

queries, $P(cpu[t+k]|cpu[t], cpu[t-1])$, we compare the average execution times of the query execution plans obtained using the partial and full materialization options of the DP algorithm with a query range $k = 5$. Results are shown in Figure 4.15. For the partial materialization option, we only show the results for $\alpha = 0, .5$ and 1. A finer-grained range of α values produce similar results, albeit more options for materialization.

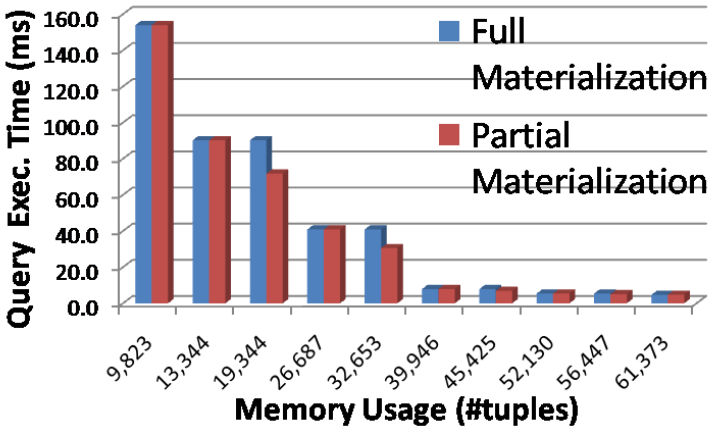


Figure 4.15: **Memory Usage vs. Computation tradeoff for the DyMon scenario using the DP algorithm with different materialization options.**

The distributions obtained in the DyMon application are much less skewed compared to the NIDS dataset. There is no small subset of the overall joint distribution that has significantly high probability. Hence, partial materialization plans perform similarly to the full materialization plans. The benefit of partial materialization in this case is its ability to offer an increased range of plans using different levels of memory.

Query Range and Memory Budget: In this experiment, we show results using the DP algorithm on the described range query for different range values and memory budgets. In Figure 4.16, the average execution time increases linearly with the query range for the case of DP with no materialization option. When we use 2X or 3X the memory required by the base relations for materialization of the intermediate relations, we can reduce the computation time for different range values. Note that the size of memory required by the base relations is independent of the query range.

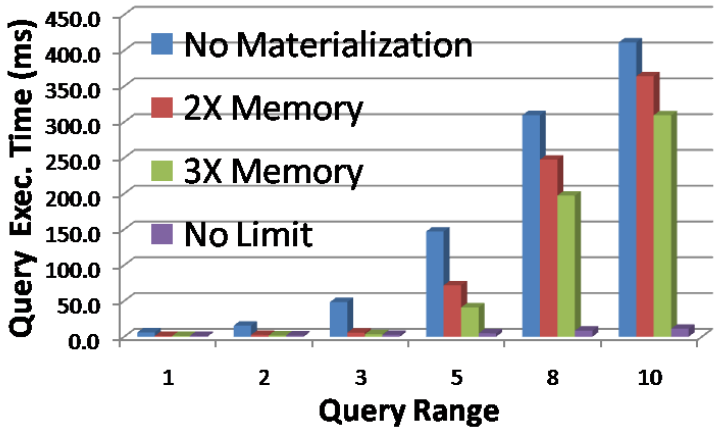


Figure 4.16: **Average query execution times for increasing query ranges and under different memory budgets.**

Finally, if we do not place a limit on the memory available for materialization, we see that query execution times increase very slowly with increasing query ranges. The memory use with the 'No Limit' option is at most 5 times the size of the base relations in all cases.

Top-k Queries: We now consider top-k queries (§4.4.3) using an example query that predicts the k-most likely CPU sequences in a future time interval based on the 2 most recent observations. In Figure 4.17, we show the average execution times for varying prediction ranges and k values. The exponential trendline labeled 'Full Joint' represents the execution time of the naive method that calculates the full joint distribution of CPU sequences to compute the top-k values. The other results reveal the linear behavior of the query execution times with the discussed optimization for top-k queries. All the results are based on the DP algorithm without materialization option to focus only on the effects of the top-k optimization.

For query ranges greater than 5, the joint distribution is larger than the 1.5GB memory available in our JVM so there are no Full Joint results for those cases. For the query range of 5 time units, the Full Joint method is 20 times slower than the top-k optimization method. While the size of the joint distribution grows exponentially with the query range, the size of memory required for the top-k optimization method is a constant factor of k.

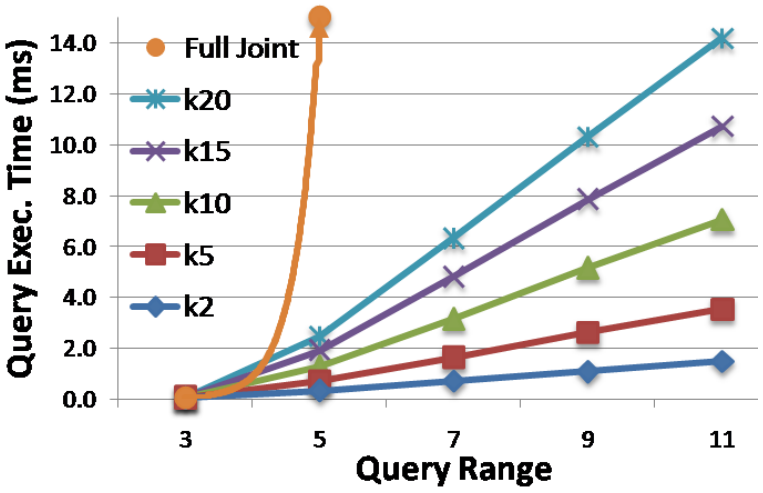


Figure 4.17: **Average execution times for the top-k queries predicting the k most likely CPU sequences for different range and k values.**

4.5.4 White-box Inference vs Off-Database Inference

A key premise of this work is that an in-database approach for inference can offer substantial benefits over the off-database approaches, which we quantitatively demonstrate in this section. As a representative off-database system, we use the open-source "Weka" software (ver. 3.6.1) [95], which contains a collection of common machine learning algorithms for data mining. We also developed specialized, isolated Java implementations for specific queries to serve as a point of comparison.

For the access type prediction query of the NIDS application (§4.5.2), we obtained 4.5 ms execution time (per incoming tuple) with Weka's BayesNet classifier. Recall from Figure 4.13 that our system has execution times of 6.52 ms (no materialization) and 2.66 ms (materialization) for the same task. Thus, our system is competitive or better (with optimization) than Weka software for this relatively simple task. Given that our system is not mature and relatively under-optimized leads us to believe we can improve upon these numbers substantially. Our specialized Java implementation of the VE algorithm achieved 0.11 ms execution time for this task, revealing that both our system and Weka suffer from various overheads (e.g., function calls, copying of intermediate results) that seem

to dominate the cost of inference for this task in which the query is straightforward (e.g., no marginalization) and the total size of the base distributions is small.

We now consider the range-based CPU prediction query from the DyMon application (§4.5.3). As Weka does not support DBNs, there was no straightforward way to use it for this task. Our initial custom Java implementation performed so poorly that we had to augment it with index support to get practical numbers. With the indexed Java implementation, we obtained execution times of 1.50 ms, 2.45 ms, 29.44 ms and 119.46 ms for query ranges of 1, 2, 3 and 5, respectively. These results demonstrate that indexing, which comes with the database approach, is crucial due to the size of the base distributions. For the same task, our system achieved (§4.5.3) 6.01 ms, 15.42 ms, 48.35 ms and 146.99 ms (no materialization) and 1.15 ms, 1.82 ms, 3.44 ms and 40.884 ms (materialization). In this case, the advantage of materialization is more than enough to compensate for the overhead of the database, achieving improvements of 30%-290% over the Java implementation. In summary, the database approach is not only more general than a specialized "roll-your-own" implementation but is also the clear performance winner with increasing data size and query complexity.

Chapter 5

The Black-box Approach for Predictive Databases

5.1 Black-box Support of Prediction Models

In the black-box approach to database support for predictive functionality, existing implementations of prediction methods, in the form of libraries or standalone applications, are directly integrated (or linked) to database systems and used as isolated components implementing prediction. With the plethora of available machine learning libraries and standalone implementations, the black-box method offers an easy and efficient way of integrating already tested and optimized prediction methods into the database systems.

In this approach, the prediction methods are presented to the user in the form of an extensible set of SQL functions and stored procedures with well-defined semantics. This way, predictive methods are easily used within SQL queries. Similarly, training and test data sets are also specified using SQL. The use of declarative queries for the specification of predictive operations and the relevant training and test data sets offers an easy and flexible method of expressing prediction tasks over complex data sets (e.g., computing aggregates over groups). 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, and cook the raw data into a form that is more amenable for effective learning.

In our study, we present a prediction interface/API consisting of a set of function templates for expressing generic prediction operations such as training and testing of prediction models. Prediction methods are registered into our system by providing implementations of the prediction interface. In this way, new prediction methods can be easily integrated into the system and various prediction models can be built and tested for the same prediction task. As discussed before, it is hard to know in advance which prediction models are the 'best' for a given scenario, so it is common practice to build and test multiple models. The use of a common prediction interface simplifies this practice by decoupling the underlying implementation from the high-level prediction task.

We note that the code executed by a predictive database function is beyond the control of the database system. Furthermore, the function itself could be an implementation of a prediction method or it could be a call (or a wrapper) to an external library. As a result, the database system cannot control or monitor the execution of a black-box prediction method. Therefore, unlike the white-box approach, in the black-box approach it is not possible to optimize the internal operations of prediction methods. However, we optimize our system to minimize the cost of data access and processing performed by black-box models. Our iterator (and block) based data access functionality and optimization hooks available in the prediction API are designed to optimize the performance of black-box methods.

Our black-box model support techniques including the prediction model API, model management functionality and the SQL prediction interface have been implemented in the Longview prototype built on top of the PostgreSQL [83] database system. In the rest of this chapter, we describe the functionality and details of our black-box techniques. We evaluate the performance and usability of the system in Chapter 6 using a database query performance prediction application as our case study.

5.2 Prediction Interface

In this section, we describe our prediction interface/API which consists of basic functions that each prediction model must implement for integration into our system. The functions correspond to the basic training and testing operations, which are the high-level generic tasks supported by all prediction models. For example, using this interface the database system can build a linear regression model over a training data set or perform prediction with a support vector machine instance. As such, the prediction interface provides interoperability between the various prediction models and the database system. The interface also decouples the implementation of a prediction model from its functionality.

The main functions included in our prediction interface are the following:

```
void* build(int ntuples, int nfeats, double **feat_list,
    int ntattrs, double** target_list, char* model_params)

double* predict(void* model_ptr, int ntuples, double **feat_list)

char* serialize(void* model_ptr)

void* deserialize(char* model_desc)
```

In Table 5.1 we describe the arguments used by the functions of the prediction interface.

Argument	Description
ntuples	number of data points.
nfeats	number of features (i.e., observed attributes) in the model.
feat_list	the values of the feature(s).
ntattrs	number of target (i.e., prediction) attributes.
target_list	the target attribute(s) values.
model_params	model specific training parameters.
model_ptr	pointer to the model instance.
model_desc	(serialized) description of a model instance.

Table 5.1: **List of Prediction Interface Arguments**

The *build* function is used to train a prediction model based on the given feature and target values. The feature and target values passed as arguments to the build function

(when applicable) are stored using a column-based representation in memory (instead of a row-based representation). The column-style representation enables Longview to perform: (i) dynamic management of training data (e.g., non-predictive features as determined by correlation metrics can easily be discarded and predictive features can be efficiently added) and (ii) efficient projection/selection operations on the training data (e.g., for use in feature selection). In addition, prediction models often have specific training parameters that can significantly affect their operations and accuracy. Therefore, we also provide an argument for specifying the training parameters for a prediction model. The result of the build function is a model-specific set of data structures and metadata, which is represented by a model pointer in the database system.

Using the *predict* function, a previously built prediction model can be used for prediction of a target attribute based on the given feature values. In the function prototype, we chose double as the feature and target types for simplicity; however, in practice we use pointers with type information to support arbitrary combinations of feature and target types. The predict function does not require the nfeats and ntattrs arguments, as they are already accessible as part of the model instance.

We require the black-box implementations to provide serialization and deserialization functions for their models. This functionality is used by our database system to store and re-use previously trained prediction models. We observed that serialization functionality is supported by most of the existing prediction model implementations and is already available for our use.

The build function of the described prediction interface uses pointers to pass the data to the prediction models. This approach has the advantage that no extra copying of the data is required. As such, prediction models that can directly operate on the given pointer-based data structures will have no data passing overhead. In addition, we can build multiple such models (e.g., with different parameters) using the same copy of the training data (possibly in parallel).

However, some black-box methods internally copy the given data to special data structures. In such implementations, using an iterator or block -based data transfer method could be preferable for minimizing the peak memory usage. Otherwise, two or more copies of the training data could simultaneously exist in memory, severely limiting the scalability of the system. Iterator and block -based versions of the build function are shown below.

```
void* build_itr(int nfeats, double* (*next_feat)(), int ntattrs,
    double* (*next_target)(), char* model_params)

void* build_blk(int nfeats, double** (*next_feat)(int, int*), int
    ntattrs, double** (*next_target)(int,int*), char* model_params)
```

Both functions rely on function pointers for data access in training. The iterator-based version can be used to access data points one-by-one whereas the block-based function provides access to n tuples at a time (where n is an argument of the function).

The iterator/block based prediction interface also targets incremental and online black-box models. Such prediction models can be updated with new data items or built incrementally by iterating over the training data. Using the iterator/block based prediction interface with online/incremental models significantly reduces the memory requirements and also enables scaling to data sets larger than the available memory. While learning on large data sets (larger than the size of available memory) is a very important problem, in Longview we mostly focus on the use and integration of the available black-box methods. We briefly discuss the large-scale learning problem in the Related Works and Conclusion chapters. In the rest of this chapter, we describe the details of model representation and management techniques in the context of database systems.

5.3 Model Management and Prediction

In this section, we describe how we manage prediction model types and perform prediction using black-box models in Longview. As a design philosophy, whenever possible we build

on the existing extension mechanisms (e.g., rules, triggers and views). This way, we try to generalize our approach and make it easier to use in modern database systems. In addition, the use of existing mechanisms reduces the implementation complexity by minimizing the amount of changes required on a target database system (e.g., in our case PostgreSQL). Finally, we also discuss examples of some of the high-level prediction functionality that we can build over the described predictive database architecture.

5.3.1 Model Registration

Longview represents models and their metadata using database catalogs (i.e., a set of system relations). The catalog schema is given in Figure 5.1. When a new prediction model type (e.g., SVM, linear regression) is integrated into Longview, an entry representing the model type and containing model metadata information is created in the `pred_model_types` relation and in other relations.

The pred_model_types relation contains the unique names of the model types along with tags/user descriptions. The information on the implementation of the prediction interface for a model type is stored in the 'library' field. More specifically, the library field, at the very least, contains the names of the functions that implement the prediction interface for a model.

When a new model type is added to Longview, our system automatically creates a set of SQL functions and stored procedures that allows the newly added model to be accessed within SQL queries. More specifically, as the prediction interface provides a means of communication between the database system and the model implementations, these SQL functions enable the user to access the predictive functionality through the database system.

We list the specifications for these automatically generated SQL functions in Table 5.2. The details of these operations are discussed in Section 5.3.2.

The definitions for the various parameters of a prediction model type (e.g., training parameters) are stored in the `pred_model_params` relation. Each model type can have a number of parameters for adjusting its operation. In addition, in our system we can

Function	Arguments	Description
build	model id	specifies the model instance
	training query	query computing the feature and target values
	model parameters	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 5.2: **Prediction Model SQL Interface**

also express default values and domain information for the model parameters (using the `pred_model_params` and `pred_attr_types` relations). The validity of the input parameter values can also be enforced with database integrity constraints.

5.3.2 Model Representation, Building and Testing

All instances of prediction models are listed in the `pred_models` relation. This relation contains a unique auto-generated id for each model, a reference to a model type and a serialization field. The serialization field is used for storing the type-specific representation of a prediction model. For instance, for a linear regression model, the serialization field would contain the coefficients used in the model and for a support vector machine we would have a list of the support vectors together with a few other parameters. The particular representation used in the serialization field depends on the model implementation (i.e., the serialization and deserialization functions provided by the black-box model) and is usually based on data-interchange formats such as XML and JSON.

The attributes (features and the target attributes) of a prediction model are stored in the `pred_model_attrs` relation. Each attribute is represented with a name and id, a type (e.g., double) and a role (i.e., feature, target). When creating a prediction model, we first create an entry in the pred_models relation and specify the attributes to be used in the model. We provide the following `create_model` stored procedure to automate this model specification process:

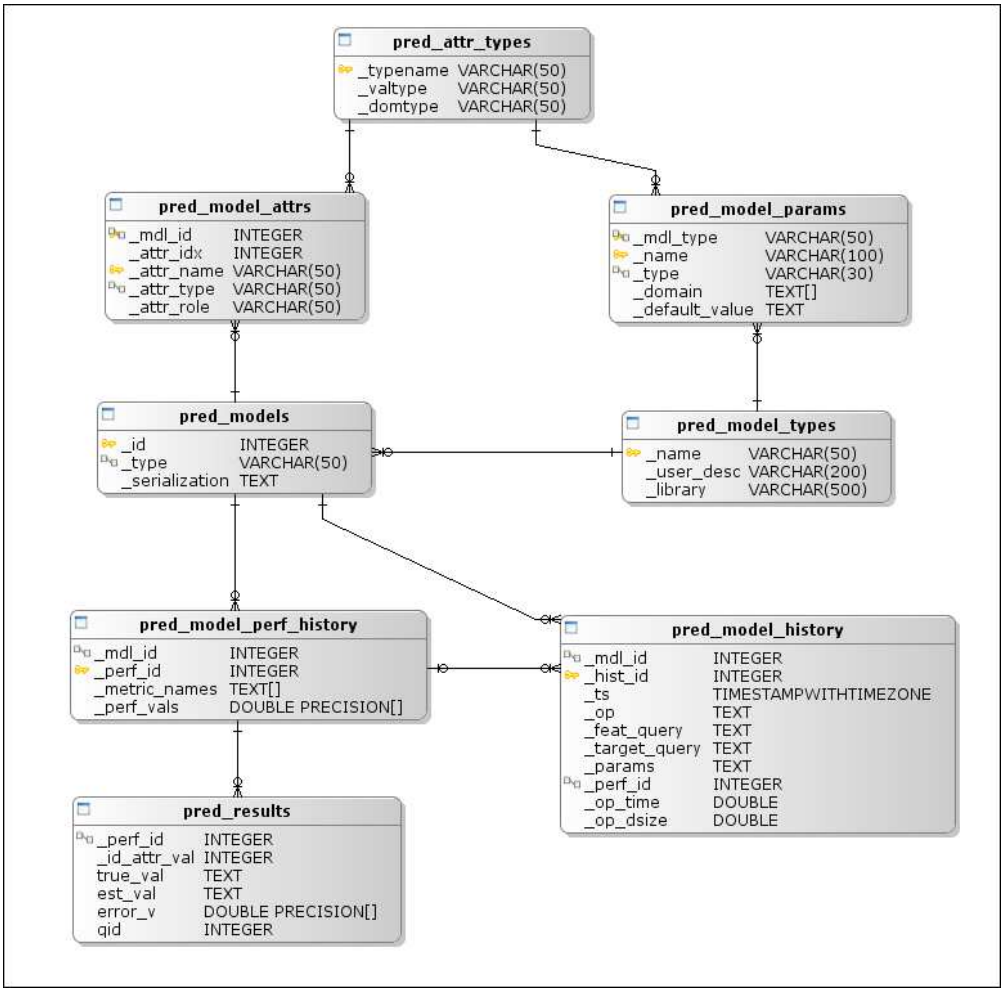


Figure 5.1: **The design schema for the database catalog used for representing and managing the black-box prediction models.**

```
integer create_model ( model_type text, model_schema text)
```

The `model_type` attribute specifies the type of the prediction model and must be one of the model types listed in pred_model_types. On the other hand, the `model_schema` attribute contains a description of the observable and target attributes. Given a model type and a valid attribute schema, the create_model function registers the model definition to the system and returns the unique model id generated for the newly created model instance.

Once a model definition is created, the training of the model can be performed using the build function of the SQL prediction interface (see Table 5.2) with the automatically assigned model_id. As mentioned before, the SQL functions listed in Table 5.2 have dynamic implementations, as wrappers around the prediction interface. As such, Longview calls the build function of the respective model type to perform the actual training operation. In addition, a model can be trained multiple times (i.e., updated). After each training pass, the resulting model is stored in the catalogs by updating the serialization field.

Prediction with a model instance is performed using the test and predict functions listed in Table 5.2. Observe that, both of these functions first need to deserialize the model by reading its serialized form from the pred_models relation and calling the appropriate deserialization function. The output of the test and predict functions are relations, containing the prediction results. The output of the test function also includes the true values and computed accuracy values for each prediction value produced in the testing operation. In addition, the output relations can be used as data sources in more complex SQL queries. An interesting use case is the nested prediction queries, where the prediction results of a sub-query are fed as features to the outer prediction query.

5.3.3 Performance Tracking

Another functionality that we provide to predictive applications in Longview is the tracking of the training and testing operations performed with prediction models. For each prediction operation, we record an entry in the `pred_model_history` relation including the used

model, the type of the operation (i.e., build or test) and other given arguments. For testing operations, computed prediction accuracy metrics and the average processing time for prediction per data point are also stored (in the `pred_model_perf_history` relation). This way, applications can monitor the evolution of models, track the used training data sets and the performance values on test data sets.

Optionally, we also store the individual predictions produced by the model, and the true values (if provided) in the `pred_results` table. While a costly operation, the ability to analyze predictions on a per-point basis can in some cases be highly useful; e.g., comparing prediction results from multiple models.

5.3.4 Automated Model Building

To explain the capabilities and extend the applicability of Longview, in this section we describe our implementation of an intelligent model building algorithm (based on well-known machine learning algorithms), that builds simple and accurate prediction models for given data sets. In order to build an accurate prediction model based on a given data set, we have to find out (i) which model type is the best for the data at hand and (ii) which set of features are to be used for accurate prediction of the target attribute(s).

First, we note that in general it is not possible to predict which model type is the 'best fit' for a data set without building and testing multiple models. Therefore, any automated algorithm for building accurate prediction models needs to train and test multiple models. The second problem is the problem of feature selection; i.e., choosing the set of features to be used in a prediction model. Most of the feature selection algorithms in the literature, such as the forward and backward selection algorithms [95], are heuristic algorithms that search the attribute space using rank-based methods. The rank of a feature is an indicator of its predictive value for the target attribute.

Our model building algorithm is in the class of forward feature selection algorithms. However, our algorithm performs both model selection and the feature selection processes simultaneously. The algorithm starts by building a set of initial models; a model per each

combination of a model type and a single feature is trained. Each created model is inserted into a bounded priority queue based on its estimated accuracy value. The maximum size of the priority queue is a system parameter and can affect both the run-time of the algorithm and accuracy of the produced models. At each step of the algorithm, the next best model from the priority queue is popped, and a new set of models based on the popped model are built and inserted into the queue. The new models are simple extensions of the old model; a new feature is added to their feature list. The newly added feature must be ranked lower than all the features already used by the old model.

Notice that the described algorithm is essentially an accuracy-driven search process that builds many models (of different configurations) using the same training data. Therefore, models which provide build functions (see prediction interface in Section 5.2) that directly operate on the provided training data will have much less overhead. Our column-based organization of the in-memory training data also enables efficient selective access to the subsets of features required by a model. In addition, for the model types that are capable of incremental learning, we also provide hooks (in the form of additional arguments to the build functions) that they can use to incrementally build new models based on older models by adding/removing features.

Finally, a second parameter, adjusts the number of speculative executions. When the speculative execution parameter is set to 0, all the new models with lower accuracy values compared to their parent models, are immediately discarded. As such, the speculative execution parameter specifies the number of times a new model with a lower accuracy value than its parent is allowed to survive the elimination from the priority queue.

Chapter 6

Query Performance Prediction

In this chapter, we study an important and challenging system-facing predictive application: `Query Performance Prediction` (QPP). QPP is the problem of predicting the execution times (or other performance metrics such as disk I/O and memory usage) of database queries without running them. Modern database systems can greatly benefit from QPP. For example, resource managers can utilize QPP to perform workload allocation such that interactive behavior is achieved or specific QoS targets are met. Optimizers can choose among alternative plans based-on expected execution latency instead of total work incurred.

Accurate QPP is important but also challenging: database systems are becoming increasingly complex, with several database and operating system components interacting in sophisticated and often unexpected ways. The heterogeneity of the underlying hardware platforms adds to this complexity by making it more difficult to quantify the CPU and I/O costs. Analytical cost models predominantly used by the current generation of query optimizers cannot capture these interactions and complexity; in fact, they are not designed to do so. While they do a good job of comparing the costs of alternative query plans of a given query, they are poor predictors of plan execution latency. The use of analytical cost models for QPP is discussed in detail in our experiments (Section 6.3).

In this chapter, we utilize learning-based modeling and prediction techniques to tackle QPP for analytical workloads using the Longview system and the black-box model support

techniques described in Chapter 5. Prior work reported evidence that predictive techniques can be used effectively for QPP, at least in constrained settings [41, 101, 2, 3]. Our study substantially improves and generalizes these results in a number of new directions, arguing that learning-based techniques tailored to database query execution are generally applicable to and can be highly effective for QPP. In addition, by implementing the QPP application entirely within Longview we demonstrate the usability and effectiveness of our modeling framework and show an example of how predictive modeling can be leveraged for building highly useful functionality in real systems.

One of our key contributions is to show that queries can be modeled at different granularities, each offering different tradeoffs involving predictive accuracy and generality. If a representative workload is available for training purposes, we can make highly accurate predictions using coarse-grained, plan-level models [41]. Such models, however, do not generalize well, performing poorly for unseen or changing workloads. For 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 propose a hybrid approach that selectively composes plan- and operator-level models to achieve high accuracy without sacrificing generality.

Finally, while we study the utility of learning-based models for query execution latency as the performance metric of interest, the proposed techniques are general, and thus can be used in the prediction of other metrics such as throughput. We should also note that in this thesis we do not consider QPP in the presence of concurrent execution, which is an important and challenging problem to address, but is outside the scope of this study.

6.1 Modeling Query Executions

As is usual in most learning approaches, all of our modeling techniques consist of two main phases: training and testing. The high-level operations involved in these phases are explained in Figure 6.1. In the training phase, prediction models are derived from a training

data set that contains previously executed queries (i.e., training workload) and the observed performance values (i.e., query execution times). In this phase, queries are represented as a set of features and corresponding performance values. The goal in training is to create an accurate and concise operational summary of the mapping between the feature values and the observed performance data points. The prediction models are then used to predict the performance of unforeseen queries in the test phase. In more complex QPP methods, the training and testing phases can be performed continuously for improved accuracy and adaptivity.

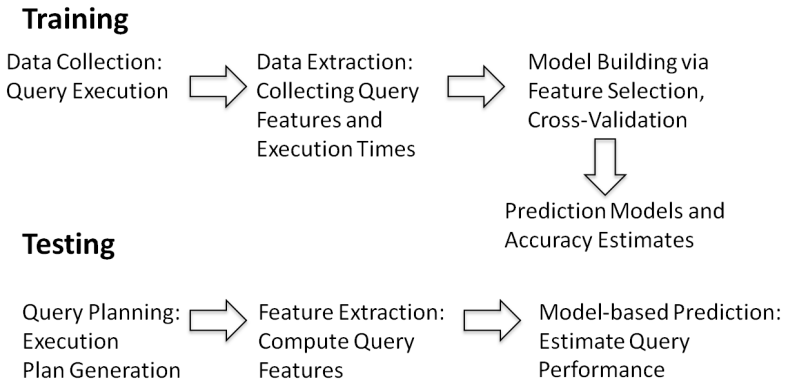


Figure 6.1: **Statistical Modeling Approach to Query Performance Prediction.**

Our approach to QPP relies on models that use only static, compile-time features, which allow us to produce predictions before the execution of queries. There are several static information sources, such as the query text and execution plans, from which query features can be extracted prior to execution. In this study, we use features that can be obtained from the information provided by the query optimizer. Many database systems provide optimizer calls that expose query-plan information and statistical estimates such as the optimized query-plan structure and operator selectivities (for example, `EXPLAIN` in PostgreSQL and `EXPLAIN PLAN` in Oracle).

In this chapter, we show that it is possible to create accurate models at varying granularities for query performance prediction. As in [41], one coarse modeling method is to

create a single, plan-level prediction model that utilizes query plan features for modeling the execution times of queries. We discuss this approach in Section 6.1.1. A finer grained approach would be to model each operator type separately and use them collectively through selective composition to model entire query plans. We describe this method in Section 6.1.2 and compare the relative advantages and drawbacks of the two approaches in Section 6.1.3. Next, in Section 6.1.4, we introduce a "hybrid" modeling approach that combines the fine and coarse grained modeling methods to form a highly accurate and general QPP approach.

6.1.1 Plan-level Modeling

In the plan-level modeling approach, the performance of a query is predicted using a single prediction model. We use the features presented in Table 6.1 for building plan-level models. This set of features contains query optimizer estimates such as operator cardinalities and plan execution costs together with the occurrence count of each operator type in the query plan.

Feature Name	Description
p_tot_cost	Estimated total plan cost
p_st_cost	Estimated plan start cost
p_rows	Estimated number of output tuples
p_width	Estimated average size of an output tuple (in bytes)
op_count	Number of query operators in the plan
row_count	Estimated total number of tuples input and output to/from each operator
byte_count	Estimated total size (in bytes) of all tuples input and output
<operator_name>_cnt	The number of <operator_name> operators in the query
<operator_name>_rows	The total number of tuples output from <operator_name> operators

Table 6.1: **Features for plan-level models. p_st_cost refers to the cost of query execution until the first output tuple. <operator_name> refers to the query operators such as *Limit, Materialize* and *Sort*.**

As mentioned in Section 2, in general we need to address two main challenges when

using model-based learning techniques. The first problem, *model selection*, is the process of picking the right prediction model for the given task and data set. As discussed before, it is not possible in general to identify the most accurate prediction model without training and testing multiple models. In our study, we show results with two types of prediction models for plan-level modeling: a regression variant of Support Vector Machines (SVMs) [21] and Kernel Canonical Correlation Analysis (KCCA) [11, 5] (also used in [41] for QPP). Both model types provided high accuracy in our experiments. We note that all of the approaches we present here are model-agnostic and can readily work with different model types.

The second problem, *feature selection*, deals with the issue of choosing the most predictive features for modeling the target variable(s) from the available set of features. Feature selection does not need to be performed for all types of prediction models. For instance, in our case we perform feature selection for SVMs but not for KCCA as it performs dimensionality reduction as part of its operation. However, for many model types feature selection is an important problem. In our experiments, we frequently observed that SVM models using the full set of features given in Table 6.1 performed less accurately than models with smaller number of features. For building SVM-based models, we use the intelligent model building algorithm described in Section 5.3.4. This algorithm starts by building models using a small number of features, and iteratively creates more complex and accurate models by adding features in order of correlation with the target variable (i.e., query execution time).

Once a plan-level prediction model is built and stored (i.e., materialized), it can be used to estimate the performance of new incoming queries based on the query-plan feature values that can be obtained from the query optimizer without executing the query.

6.1.2 Operator-level Modeling

We now introduce a finer-grained approach to QPP: operator-level modeling. Unlike the plan-level approach, which uses a single prediction model, the operator-level technique relies on a collection of models that are selectively composed for end-to-end query performance prediction. In the operator-level modeling approach, two separate prediction models are

built for each query operator type:

A start-time model is used for estimating the time spent during the execution of an operator (and in the sub-query plan rooted at this operator) until it produces its first output tuple. This model captures the (non-)blocking behavior of individual operators and their interaction with pipelined query execution.

A run-time model is used for modeling the total execution time of query operators (and the sub-plans rooted at these operators). Therefore, the run-time estimate of the root operator of a given query plan is the estimated execution time for the corresponding query.

To illustrate the semantics and the use of the start-time model, we consider the *Materialize* operator, which materializes its input tuples either to disk or memory. Assume that in a query tree, the Materialize operator is the inner child operator of a *Nested Loop* join. Although the materialization operation is performed only once, the join operator may scan the materialized relation multiple times. In this case, the start-time of the Materialize operator would correspond to the actual materialization operation, whereas the run-time would represent the total execution time for the materialization and scan operations. In this manner, the parent Nested Loop operator can use the start-time and run-time estimates to form an accurate model of its own execution time. This technique also allows us to transparently and automatically capture the cumulative effects of blocking operations and other operational semantics on the execution time.

We used a single, fixed collection of features to create models for each query operator. The complete list of features is given in Table 6.2. This list includes a generic set of features that are applicable to almost all query operators. They can also be easily acquired from most, if not all, existing DBMSs. As in the case of plan-level modeling approach, we use the intelligent model building algorithm from Section 5.3.4 to build accurate prediction models with the relevant set of features. We used multiple linear regression (MLR) models for modeling the query operators. In addition to performing accurately in our experiments,

similar to analytic cost models MLR models are intuitive and easily interpretable.

Feature Name	Description
np	Estimated I/O (in number of pages)
nt	Estimated number of output tuples
nt1	Estimated number of input tuples (from left child operator)
nt2	Estimated number of input tuples (from left right operator)
sel	Estimated operator selectivity
st1	Start-time of left child operator
rt1	Run-time of left child operator
st2	Start-time of right child operator
rt2	Run-time of right child operator

Table 6.2: **Features for the operator-level models. Start time refers to time spent in query execution until the first output tuple is produced.**

The individual operator models are collectively used to estimate the execution latency of a given query by selectively composing them in a hierarchical manner akin to how optimizers derive query costs from the costs of individual operators. That is, by appropriately connecting the inputs and outputs of prediction models following the structure of query plans, it is possible to produce predictors for arbitrary queries.

In Figure 6.2, we illustrate this process for a simple query plan consisting of three operators. The performance prediction operation works in a bottom-up manner: each query operator uses its prediction models and feature values to produce its start-time and run-time estimates. The estimates produced by an operator are then fed to the parent operator, which uses them for its own performance prediction.

6.1.3 Plan- versus Operator-level Modeling

The premise of the plan-level approach is that queries with similar feature vectors will have similar query plans and plan statistics, and therefore are likely to exhibit similar behavior and performance. Such an approach is specifically targeted to `static workload` scenarios where the queries in the training and test phases have similar execution plans (e.g., generated from the same query templates or from the same user program).

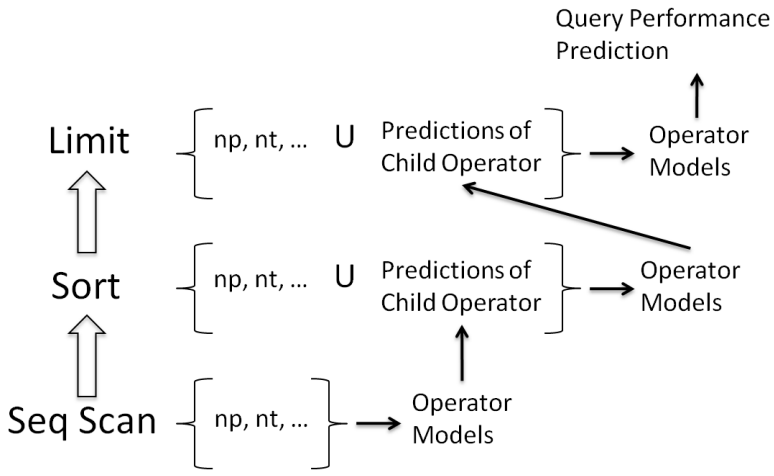


Figure 6.2: **Operator-level query performance prediction: operator models use operator-level features together with the predictions of child operators for performance prediction.**

Furthermore, this approach is based on the correlation of the query execution plans and statistics with the query execution times. This correlation is used directly in mapping query-plan based features to execution performance. The high-level modeling approach used in this case therefore offers the ability to capture the cumulative effects of a set of hidden lower level factors, such as operator interactions during query processing, on the query execution times with a single, low complexity model.

The plan-level approach, however, is prone to failure in some common real-world scenarios. A significant problem exists in the case of `dynamic query workloads` where queries with unforeseen execution plans are frequently observed. Even worse, there can also be problems in static query workloads. As the feature values only represent a limited view of a query plan and its execution, it is possible that different queries can be mapped to very similar feature values and therefore be inaccurately modeled. While it is unlikely for completely different queries to be mapped to identical features, similar queries can sometimes have different execution performance. For instance, increasing the number of time consuming aggregate operations in a query will not significantly change its feature vector, but may highly increase its execution time. Adding more features (e.g., number of aggregates and

constraints) to the model would alleviate such issues, however, each added feature would also increase the size of the required training data.

By using multiple prediction models collectively in a hierarchical manner, the operator-level prediction method is able to produce performance predictions for arbitrary queries. Therefore, it is a more general approach compared to the plan-level method and has the potential to be more effective for dynamic query workloads where unforeseen query plan structures are common.

On the downside, the operator-level prediction method may suffer from drawbacks similar to those that affect analytical cost estimation methods (as both methods rely on low-level operator-based models). A key problem is that the prediction errors in the lower levels of a query plan are propagated to the upper levels and may significantly degrade the end prediction accuracy.

Another potential problem is that the concurrent use of multiple resources such as CPU and disk may not be correctly reflected in the operator-level (or the analytical) models. For instance, a query could be simply performing an aggregate computation on the rows of a table that it sequentially scans from the disk. If the per-tuple processing takes less time than reading a tuple from the disk, then the query execution time is approximately the same as the sequential scan time. However, if the processing of a tuple takes longer than reading it from the disk, then the execution time will be closer to the processing time. As such, the interactions of the query execution system and the underlying hardware/software platforms can get quite complex. In such cases, simple operator-level modeling approaches may fall short of accurately representing this sophisticated behavior. Therefore, in static query workloads where training and testing queries have similar plan structures we expect the high-level information available in the plan-level approach to result in more accurate predictions.

6.1.4 Hybrid Modeling

In hybrid query performance prediction, we combine the operator- and plan- level modeling techniques to obtain an accurate and generally applicable QPP solution. As discussed, this is a general solution that works for both static and dynamic workloads. We note that as long as the predictive accuracy is acceptable, operator-level modeling is effective. On the other hand for queries with low operator-level prediction accuracy, we learn plan-level models for the inaccurately modeled query sub-plans and compose both types of models to predict the performance of the entire plan. We argue, and later also experimentally demonstrate that this hybrid solution indeed combines the relative benefits of the operator-level and plan-level approaches by not only retaining the generality of the former but also yielding predictive accuracy values comparable or much better than those of the latter.

Hybrid QPP Example: To illustrate the hybrid method, we consider the performance prediction of an example TPC-H [93] query (generated from TPC-H template-13), whose execution plan is given in Figure 6.3. This plan is obtained from a 10GB TPC-H database installed on PostgreSQL. As we describe in detail in the Experiments section, we build operator-level models on a training data set consisting of example TPC-H query executions. When we use the operator-level models for performance prediction in this example query, we obtain a prediction error (i.e., $|$*true value* - *estimate*$|$ / *true value*) of 114%. Upon analysis of the individual prediction errors for each operator in the query plan, we realized that the sub-plan rooted at the *Materialize* operator (the highlighted sub-plan in the figure) is the root cause of the prediction errors in the upper level query operators. The operator-level model based prediction error for the materialization sub-plan is 97%.

In the hybrid approach, we build a separate plan-level model for the highlighted sub-plan. The model is trained using the occurrences of this sub-plan in the training data. The hybrid method uses the plan-level model to directly predict the execution performance of the materialization sub-plan, while the rest of the prediction operations is unchanged, i.e., performed with the operator-level models. The prediction errors obtained with the hybrid

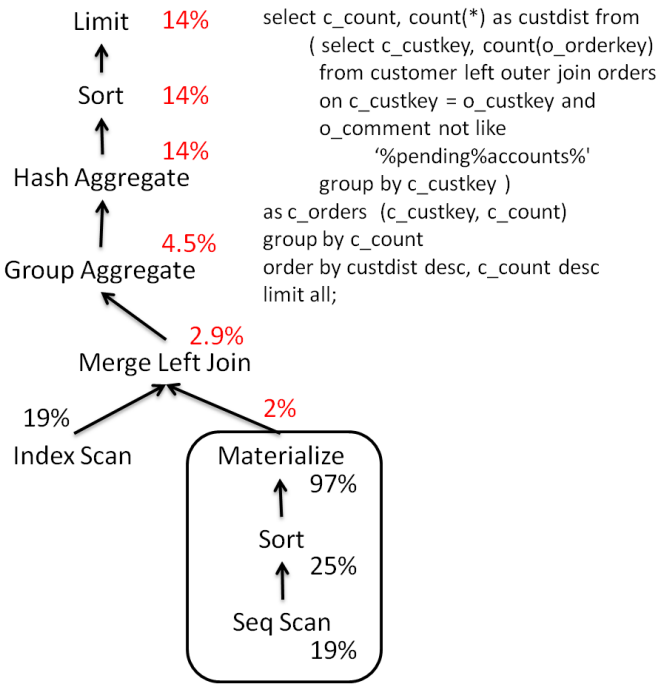


Figure 6.3: **Hybrid QPP example: plan-level prediction is used for the highlighted sub-plan together with operator-level prediction for the rest of the operators to produce the end query performance prediction.**

approach are shown with the red values in the figure. The new overall prediction error for this example query drops down to 14%.

Given a training data set consisting of example query executions, the goal of the hybrid method is to accurately model the performance of all queries in the data set using operator-level models together with a minimal number of plan-level models. In this way, we maximize the applicability of the operator-level models in query performance prediction and maintain high prediction accuracy with the integration of plan-level models.

The hybrid performance prediction method is described in Algorithm 2. The algorithm starts by building prediction models for each query operator based on the provided training data. The accuracy of operator-level prediction is then estimated by application on the training data (e.g., either through cross-validation or holdout test data). Next, the algorithm tries to increase the QPP accuracy by building and testing plan-level models.

Each plan-level model is used for directly modeling the performance of a specific query plan (or sub-plan). In a query plan with N operators, there is a maximum of $N-1$ sub-plans (e.g., in a chain of operators) for plan-level modeling. Then a training data set with M queries can have $O(MN)$ candidate sub-plans for modeling.

In theory, we could build and test plan-level models for each distinct sub-plan (with at least a minimum number of occurrences in the training data set) and try to find a minimal subset of these models for which the prediction accuracy is sufficiently high. However, this would require a large amount of time since (i) we need to build and test models for all candidate sub-plans, and (ii) the prediction accuracy of each subset of models (in increasing sizes) needs to be separately estimated with testing.

Instead, we propose heuristics that iteratively build a collection of plan-level models to maximize the expected predictive accuracy. In each iteration, a new plan-level model is built, tested and added to the model set, if it improves the overall prediction accuracy (by more than a threshold value, ϵ). The models are chosen, built and tested according to *plan ordering strategies*. We consider the following strategies for the hybrid approach:

Size-based: order the plans by size (in increasing *number of operators*).

The size-based strategy considers generating models for smaller plans before larger ones. This strategy is based on the fact that smaller plans occur more frequently (since by definition all sub-plans of a large plan are at least as frequent) in any data set, and therefore models for smaller plans are more likely to appear in future unforeseen queries. In case of a tie involving two plans with the same size, the more frequent plan is given priority.

Frequency-based: order the plans in decreasing *occurrence frequency*.

The frequency-based strategy is similar to the size-based strategy except that it directly uses the occurrence count of a plan from the training data for ranking. In case the occurrence count is same for two plans, smaller plans are considered first. An important difference from the size-based strategy is that when a large plan has

Algorithm 2 Hybrid Model Building Algorithm

Input: *data* = example query executions
Input: *strategy* = plan selection strategy
Input: *target_accuracy* = target prediction accuracy
Output: *models* = prediction models
Output: *accuracy* = estimated prediction accuracy

```
1. models = build_operator_models(data)
2. [predictions, accuracy] = apply_models(data, models)
3. candidate_plans = get_plan_list(strategy, data, predictions)
4. while accuracy ≤ target_accuracy and not stop_condition() do
5.    plan = get_next(strategy, candidate_plans)
6.    plan_model = build_plan_model(data, plan)
7.    [predictions, new_accuracy] = apply_models(data, models ∪ plan_model)
8.    if new_accuracy − ϵ ≤ accuracy then
9.       candidate_plans.remove(plan)
10.   else
11.      models = models ∪ plan_model
12.      candidate_plans.update(predictions, plan_model)
13.      accuracy = new_accuracy
```

a high occurrence frequency, the frequency-based strategy will consider modeling its sub-plans sequentially before considering other plans.

Error-based: order the plans in decreasing value of *occurrence frequency* × *average prediction error*.

The error-based strategy considers plans with respect to their total prediction error across all queries in the training data. The assumption is that more accurate modeling of such high error plans will more rapidly reduce the overall prediction error.

In all of the above strategies, the plans for which (i) the average prediction accuracy with the existing models is already above a threshold, or (ii) the occurrence frequency is too low are not considered in model generation.

In order to create the list of candidate plans (i.e., `candidate_plans`) for modeling, we traverse the plans of all queries in the training data in a depth-first manner in function `get_plan_list`. During the traversal, this function builds a hash-based index using keys based on plan tree structures. In this way, all occurrences of a plan structure are hashed

to the same value and metrics required by the heuristic strategies such as the occurrence frequency and average prediction error can be easily computed.

When a new plan-level model is added to the set of chosen models (i.e., `models`), the candidate plan list needs to be updated with the new prediction errors and occurrence frequencies for all plans. The occurrence frequency of a plan p will change with the addition of a new model when the plan for the added model contains p as a sub-plan (since such occurrences of p are consumed by the newly added model).

We can efficiently identify the set of plans for which the prediction errors or the occurrence frequencies might change with the addition of a model as follows: In the hash-based index built by the get_plan_list function, we also store the identifiers for the corresponding queries (which own the plans). As such, when a new model is added, the only plans that need to be updated are the plans that can be applied to one or more of the queries that the newly added plan is also applicable.

Finally, in cases where the target accuracy is unachievable, a maximum number of iterations can be used as a stop condition to terminate the algorithm. Other variations for the stop condition, such as setting a maximum number of iterations without accuracy improvement, are also possible but not evaluated in this study.

6.2 Online Model Building

In dynamic query workloads where queries with unforeseen plan structures are present, the plan-level performance prediction method performs poorly due to lack of good training data. The operator-level and the hybrid prediction methods are designed to be much more applicable to unforeseen plan structures. In addition, the hybrid method will utilize its plan-level models as much as possible to provide accuracy levels much higher than those achievable through pure operator-level modeling.

The prediction accuracy of the hybrid approach in dynamic workload scenarios depends on the applicability of its plan-level models in future queries. As a case study, we analyze

the generated execution plans for the TPC-H query workload on a 10GB TPC-H database running on PostgreSQL. In Figure 6.4(b), we show the most common sub-plans within the execution plans of queries generated from the 14 TPC-H templates for which we could use operator-level prediction techniques in our experiments (See Experiments Section for more details.). Our key observations for this data set include:

(1) Smaller sub-plans are more common across the TPC-H query plans (see Figure 6.4(a)).

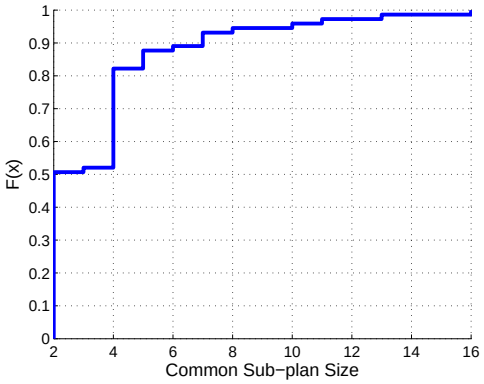
(2) The plans for the queries of each TPC-H template (except template-6) share common sub-plans with the plans of queries of at least one other TPC-H template (see Figure 6.4(c)).

These observations suggest that for the TPC-H workload: (i) it is possible to create plan-level models based on the execution plans for the queries of a TPC-H template and utilize them in the performance prediction of queries from other TPC-H templates, and (ii) the size-based plan ordering strategy discussed in Section 6.1.4 will likely achieve higher applicability compared to the other strategies in the dynamic workload case.

However, the hybrid approach may fail to increase the prediction accuracy for dynamic workloads in some cases. For example, the prediction errors for some unforeseen query plans may not originate from the common sub-plans, and as a result, plan-level models from the training data cannot reduce the error. In other cases, the common sub-plans could actually be the source of prediction errors, but the plan-ordering strategies may not necessarily choose to build plan-level models for them. For instance, some applicable plan-level models may be discarded, because they did not improve the prediction accuracy in training.

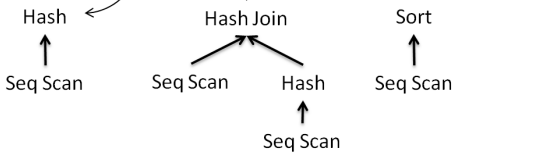
To address these issues, in the online modeling technique, we build new plan-level models for performance prediction at run-time upon the receipt of a query. We initially produce predictions with the set of existing models, and then update our results after new plan-level models are built for the received query.

Online model building is performed similarly to offline model building described for the hybrid method. However, in the online case, the set of candidate plans are generated based

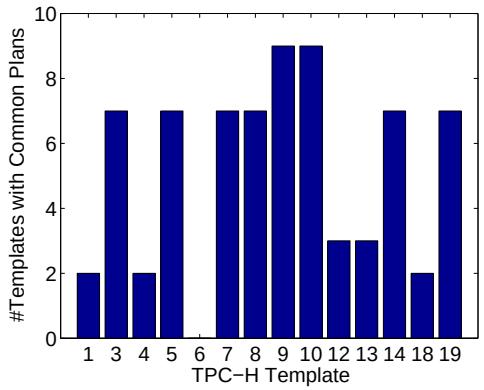


(a) CDF for common sub-plan sizes (#query operators)

Templates	3, 5, 7, 10, 8, 14, 19, 9	3, 5, 10, 8, 14, 19, 9	13, 12, 10, 9	3, 5, 8	3, 5, 8	1, 4, 18
Plan Size	2	4	2	5	7	2



(b) 6 most common sub-plans across queries of 14 TPC-H Templates



(c) #templates the queries of a TPC-H template shares common sub-plans with

Figure 6.4: **Analysis of common sub-plans for the execution plans of queries generated from 14 TPC-H Templates.**

on the set of sub-plans of the execution plan for the newly received query. The online building of plan-level models guarantee that if the execution plan for a test query has a common sub-plan (with high prediction error) with the queries in the training data, then a plan-level model will be built and used for its prediction (if a plan-level model with better estimated accuracy than the operator-level prediction method exists).

6.3 Experiments

6.3.1 Setup

In our experiments we use the TPC-H decision support benchmark [93] (implemented on top of PostgreSQL) to generate our query workload and Longview for the modeling and prediction tasks. The details are presented below.

Database Management System. We use an instrumented version of PostgreSQL 8.4.1. The instrumentation code monitored the described set of features and performance metrics from query executions; i.e., for each query, the execution plan, the optimizer estimates and the actual values of features as well as the performance metrics were logged.

Data sets and workload. We created 10GB and 1GB TPC-H databases according to the specification. The primary key indices as indicated in the TPC-H specification were created for both databases. We enforced a limit of one hour execution time (per query) to keep the overall experimentation duration under control. This resulted in 18 of the 22 TPC-H templates being used, as the remaining 4 templates always took longer than 1 hour to execute in the 10GB case.

There are approximately 55 queries from each template in both databases. With the 1GB database, all queries finish under an hour and the data set contains 1000 queries. On the other hand, with the 10GB database only 17 of the queries from template-9 finished within an hour, so we have 17 template-9 queries in the 10GB data set. Thus, the resulting 10GB data set we used contains 960 queries.

Hardware. Unless stated otherwise, all the queries were executed on a single commodity desktop with 4GB RAM running Linux kernel 2.6.28 and the database buffer pool size was set to 1GB (25% of the total RAM as the rule of thumb). All queries were executed sequentially with cold start (i.e., both filesystem and DB buffers were flushed before the start of each query).

Predictive models. In our experiments, we relied on the prediction functionality of Longview. All prediction operations were performed using Longview and its integrated

prediction models. For plan-level modeling, we used Support Vector Machines (available from the libsvm library [21]) with the nu-SVR kernel for support-vector based regression and the Kernel Canonical Correlation Analysis (KCCA) method. We implemented KCCA using GSL, the GNU Scientific library. On the other hand, for operator-level QPP we used Multiple Linear Regression based models available from the Shark machine learning library [56]. All models were integrated to the database as user defined functions or libraries. Our algorithms were implemented as a combination of C-based user-defined functions in PostgreSQL and as external applications written in C++ and Python. The feature selection algorithm, described in Section 5.3.4, was used to build accurate prediction models using a small number of features.

Metrics and validation. We use the mean relative error as our metric for prediction error:

$$\text{Mean Relative Error} = \frac{1}{N}\sum_{i=1}^{N}\frac{|actual_i - estimate_i|}{actual_i}$$

This metric is useful when we would like to minimize the relative prediction error in all queries *regardless* of their execution time. Non-relative error metrics such as the mean square error would be better for minimizing the absolute difference (or its square) in actual and predicted execution times. Other popular metrics include R^2 and predictive risk [41]. These metrics measure the performance of the estimates with respect to a point estimate (i.e., the mean). As such, in many cases, they can have deceptively low error values even when the actual estimates have high error, as these metrics depend on the scale and statistical characteristics of the entire data set.

Our results, except for the dynamic workload cases, are based on 5-fold cross validation (Chapter 2). That is, the data is divided into 5 equal-sized parts, 4 of which are used to build models for prediction on the remaining part. This process is repeated 5 times, hence all parts are used in testing. The reported prediction accuracy is the average of the

individual accuracy values from the testing of each cross-validation part. We used *stratified sampling* for dividing the data into 5 parts to ensure that each part contains roughly equal number of queries from each template.

6.3.2 Prediction with Optimizer Cost Models

We start with results showing predictions on top of analytical cost models used by conventional optimizers are non-starters for QPP. Specifically, we built a linear regression model to predict the query execution times based on the query optimizer cost estimates. Overall, the maximum relative error is 1744%, the minimum relative error is 30% and the mean relative error is 120%[1].

To provide more intuition into the reasons, we show the optimizer costs versus the query execution times for a subset of the queries (a stratified sample) on the 10GB TPC-H data set in Figure 6.5. Observe that the lower left and lower right data points correspond to queries with roughly the same execution times, even though their cost estimates have a magnitude of difference. Similarly, the data points on the lower and upper right corners are assigned roughly identical plans costs by the optimizer but differ by two orders of magnitude in their execution times.

In this setup, most queries are I/O intensive. We expect this to be the ideal case for predicting with analytical cost models. The reason is that optimizer cost models generally rely on the assumption that I/O is the most time consuming operation. Therefore, for CPU intensive workloads, we would expect to see even lower accuracy values.

As a concrete example, consider TPC-H template-1, which includes an aggregate over numeric types. We noticed that evaluating aggregates over numeric types can easily become the bottleneck, because arithmetic operations are performed in software rather than hardware. As such, introducing additional aggregates to a query can significantly alter the execution time even though the volume of I/O (and hence the predictions with the cost

[1] In this case, the predictive risk [41] is about .93, which is close to 1. This result suggests that it performs much better compared to a point estimate, although the actual relative errors per query as we reported are high.

model) remains approximately constant.

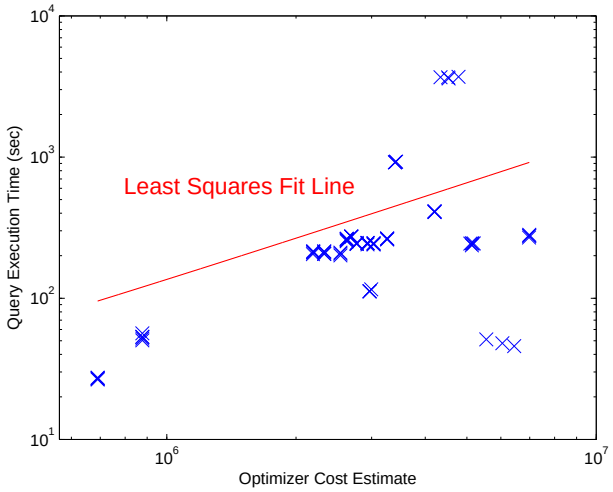


Figure 6.5: **Optimizer Cost vs Query Execution Time (log-log plot)**

6.3.3 Predicting for Static Workloads

Results for the plan-level and operator-level prediction methods both for the 10GB and 1GB TPC-H scenarios are given in Figure 6.6 and 6.7. These results were obtained using estimate-based features for building models in training and for prediction in testing. The use of actual (observed) values for features is discussed in Section 6.3.3.

Plan-level Modeling

Plan-level prediction is performed on all the 18 TPC-H templates. Overall, using SVMs we obtained on average 6.75% and 17.43% prediction errors for the 10GB and 1GB databases, respectively (Figure 6.6(a)-(c)). The prediction errors with KCCA modeling in the same scenarios were 2.1% and 3.1% (Figure 6.6(d)-(f)). The high accuracy results imply that plan-level modeling can be very effective for static workloads.

To shed some light on the difference in accuracy between the two model types, here we briefly describe the characteristics of these two models. With SVM-based regression, the

general approach is to map the query features to a high dimensional space (using a non-linear mapping) and perform linear regression in that space. In KCCA, the query features and the target values are projected to separate subspaces such that their projections are maximally correlated. Prediction with KCCA is then performed using a nearest neighbor strategy. As such the predictions of KCCA use all of the training data, whereas SVM results are only based on a subset of the points (support vectors). While SVM performs regression to produce its estimates, KCCA is a nearest neighbor approach. In addition to the feature selection used in SVMs and the tuning of the meta model parameters that affect training, we think that the described difference between the model types is also important. As such, we will see that SVM-based regression modeling will generalize better than KCCA to dynamic workload scenarios.

With SVM-based modeling, queries from template-9 stand out as the worst predicted set of queries. We note that template-9 queries take much longer than the queries of the other templates. As the number of instances of template 9, and therefore of longer running queries, is relatively few in both data sets, the prediction models may not fit well. To alleviate this problem, we built a separate SVM model for template-9 for the 10GB case, which reduced its error down to 7%.

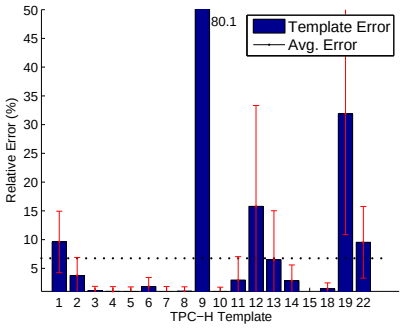
In addition, prediction for the 1GB database is a harder problem, as the respective ratios of the standard deviation to the average execution time of queries is about 2.63 times greater in the 1GB database case then the 10GB case.

Operator-level Modeling

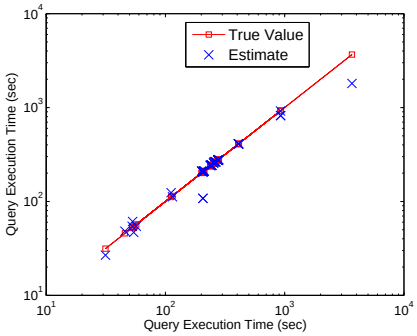
We now show operator-level prediction results on 14 of the 18 TPC-H templates[2].

For the 10GB case, in 11 of the 14 templates the operator-level prediction method performed better than 20% error (Figure 6.7(a)). For these 11 templates the average error is 7.30%. The error, however, goes up to 53.92% when we consider all the 14 templates, a

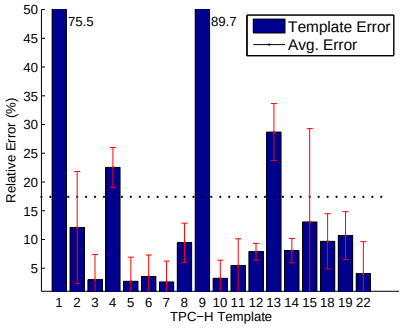
[2]The execution plans for the queries of the remaining 4 templates contain PostgreSQL-specific structures, namely INITPLAN and SUBQUERY, which lead to non-standard (i.e., non tree-based) execution plans with which our current operator-level models cannot cope at present.



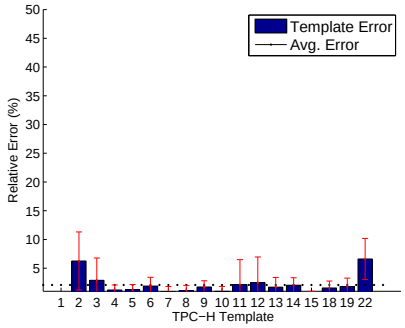
(a) SVM-based plan-level modeling, errors by template (10GB)



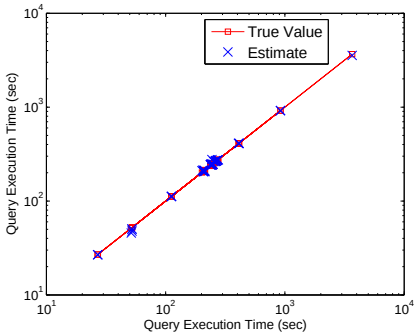
(b) SVM-based plan-level prediction (10GB)



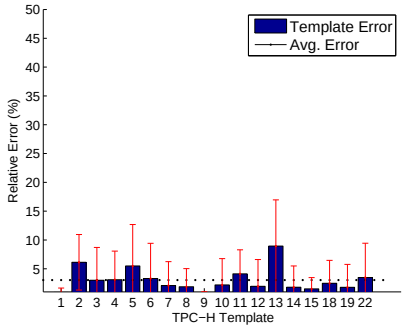
(c) SVM-based plan-level modeling, errors by template (1GB)



(d) KCCA-based plan-level modeling, errors by template (10GB)



(e) KCCA-based plan-level prediction (10GB)



(f) KCCA-based plan-level modeling, errors by template (1GB)

Figure 6.6: **Static workload experiments with plan-level modeling using SVMs and KCCA in 1GB and 10GB TPC-H databases. The error values in bar-plots are capped at 50%. Error values greater than 50% are printed next to the bars.**

significant degradation.

For the 1GB scenario, we show the results of operator-level prediction for the 14 TPC-H templates in Figure 6.7(c). In this case, for 8 of the templates the average error is below 25% and the mean error is 16.45%. However, the mean error for all the 14 TPC-H templates is 59.57% (slightly larger than the 10GB case).

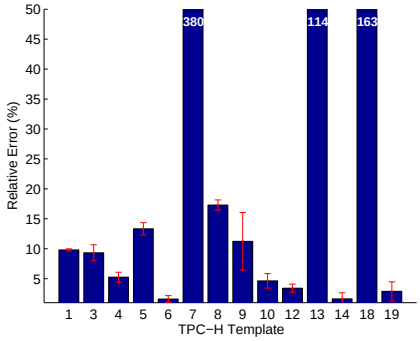
We see that operator-level prediction produces modest errors for many cases, but also does perform poorly for some. We analyzed the set of templates that belongs to the latter case, and noticed that they commonly exhibit one or more of the following properties:

- (Estimation errors) the optimizer statistic estimates are significantly inaccurate.
- (I/O-compute overlap) there is significant computation and I/O overlap in the query. The end-effect of such concurrent behavior on execution time is difficult to capture due to pipelining.
- (Operator interactions) The operators of the same query heavily interact with each other (e.g., multiple scans on the same table that use the same cached data).

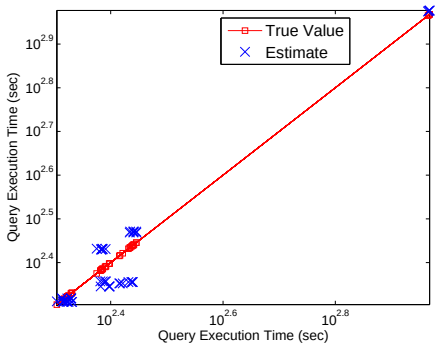
Next, we discuss the practical impact of statistics estimation errors on model accuracy. We then turn to the latter two issues that represent the fundamental limitations of operator-level modeling; that is, such models learn operator behavior "in isolation" without representing the context within which they occur.

Impact of Estimation Errors

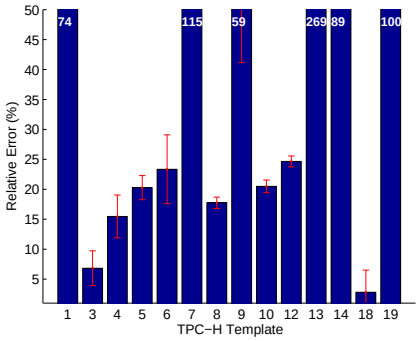
We tried all the combinations of actual and estimate feature values for training and testing for (SVM-based) plan-level and operator-level prediction. The results are given in Figure 6.8(a) for the 10GB scenario. For further detail, we also show the prediction errors grouped by TPC-H templates in Figure 6.8(b) for the actual/actual case and plan-level prediction (over the 10GB scenario). These results are to be compared with those in Figure 6.6(a).



(a) Operator-level, Errors by Template (10GB)



(b) Operator-level Prediction (10GB)



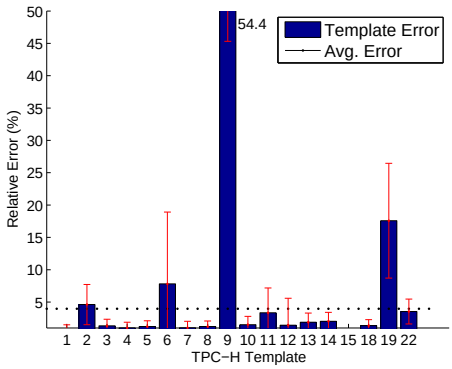
(c) Operator-level, Errors by Template (1GB)

Figure 6.7: **Static workload experiments with operator-level prediction methods using 1GB and 10GB TPC-H databases. The error values in bar-plots are capped at 50%. Error values beyond the limits of the plots are printed on the bars.**

+ Results for 14 TPC-H templates
* Results for 18 TPC-H templates

Train/ Test	Plan-level	Operator-level
ACT/ACT	%3.95* %3.40+	%41.85+
ACT/EST	%164.63* %69.32+	%1149.5+
EST/EST	%6.75* %5.95+	%53.92+

(a) Prediction with Actual Values vs Estimates



(b) SVM-based Plan-level Modeling with Actual Values (10GB)

Figure 6.8: **Impact of Estimation Errors on Prediction Accuracy in Static Workload Experiments**

Unsurprisingly, the best results are obtained in the actual/actual case (i.e., training and testing with actual feature values), which is not a viable option in practice due to the unavailability of the actual feature values without running the queries. The next best results are obtained with the estimate/estimate option (i.e., training and testing with estimated feature values), the option that we used in the rest of the paper. Finally, the results obtained with actual/estimate (i.e., training on actual values and testing on estimates) are much worse than the other two, primarily due to optimizer estimation errors that are not taken into account during training.

To provide a sense of the magnitude of the estimation errors made by the optimizer,

consider template-18, which is one of the templates that exhibit the biggest error in operator-level prediction with actual/estimate model building. Instances of template-18 include the following `group by` clause on table `lineitem`:

```
group by l_orderkey having sum(l_quantity) > 314
```

There are 15 million distinct `l_orderkey` values in `lineitem` (out of approximately 60 million tuples). The estimated number of groups satisfying `sum(l_quantity)` > 314 is 399521, whereas the actual number is 84. The PostgreSQL query optimizer computes this estimate using histograms (with 100 bins) for each column based on the attribute independence assumption. The results are later fed into a Hash-Semi-Join, whose cost estimate is correspondingly very much off the mark.

Comparing the actual/actual against the estimate/estimate results, we observe that optimization estimate errors lead to, perhaps surprisingly, only a modest degradation in prediction accuracy. This result is due to the ability of the models to also integrate error corrections during learning. Thus, while better estimations generally mean better results, it is possible to produce highly accurate predictions even with rather mediocre estimations (as in the case of PostgreSQL).

Hybrid Prediction Method

We now present comparative results of the three plan ordering strategies (see Section 6.1.4) discussed for offline hybrid model selection. The results, shown in Figure 6.9, were obtained with the 14 TPC-H templates used in operator-level modeling and the 10 GB database.

As described earlier, we first create an ordered list of query sub-plans based on the chosen plan ordering strategy, leaving out sub-plans with average error lower than a given threshold (.1 in this experiment) for the size-based and frequency-based strategies. Then, at each iteration (x-axis), we create a (SVM-based) model for the next plan in the ordered list, add this model to the current model set and then re-evaluate predictive error on the

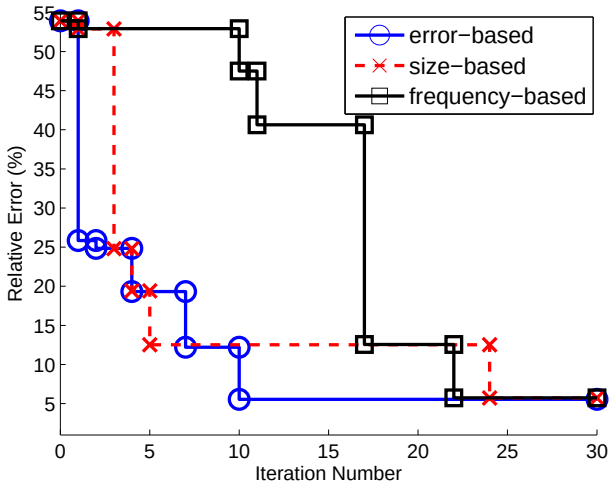


Figure 6.9: **Hybrid Prediction Plan Ordering Strategies**

test workload (y-axis). The step behavior is observed when a newly created model decreases the error.

We observe that the size-based and error-based strategies quickly reduce the error rate. The size-based strategy takes longer to reach the minimum error level, as in some cases larger sub-plans should be modeled for reducing the error and it takes time for this strategy to reach those plans.

The frequency-based strategy initially takes longer to reduce the error. The reason is that this strategy can easily get stuck in a relatively large sub-plan that has a high occurrence rate, since it needs to explore all the sub-plans involved in the larger sub-plan (starting from the smallest sub-plan) until it decreases the error rate. As discussed earlier, all such sub-plans are by definition at least as frequent, hence need to be explored with this heuristic. Overall, the error-based strategy provides a well balanced solution, quickly and dramatically reducing the prediction errors only with a small number of additional models. We also note that the final accuracy obtained with the hybrid-method approaches to that of the KCCA-based plan-level model.

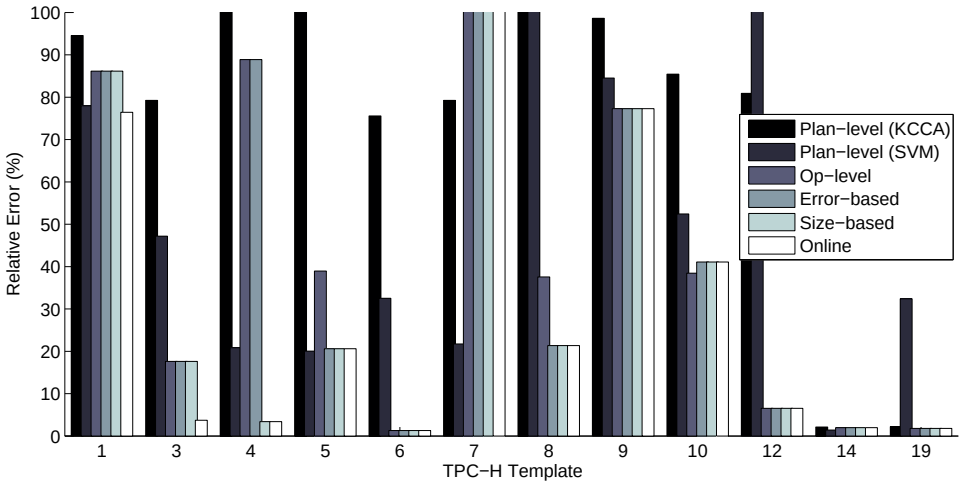


Figure 6.10: **Dynamic Workload Prediction Results**

6.3.4 Predicting for Dynamic Workloads

The results so far have shown that for known, static workloads, plan-level modeling performs well. They have also revealed that hybrid models offer similar accuracy to plan-level models for static workloads. Next, we present results demonstrating that plan-level modeling has serious limitations for unknown or changing workloads, whereas hybrid modeling still continues to provide high accuracy. We also report comparative results for online model building (Section 6.2) that creates custom hybrid models for a given query from the available training data.

For this experiment, we used the 12 templates shown in Figure 6.10, with 11 of them used in training and the remaining for testing. That is for each template we build and test separate prediction models based on the training data of the other templates. The two other TPC-H templates were excluded because they include specific operators exclusively found in those templates, and thus cannot be modeled with our current setup. We show results for KCCA-based plan-level, SVM-based plan-level, operator-level, hybrid (with error-based and size-based strategies), and online modeling algorithms.

As expected, plan-level models perform poorly across the board and thus do not offer

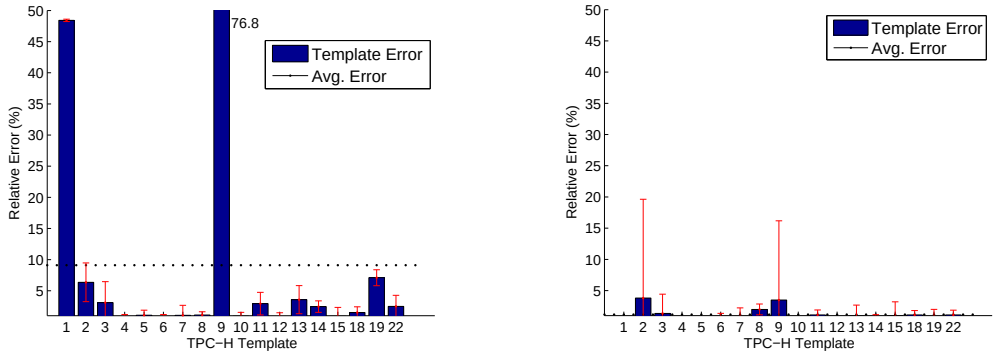
much value in the presence of dynamic workloads. However, SVM-based plan-level modeling performs significantly better than the KCCA-based approach. We also observe that the online (hybrid) modeling algorithm performs best in all cases, except for template-7. Further investigation reveals that the training data lacks a specific sub-plan that is the root cause of the error on template-7. These results confirm the ability of online modeling to identify the models that are very likely to help by utilizing the knowledge of a given query plan. Such models can be eliminated by offline strategies if they do not help improve training accuracy.

Another interesting observation is that the size-based hybrid strategy performs somewhat better than the error-based strategy in these experiments. This can be explained by the ability of the former to favor models for smaller sub-plans that are more likely to occur in unseen queries.

6.3.5 Platform Independence

In this experiment, we apply our QPP techniques on a different hardware platform to demonstrate its applicability in different environments. We show results from a 10GB TPC-H experiment executed on a 2.8GHz machine with 8GB RAM running Linux kernel 2.6.31. The database buffer pool size was set to 2GB. We used the same set of queries that were previously used in the 10GB experiment. Plan-level prediction results (following a static workload scenario) using SVMs and KCCA are shown in Figure 6.11(a) and 6.11(b) respectively.

The average prediction errors are slightly lower than the errors obtained in the previous 10GB experiment (see Figure 6.6). On this new hardware platform the TPC-H queries execute faster than on the previous platform. For instance, template-9 queries finish under 20 minutes instead of an hour. As such the query run times are less divergent. This is because of the higher disk speed in the new platform, 83MB/sec sustained read rate (versus the 55MB/sec read rate before) as well as the faster CPU and increased RAM size. In addition, there is much less variance on the execution times of queries in this setup.



(a) SVM-based plan-level modeling, errors by template (10GB)

(b) KCCA-based plan-level modeling, errors by template (10GB)

Figure 6.11: **QPP on a different hardware platform**

We attribute this to the higher disk speed and to the increased RAM size in the new platform. Since 80% of the database can fit into memory, the number of duplicate reads from the disk are significantly reduced (due to the filesystem and database caches). The lower variance of query execution times observed in this data set enables improved accuracy on the predictions.

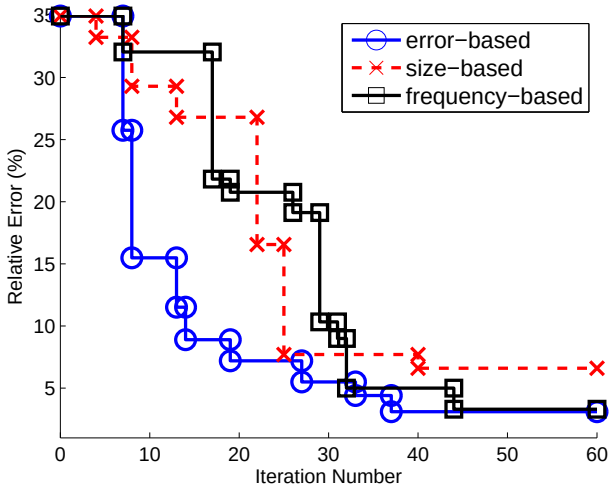


Figure 6.12: **Platform Independence: Hybrid QPP**

In Figure 6.12, we show QPP results (based on the static workload scenario) using

hybrid modeling with different plan ordering strategies. As before, the error-based strategy reduces the prediction error faster than the other strategies. While the error and frequency -based strategies converge to the same prediction error, the size-based method resulted in a slightly higher error value. This is possible since each plan-ordering strategy considers plan-level models in different order and the initially chosen models may cause the later plan-level models to be discarded. Finally, we note that the accuracy obtained with the hybrid prediction method (using SVM-based plan-level models) is higher than the accuracy of SVM-based plan-level modeling and close to the accuracy of KCCA-based plan-level model.

Chapter 7

Related Work

7.1 Machine Learning and Data Mining

7.1.1 Computational Learning

Learning algorithms have been studied for a long time in the statistics and applied math fields and more recently in machine learning and data mining areas. While statistics research mostly focused on the question of "What conclusions can be inferred from a data set?", machine learning and data mining research also focused on the applications and the computational side (i.e., tractability / intractability) of the question [75].

Research on learning produced a very large set of algorithms based on different assumptions and a variety of statistical techniques. The produced methods have been applied to a large set of applications such as computer vision [39], speech recognition [60], information retrieval [69] and bioinformatics [14]. For a summary of popular learning algorithms, including regression methods, classification techniques, neural networks, kernel-based methods and graphical models see [57, 15, 74, 51, 82]. [18] provides a good, practical introduction to time-series forecasting and [44, 68] provide a comprehensive survey of the area.

There is substantial work in the broad area of both database and stream mining of time-series data (see [50, 95] for a general overview), especially on similarity search and pattern matching (e.g.,[36, 81, 43, 102, 62]). The data mining field also contains a large

body of work on relevant sub-problems such as prediction, anomaly detection, forecasting, event prediction, and feature selection (e.g., [6, 64, 46, 98, 49, 70]).

7.1.2 Learning Packages

There are several open-source and commercial machine learning libraries and applications (see [38]) that provide a number of features. Weka [95] is an open source data mining library implemented in Java. It contains a large variety of classification, regression, clustering and visualization tools for data analysis. R [85] is a highly-extensible and widely used language and environment for statistical programming. It offers a wide range of integrated statistical and graphical techniques. RapidMiner [86] is an open-source system for data mining that can be integrated into applications or used as a standalone application. MATLAB [72] and Mathematica [90] are scientific computing platforms that both offer statistics solutions supporting a highly rich set of algorithms and visualization methods.

While most machine learning packages can use the database as a backend data source, they are not at all integrated with databases. This significantly limits their ease of use, scalability and efficiency; especially in large scale applications. Furthermore, none of these products provide continuous prediction capabilities over streaming data. In this thesis, we argued that the tight integration of models and data in the same system alleviates these performance and usability issues and could potentially change the way people build predictive, data-intensive applications.

7.1.3 Database Support for Models

Major commercial databases support predictive modeling tools (e.g., Oracle Data Mining tools [80], SQL Server Data Mining [73] and DB2 Intelligent Miner [54]). These allow users to invoke model instances (akin to stored procedures) on database tables using simple SQL extensions (e.g., the FORECAST clause in the Oracle). Some of these systems, such as the MSSQL Server, also provide graphical interfaces for their predictive analysis functionality.

In most cases, the provided predictive functionality lacks high-level capabilities (e.g.,

feature selection) and easy programmability. The integration of prediction models with the database does not go beyond the ability to run the models on database tables. Our system provides a tighter model integration and increased usability.

7.2 Probabilistic Databases and Uncertainty

There is extensive prior work on uncertain data management and probabilistic query processing [30, 13, 8]. These studies focus on the representation of uncertain data, generally rely on the possible worlds semantics, and aim to develop efficient query execution techniques over the proposed succinct data representations.

One example is the BayesStore [94] system, which is a probabilistic data management system that considers statistical models, data and inference algorithms as first-class citizens. Similar to our work, BayesStore also promotes the prediction operations and statistical models as part of the database operations. However, the problem of handling uncertain data within the possible worlds framework is an important but orthogonal problem to our work. In our study, we rely on statistical models to extract information from and perform inference using the existing data in a database system.

Both continuous and discrete probabilistic models have been used to represent imprecision and uncertainty (e.g., [26, 12]). There is also some existing work on statistical estimation of aggregation queries using random samples, such as in statistical databases [32] and online aggregation [52].

7.3 Model-based Data Management and Query Processing

MauveDB [34] is one of the inspirations for our predictive database system. The former supports model-based views defined using statistical models (e.g., a regression-based view) instead of standard SQL queries for a variety of purposes including cleaning, interpolation and prediction. MauveDB pushes the statistical modeling into the database system in order

to efficiently process and manage data (especially sensor data). In our system, we push the envelope significantly further for time-series forecasting models by treating models as first-class citizens with optimized training and model selection, as well as novel optimization techniques such materialized models, and staged processing.

Of particular relevance to our work is the Fa system [35] that supports both one-time and continuous declarative forecasting queries on time-series data sets. Fa discusses the various stages of model building and selection and introduces efficient execution strategies. Fa also describes incremental and shared computation techniques for the feature selection process in model building. In our work, we have not yet studied the optimization opportunities for feature selection, however such strategies can easily be integrated into our system for improving the model building times. Fa is a solid step towards basic forecasting query execution that works well for a small set of pre-defined models and small data sets. Our work is a substantial super-set; we scale up forecasting through novel cost-based optimization techniques such as model materialization, staged optimizations and also focus on integrating models as first class citizens.

Probabilistic inference on BNs is discussed in [57, 15, 82]. Wong et al. [97, 96] discussed how to implement the probabilistic inference methods within the database engine using an extended relational model. The product-join operator was also introduced in the same study. More recent work built on these results and discussed how to support and optimize BN-based inference queries inside a traditional database engine [17]. Other studies (e.g., [61]) used BNs in conjunction with databases for inference on streaming data. However, these are off-database approaches, as the probabilistic models are usually handled outside the DB engine.

In [63, 87], authors work on the problem of efficient processing of Markovian streams where Markovian streams are defined as the result of probabilistic inference on a temporal graphical model. For instance, the authors represent the trajectory of a person obtained through an RFID sensor network as a Markovian stream. More recently [63], a variety of access methods for Markovian streams are presented. One of the introduced techniques,

called the Markov Chain Index, is based on an idea similar to our materialization approach for BN-based prediction queries, as it provides efficient access to precomputed joint distributions of distant time steps in the stream.

In [33], the BBQ system, a declarative query processing engine for sensor networks, is described. In BBQ, multivariate Gaussian distributions are used to represent joint probability distributions on the monitored sensor network variables (e.g., temperature on sensor node 7). Users can specify model-based queries to compute range and point predictions as well as average aggregate queries.

7.4 Learning on Big Data

There has been a lot of work on distributed learning techniques within the machine learning and data mining communities [65]. These studies generally focus on parallelizing individual algorithms. For instance, in [45], a parallel version of Support Vector Machines (SVMs), called the Cascade SVM, is described. Cascade SVM, divides the data sets into partitions and optimizes each partition separately with multiple SVMs. These results are then combined and processed again using a 'cascade' of SVMs until convergence. However, scaling select machine learning algorithms, often by using very different methods, is not a general solution for scalable learning.

Recently, there has been a growing amount of interest in building more general distributed learning frameworks/solutions that support a variety of algorithms [27, 28, 55]. In [27], authors point out that learning algorithms which follow a Statistical Query Model are easily parallelizable using a small number of sufficient statistics. Examples of such algorithms include linear and multiple regression, k-means clustering, naive Bayes and PCA. The Mahout project [9] is an application of this approach in the well-known Map-Reduce [31] context. The MAD Skills study [28] focuses on similar parallelization methods and their integration within a parallel database system.

7.5 Query Performance Prediction

The query-plan level approach to query performance prediction has recently been studied [41]. In [41], authors consider plan-level query performance prediction using the TPC-DS query benchmark [79] and a customer query workload following the static workload assumption. They report that they can predict individual query execution times within 20% of the actual time for 85% of their test queries. In addition to the query execution time, estimation of other performance metrics such as disk I/O and message bytes is also considered. In this thesis, we focused only on the execution time performance metric. While we can apply our techniques separately for each performance metric, we plan to consider the extension to joint prediction of multiple metrics in future work.

In previous work, machine learning techniques have been used in the context of the database query optimizer [71, 92, 101]. In the learning optimizer project (LEO) [71, 92], model-based techniques have been used to create a self-tuning database query optimizer. The goal in [71, 92] is to produce better execution cost estimates for use in query optimization. The approach taken is to compare the estimates of the query optimizer with the actual values observed during query execution and repair the inaccurate estimates based on the obtained information. In [101], a statistical modeling technique called *transform regression* is used to create cost models for XML query operators. In addition, new training data can be efficiently integrated into their existing cost models for adapting to changing workloads.

Recently, there have been successful applications of machine learning techniques in system self-management problems. In [40], authors present a statistics-driven modeling framework for data-intensive cloud applications. Kernel Canonical Correlation Analysis (KCCA) predictive modeling techniques are used to make predictions for the execution performance of map-reduce jobs. In [25], a statistics-driven workload generation framework is presented for the purpose of identifying suggestions (e.g., scheduling and configuration) to improve the energy efficiency of map-reduce systems.

In [2, 3] an experimental modeling approach for capturing interactions in *query mixes*

(i.e., sets of concurrently running queries) is described. Given a query workload, the goal is to come up with a query execution schedule (in terms of query mixes) that minimizes the total execution time. The query interactions are modeled using statistical models based on selectively chosen sample executions of query mixes. In our study, we have not yet considered performance prediction in concurrent query workloads.

Finally, there has also been work on query progress indicators in database systems [22, 66]. Query progress indicators provide estimations on the completion degrees of running queries. Such studies assume that the work done by individual query operators are transparent, i.e., externally visible. While these studies are also closely related to query execution performance, they do not provide predictions for the execution time of queries.

Chapter 8

Conclusions

8.1 Summary

We presented techniques for in-database support of predictive functionality in order to address the usability and performance problems of the currently used off-database design for predictive applications. Specifically, we described techniques to integrate prediction models as first-class entities into database systems and provide declarative predictive functionality to users. We highlighted model management as the key underlying technology and showed that model management can greatly benefit from the well-established optimization and execution methods used for data management in database systems.

We discussed two main strategies for integrating prediction models within database systems: the white-box and black-box methods. In Chapter 4, we studied white-box support of Bayesian Networks and Dynamic Bayesian Networks in a streaming database system. We presented highly efficient and resource-adaptive execution and optimization strategies for prediction queries over the (Dynamic) Bayesian Network models. The type of prediction queries that were considered include classification queries, point and range -based queries, and top-k queries. We used two example applications to illustrate our techniques: Network Intrusion Detection and Software Performance Monitoring. Finally, we have implemented our white-box solutions on top of an open-source database system called H2 [48]

and demonstrated the efficiency and benefits of our system with extensive experimentation.

In Chapter 5, we described black-box support for prediction models in database systems. More specifically, we presented a framework for integrating and using existing implementations of prediction models in database systems. We implemented our techniques for black-box prediction support in Longview, a prototype predictive database system that we built on top of PostgreSQL [83]. Then in Chapter 6, using the Longview system we considered a system-facing predictive application, Query Performance Prediction (QPP), to demonstrate how predictive functionality can be leveraged to implement introspective services in existing systems. In the QPP application, we used predictive modeling techniques to learn query execution behavior at different granularities, ranging from coarse-grained plan-level models to fine-grained operator-level models. Our experimental results using the TPC-H benchmark [93] demonstrated that accurate QPP is possible both in static and dynamic workload scenarios.

Based on our research and experimental results, in this thesis, we argue that next generation database systems should natively support predictive functionality to improve performance, usability and extensibility of predictive applications. Our methods for integrating prediction models in database systems, and optimization and execution techniques for declarative prediction queries form a solid step in this direction. However, there is still much more to research and develop before predictive database systems can actually be useful in deployment. In the next, we outline a variety of these open issues for future investigation.

8.2 Open Challenges

8.2.1 Prediction Query Optimizer

We demonstrated the white and black -box model support techniques in different contexts (streaming vs. traditional). However, the query optimizer in a predictive database system should seamlessly integrate the use of both white-box and black-box prediction models

in query execution. In this way, the prediction query optimizer can compare a variety of prediction models with different characteristics based on their accuracy and efficiency, and choose a specific model for use in prediction. Such an approach paves the way for a more extensive and automated modeling functionality where users only specify the training and test data sets and predictive modeling (i.e., model building and testing) is performed automatically by the system.

In this approach, users are provided with a higher level predictive functionality where the prediction query optimizer is completely responsible for managing prediction models. To this end, the described prediction query optimizer would need to compute efficient execution plans that satisfy the required accuracy levels. Therefore, the planning process for prediction queries also depends on the cost of prediction and training with a prediction model. As such we need estimates of computation and storage costs for prediction models. Given these estimates, an initial attempt at building a prediction query optimizer would be to treat the prediction part of a query as a sub-query with a set of execution plans and associated costs. This simple approach would fit well with the current Dynamic Programming based query plan generators.

Typically, the query optimizer would need to train and test multiple prediction models as part of its plan generation process. However, depending on the size of the training data and the prediction model, the training of a model can take significant time. While this may not be an issue for continuous prediction queries (i.e., plan once, run forever) and offline processing environments, for interactive and exploratory data analysis environments, the query optimization time might be too large for user satisfaction. In such cases, we envision the use of pre-built prediction models. That is, instead of training and testing prediction models on the fly, the query optimizer would first check the set of available prediction models for existing accurate and low-cost models. This would require the database system to build and manage a set of prediction models specifically targeted for the observed query workload. For this purpose, the database system would need to identify the most common prediction attributes from the query workload and then find the set of features that

are highly predictive of those attributes. This process would also include identifying other characteristics of the query workload such as typical interval lengths in range queries, lead times in point-based prediction queries etc.

8.2.2 Optimized Model Training

Depending on the model type and training data set, building a prediction model can be a time consuming process. With the additional requirement of building multiple models for testing prediction accuracy and efficiency, the model training process can easily become the bottleneck in execution of prediction queries. To alleviate the model training problem, previously we discussed the idea of using pre-built prediction models in the plan generation process. We expect significant reduction in planning time for anticipated query workloads with the use of pre-built models. However, model building is still a problem for unpredictable/varying query workloads and the model pre-building phase.

The development of optimization techniques that reduce the I/O requirements of the automated model building algorithm (Section 5.3) are essential, especially for large data sets. The use of well-known optimization and data management techniques from database systems such as (multi-dimensional) indices, (materialized) views and column-based storage methods are expected to be highly applicable for this purpose.

We can also apply additional optimization strategies to the automated model building algorithm. Observe that, at each iteration of the algorithm, several new models are built by adding a feature to the set of features of an existing model. In this case, we can avoid repeated access to disk by combining the multiple read operations into a single operation and building the set of new models in parallel. In many cases, this strategy would significantly reduce the I/O requirements. However, some models create a replica of the training data in memory before building the model. In those cases, this strategy may not be applicable due to the elevated memory use. Such in-memory algorithms are also not usable with large-data sets (larger than memory). We can only use disk-based learning algorithms, online models

(i.e., models that can simultaneously be built as data is being read), and incremental models (i.e., models that are easily updated with new data) with large-data sets.

The described disk access optimization methods are targeted for the I/O intensive cases. However, the training of some prediction models could be CPU-intensive or the parallel building of multiple models could create a CPU-intensive scenario. This is one of the target environments for the use of white-box prediction models as aggressive, model-specific optimizations can be used for computational efficiency. On the other hand, the use of black-box prediction methods limits the available options for optimization. However, sampling-based techniques are still applicable. Sampling can be used to reduce the size of the training data and help reduce both the required computation and disk usage. It is possible to use sampling-based prediction models directly in query execution and also as a method of identifying promising models before prediction models on the entire training data are built.

Until now, all the described optimizations and execution strategies are targeted at improving the performance of modeling and prediction operations on single machines. However, with the increasing size of data sets and the availability of cost-effective cloud computing platforms, the use of distributed learning algorithms and parallel execution strategies is becoming a requirement for the large-scale/big-data applications. We would like to note that most of our techniques would be applicable in distributed environments and immediately benefit from the parallelization opportunities. However, a distributed environment brings additional research opportunities in many areas, e.g., data partitioning, the development and usage of distributed learning algorithms and parallel modeling on multiple nodes. We discussed some of the existing efforts in this direction in related works (Chapter 7).

Bibliography

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[2] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Modeling and exploiting query interactions in database systems. In *Proceeding of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 183–192, New York, NY, USA, 2008. ACM.

[3] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Qshuffler: Getting the query mix right. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1415–1417, Washington, DC, USA, 2008. IEEE Computer Society.

[4] Yanif Ahmad, Olga Papaemmanouil, Ugur Cetintemel, and Jennie Rogers. Simultaneous equation systems for query processing on continuous-time data streams. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 666–675, Washington, DC, USA, 2008. IEEE Computer Society.

[5] S. Akaho. A kernel method for canonical correlation analysis. In *In Proceedings of the International Meeting of the Psychometric Society (IMPS2001)*. Springer-Verlag,

2001.

[6] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1:66–77, August 2008.

[7] Mert Akdere, Uğur Çetintemel, and Eli Upfal. Database-support for continuous prediction queries over streaming data. *Proc. VLDB Endow.*, 3:1291–1301, September 2010.

[8] Lyublena Antova, Christoph Koch, and Dan Olteanu. Maybms: Managing incomplete information with probabilistic world-set decompositions. In *International Conference on Data Engineering*, pages 1479–1480, 2007.

[9] Apache. Mahout: Scalable machine-learning and data-mining library. `http://mahout.apache.org`.

[10] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15:121–142, June 2006.

[11] Francis R. Bach and Michael I. Jordan. Kernel independent component analysis. *J. Mach. Learn. Res.*, 3:1–48, 2003.

[12] D. Barbará, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Trans. on Knowl. and Data Eng.*, 4:487–502, October 1992.

[13] Omar Benjelloun, Anish Das Sarma, Alon Halevy, Martin Theobald, and Jennifer Widom. Databases with uncertainty and lineage. *The VLDB Journal*, 17:243–264, March 2008.

[14] Conrad Bessant, Ian Shadforth, and Darren Oakley. *Building Bioinformatics Solutions: with Perl, R and MySQL*. Oxford University Press, Inc., New York, NY, USA, 2009.

[15] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[16] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 54–65, 1999.

[17] Héctor Corrada Bravo and Raghu Ramakrishnan. Optimizing mpf queries: decision support and probabilistic inference. In *SIGMOD Conference*, pages 701–712, 2007.

[18] P. J. Brockwell and R. A. Davis. *Introduction to Time Series and Forecasting*. Springer, New York, 1996.

[19] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment, 2002.

[20] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[21] Chih-Chung Chang and Chih-Jen Lin. Libsvm : a library for support vector machines. `http://www.csie.ntu.edu.tw/~cjlin/libsvm`, 2001.

[22] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 803–814, New York, NY, USA, 2004. ACM.

[23] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 354–366, 1994.

[24] Surajit Chaudhuri and Kyuseok Shim. Optimizing queries with aggregate views. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '96, pages 167–182, London, UK, 1996. Springer-Verlag.

[25] Yanpei Chen, Archana Sulochana Ganapathi, Armando Fox, Randy H. Katz, and David A. Patterson. Statistical workloads for energy efficient mapreduce. Technical Report UCB/EECS-2010-6, EECS Department, University of California, Berkeley, Jan 2010.

[26] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 551–562, New York, NY, USA, 2003. ACM.

[27] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schlkopf, John Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[28] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. Mad skills: new analysis practices for big data. *Proc. VLDB Endow.*, 2:1481–1492, August 2009.

[29] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20:273–297, September 1995.

[30] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal, The International Journal on Very Large Data Bases*, 16(4):523–544, October 2007.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[32] Dorothy E. Denning. Secure statistical databases with random sample queries. *ACM Trans. Database Syst.*, 5:291–315, September 1980.

[33] Amol Deshpande, Carlos Guestrin, Samuel R. Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 588–599. VLDB Endowment, 2004.

[34] Amol Deshpande and Samuel Madden. Mauvedb: supporting model-based user views in database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 73–84, New York, NY, USA, 2006. ACM.

[35] Songyun Duan and Shivanath Babu. Processing forecasting queries. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 711–722. VLDB Endowment, 2007.

[36] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. *SIGMOD Rec.*, 23:419–429, May 1994.

[37] FIFA. Worldcup98 access logs. `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`.

[38] Forecasting. Principles web site. software programs. `http://www.forecastingprinciples.com/content/view/9/9`.

[39] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, 2002.

[40] Archana Ganapathi, Yanpei Chen, Armando Fox, Randy H. Katz, and David A. Patterson. Statistics-driven workload modeling for the cloud. In *ICDE Workshops*, pages 87–92, 2010.

[41] Archana Ganapthi, Harumi Kuno, Umeshwar Daval, Janet Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple performance metrics for queries: Better decisions enabled by machine learning. In *International Conference on Data Engineering*, 2009.

[42] Zoubin Ghahramani. Learning dynamic bayesian networks. In *Adaptive Processing of Sequences and Data Structures*, pages 168–197. Springer-Verlag, 1998.

[43] Dina Q. Goldin and Paris C. Kanellakis. On similarity queries for time-series data: Constraint specification and implementation. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 137–153, London, UK, 1995. Springer-Verlag.

[44] J. G. De Gooijer and R. J. Hyndman. 25 years of iif time series forecasting: A selective review. Tinbergen Institute Discussion Papers No. TI 05-068/4., 2005.

[45] Hans Peter Graf, Eric Cosatto, Leon Bottou, Igor Durdanovic, and Vladimir Vapnik. Parallel support vector machines: The cascade svm. In *In Advances in Neural Information Processing Systems*, pages 521–528. MIT Press, 2005.

[46] Xiaohui Gu, Spiros Papadimitriou, Philip S. Yu, and Shu-Ping Chang. Online failure forecast for fault-tolerant data stream processing. In *Proceedings of the 2008 IEEE*

24th International Conference on Data Engineering, pages 1388–1390, Washington, DC, USA, 2008. IEEE Computer Society.

[47] Lin Guo, Sihem Amer Yahia, Raghu Ramakrishnan, Jayavel Shanmugasundaram, Utkarsh Srivastava, and Erik Vee. Efficient top-k processing over query-dependent functions. *Proc. VLDB Endow.*, 1:1044–1055, August 2008.

[48] H2. Database engine. `http://www.h2database.com`.

[49] Mark A. Hall and Geoffrey Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Trans. on Knowl. and Data Eng.*, 15:1437–1447, November 2003.

[50] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[51] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2008.

[52] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In Joan Peckham, editor, *ACMSIGMOD International Conference on Management of Data*, pages 171–182, Tucson, May 1997. ACM Press.

[53] S. Hettich and S. D. Bay. The uci kdd archive. `http://kdd.ics.uci.edu`, Irvine, CA: University of California, Department of Information and Computer Science, 1999.

[54] IBM. Db2 intelligent miner web site. `http://www-01.ibm.com/software/data/iminer/`.

[55] IBM. Ibm parallel machine learning toolbox. `http://www.alphaworks.ibm.com/tech/pml`.

[56] Christian Igel, Verena Heidrich-Meisner, and Tobias Glasmachers. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.

[57] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer Publishing Company, Incorporated, 2nd edition, 2007.

[58] Jetty. open source web server. `http://www.mortbay.org/jetty`.

[59] M. Jordan. *Learning in Graphical Models (Adaptive Computation and Machine Learning)*. MIT Press, 1998.

[60] Jean-Claude Junqua and Jean-Paul Haton. *Robustness in Automatic Speech Recognition: Fundamentals and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[61] Bhargav Kanagal and Amol Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1160–1169, Washington, DC, USA, 2008. IEEE Computer Society.

[62] Byoung kee Yi, N. D. Sidiropoulos, Theodore Johnson, and H. V. Jagadish. Online data mining for co-evolving time sequences. In *Proceedings of the 16th International Conference on Data Engineering*, pages 13–, Washington, DC, USA, 2000. IEEE Computer Society.

[63] Julie Letchner, Christopher Re, Magdalena Balazinska, and Matthai Philipose. Access methods for markovian streams. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 246–257, Washington, DC, USA, 2009. IEEE Computer Society.

[64] Xiaolei Li and Jiawei Han. Mining approximate top-k subspace anomalies in multi-dimensional time-series data. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 447–458. VLDB Endowment, 2007.

[65] Kun Liu and Hillow Kargupta. Distributed data mining bibliography. `http://www.cs.umbc.edu/hillol/DDMBIB`, 2006.

[66] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. Increasing the accuracy and coverage of sql progress indicators. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 853–864, Washington, DC, USA, 2005. IEEE Computer Society.

[67] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[68] Spyros G. Makridakis, Steven C. Wheelwright, and Rob J. Hyndman. *Forecasting: Methods and Applications*. John Wiley & Sons, Ltd., January 1998.

[69] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[70] Jie Mao, John Jannotti, Mert Akdere, and Ugur Cetintemel. Event-based constraints for sensornet programming. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 103–113, New York, NY, USA, 2008. ACM.

[71] V. Markl, G. M. Lohman, and V. Raman. Leo: An autonomic query optimizer for db2. *IBM Syst. J.*, 42:98–106, January 2003.

[72] The Mathworks. Matlab. `http://www.mathworks.com`.

[73] Microsoft. Sql server 2008. `http://www.microsoft.com/sqlserver/2008/en/us/data-mining.aspx`.

[74] T. M. Mitchell. *Machine learning*. McGraw Hill, New York, 1997.

[75] T. M. Mitchell. The discipline of machine learning. `http://www.cs.cmu.edu/~tom/pubs/MachineLearning.pdf`, 2006.

[76] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit

Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[77] Kevin Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley, Computer Science Division, July 2002.

[78] MySQL. Prepared statements. `http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html`.

[79] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 1049–1058. VLDB Endowment, 2006.

[80] Oracle. Data mining web site. `http://www.oracle.com/technology/products/bi/odm/index.html`.

[81] Spiros Papadimitriou, Jimeng Sun, and Philip S. Yu. Local correlation tracking in time series. In *Proceedings of the Sixth International Conference on Data Mining*, ICDM '06, pages 456–465, Washington, DC, USA, 2006. IEEE Computer Society.

[82] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[83] PostgreSQL. Database engine. `http://www.postgresql.org/`.

[84] Project. Longview: Querying the future now. `http://database.cs.brown.edu/projects/longview/`.

[85] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008.

[86] RapidMiner. Data mining tool. `http://rapid-i.com`.

[87] Christopher Ré, Julie Letchner, Magdalena Balazinksa, and Dan Suciu. Event queries on correlated probabilistic streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 715–728, New York, NY, USA, 2008. ACM.

[88] Steven P. Reiss. Dynamic detection and visualization of software phases. *SIGSOFT Softw. Eng. Notes*, 30:1–6, May 2005.

[89] Steven P. Reiss. Visual representations of executing programs. *J. Vis. Lang. Comput.*, 18:126–148, April 2007.

[90] Wolfram Research. Mathematica. `http://www.wolfram.com`.

[91] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen chuan Chang. Top-k query processing in uncertain databases. In *International Conference on Data Engineering*, pages 896–905, 2007.

[92] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. Leo - db2's learning optimizer. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[93] TPC-H. Database benchmark specification. `http://www.tpc.org/tpch/`.

[94] Daisy Zhe Wang, Eirinaios Michelakis, Minos N. Garofalakis, and Joseph M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. *Proceedings of The VLDB Endowment*, 1:340–351, 2008.

[95] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Francisco, CA, 2nd edition, 2005.

[96] S. K. M. Wong, C. J. Butz, and Y. Xiang. A method for implementing a probabilistic model as a relational database. In *In Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 556–564. Morgan Kaufmann Publishers, 1995.

[97] Dan Wu and Michael Wong. Global propagation in bayesian networks vs semijoin programs in relational databases. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 13(5):539–560, 2005.

[98] Lei Yu and Huan Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *In Intl. Conf. on Data Engineering*, pages 856–863, 2003.

[99] Nevin Lianwen Zhang and David Poole. A simple approach to bayesian network computations. Tenth Canadian Conference on Artificial Intelligence, 171-178, 1994.

[100] Nevin Lianwen Zhang and David Poole. Exploiting causal independence in bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.

[101] Ning Zhang, Peter J. Haas, Vanja Josifovski, Guy M. Lohman, and Chun Zhang. Statistical learning techniques for costing xml queries. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 289–300. VLDB Endowment, 2005.

[102] Yunyue Zhu and Dennis Shasha. Query by humming: a time series database approach. *Proc of SIGMOD*, 2003.