Database-support for Continuous Prediction Queries over Streaming Data

Mert Akdere
Brown University
Providence RI, USA
makdere@cs.brown.edu

Uğur Çetintemel
Brown University
Providence RI, USA
ugur@cs.brown.edu

Eli Upfal
Brown University
Providence RI, USA
eli@cs.brown.edu

ABSTRACT

Prediction is emerging as an essential ingredient for real-time monitoring, planning and decision support applications such as intrusion detection, e-commerce pricing and automated resource management. This paper presents a system that efficiently supports continuous prediction queries (CPQs) over streaming data using seamlessly-integrated probabilistic models. Specifically, we describe how to execute and optimize CPQs using discrete (Dynamic) Bayesian Networks as the underlying predictive model. Our primary contribution is a novel cost-based optimization framework that employs materialization, sharing, and model-specific optimization techniques to enable highly-efficient point- and range-based CPQ execution. Furthermore, we support efficient execution of top-k and threshold-based high probability queries. We characterize the behavior of our system and demonstrate significant performance gains using a prototype implementation operating on real-world network intrusion data and deployed as part of a real-time software-performance monitoring system.

1. INTRODUCTION

Traditional data management systems enable users to efficiently query the past state of the world as is represented by the database. Many modern applications require a streaming database system that can deliver results with low latency, hence enabling the user to query the present state of the world. 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. 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.

An important subclass of predictive applications involve real-time processing. 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). Such real-time predictive applications demand the key ability to continually perform real-time predictions. Currently, these applications are typically supported by application-level code that uses the underlying database as a dumb data store. The broad goal of our work is to push this key real-time predictive capability to the database layer in an attempt to leverage existing data modeling, query execution and optimization frameworks while providing generic, highly-efficient CPQ support. Most of this functionality, which are available in DB systems (already debugged/optimized), needs to be duplicated to perform similar optimizations when prediction is done outside the DB engine.

To this end, this paper describes a system that efficiently supports Continuous Prediction Queries (CPQs) over real-time data streams. In particular, we describe how to execute and optimize CPQs on top of probabilistic inference models, specifically the discrete Bayesian Networks (BNs) [1, 4] and their variant Dynamic Bayesian Networks (DBNs) [16]. Of course, 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 [1, 4]. 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.

Previous work showed how BNs can be represented by a functional relational model and predictive queries be supported using extended relational operators [2, 8]. We build on these results and 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 optimization framework that combines materialization, sharing and model-specific optimization techniques for CPQs using BN-based predictive models.

In more detail, our contributions can be outlined as follows:
- CPQs using BNs: We use BNs to support CPQs without temporal attributes (e.g., predict whether each incoming network connection is an attack or not). BNs are graphical models that use conditional independence information to compactly represent joint distributions of variables. Given a BN on a set of variables, we have a complete representation of the joint distribution, on which various inference algorithms (e.g., variable elimination and junction tree propagation [1, 4, 10, 22]) can be used to perform probabilistic prediction. To provide real-time response, it is imperative to operate in main memory. Since joint distributions can get extremely

VLDB '09, August 24-28, 2009, Lyon, France

large, even for relatively small BNs, we introduce a memory-aware *materialization* technique in which entire distributions or their select portions can be materialized to leverage the available memory. To guide the materialization choices, we describe a cost model with which computation and memory requirements of each candidate plan are estimated using simple statistics. We then propose a Dynamic Programming (DP) algorithm to find the minimum cost query plan that can be executed with the available memory.

- Temporal CPQs using DBNs: Dynamic BNs are a class of BNs specialized for modeling time-varying variable sequences and can compactly represent point- and range-based CPQs (e.g., predict the CPU usage for the next 10 secs). Since DBNs can be represented as constrained BNs, the optimization techniques we developed for BNs are immediately applicable to DBNs. Moreover, supporting range-based prediction queries over DBNs require a form of iterative inference, which we speed up significantly using a novel *sharing and incremental evaluation* technique.

- Integrated database-style and model-specific optimizations: In addition, we introduce a number of (D)BN-specific optimization techniques for queries computing high probability events and for top-k queries predicting the most likely sequences of future events.

- CPQ Engine Prototype on a main-memory DBMS and evaluation using real-world data and applications: We prototyped a CPQ system that implements a BN-based CPQ execution engine and our cost-based optimization framework by modifying H2, an open-source embedded DBMS [21]. We evaluate our system using a real-world network intrusion data set and a software performance monitoring system that we deployed on our local cluster.

The rest of the paper is structured as follows. In §2.1, we describe CPQs in detail using the motivating applications. CPQ execution is presented in §2.2. Then, we present the dynamic programming that guides our CPQ plan selection in §2.3. We discuss a collection of optimization extensions in §2.4. We present our experimental evaluation in §3. The related work is in §4 and conclusion is in §5. The appendix contains background information on BNs, describes inference on BNs using database operations based on [2, 8] and the details of our cost model.

2. CONTINUOUS PREDICTION QUERIES OVER DATA STREAMS

2.1 Overview and Running Examples

We describe the CPQs and the overall functionality of our system via two representative use cases: (i) a network intrusion detection system (NIDS) adapted from [9], and (ii) a dynamic software performance monitoring (DyMon) application [13, 14]. NIDS is primarily used to illustrate the BN-based prediction techniques, whereas DyMon is used to discuss the temporal CPQs supported by DBNs.

The NIDS Application: In the 1999 KDD Intrusion Detection Contest [9], each network connection is described with a set of 42 variables e.g., 'protocol type' 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'. [1]

We use the BN given in Figure 1 for predicting the access type for each connection. The given BN 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 [3] for forming the BN structure. Structure and parameter learning for BNs are discussed in appendix §A.3.

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

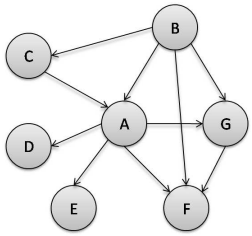

Figure 1: Network Intrusion Detection System BN. Variable A is the `access_type` and others are the observable variables. The BN corresponds to the factorization: $P(A,B,C,D,E,F,G) = P(B)P(C|B)P(A|B,C)P(G|A,B)P(D|A)P(E|A)P(F|A,B,G)$.

The DyMon Application: The Dynamic Software Performance Monitoring (DyMon) 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, CPU use and memory use. We monitor the performance of a web server [20] under a load of web requests replayed from the 1998 FIFA web logs [18][2]. The setup for the system is described in detail in appendix §B. We will discuss both point and range prediction queries using the DyMon application. We will use the DBN shown in Figure 3 for illustrating our query optimization techniques in §2.2.2.

2.2 CPQ Execution

2.2.1 Query Execution with Bayesian Networks

We describe the CPQ execution for BN-based inference using the network intrusion detection query example given in Section 2.1. Recall that we would like to calculate $P(access\_type|evidence)$. 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 calculate $P(access\_type, evidence)$ and normalize it to produce the query result. Substituting in the variable names used in Figure 1 and using lowercase letters for observed variables, we need to calculate $P(A, b, c, d, e, f, g)$. We use the example query evaluation tree given in Figure 2. The query tree contains product-join ($\overset{*}{\bowtie}$) and selection (σ) operations. The product-join together with marginalization operation [2] are used for BN inference in databases (see §A.2). At a high level, the product-join operation is used for computing the joint distribution of input distributions (e.g., $P(X,Y) = P(X) \overset{*}{\bowtie} P(Y|X)$), whereas the marginalization operation (combination of projection and group-by operations) is used for eliminating variables from distributions.

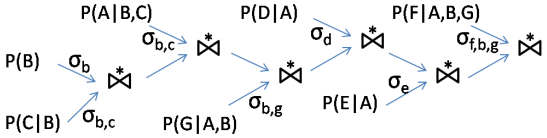

Figure 2: An example query execution tree for $P(A, b, c, d, e, f, g)$.

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 query execution with `prepared` statements [15].

[1]The KDD99 contest also involves identifying the attack type for bad connections, which we do not consider in this paper.

[2]Only the number of requests per time unit was used from the FIFA web logs. The accessed pages were created by the authors.

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 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 $P(B)$ and $P(C|B)$. 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 2. 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., marginalization) to calculate $P(A, b, c, d, e, f, g)$. Hence, there are no group-by or projection operations, which are used to implement the marginalization operation in a database as discussed in §A.2, in the query tree of Figure 2. 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 §2.2.2.

2.2.2 *Query Execution with Dynamic Bayesian Networks*

We use the DBN shown in Figure 3 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, there is a further opportunity of sharing computation and materialized results due to the structure and assumptions of a DBN, especially in the case of 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 3, 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 $cpu[t + k]$.

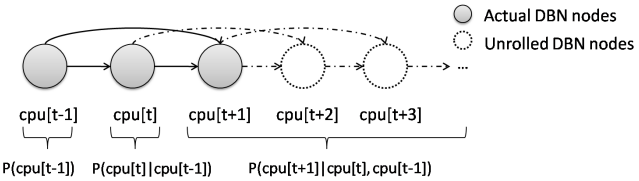

Figure 3: A three time-slice DBN for the CPU variable is unrolled to include future time points. Newly added nodes share the same distribution with the variable cpu[t+1].

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. 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.

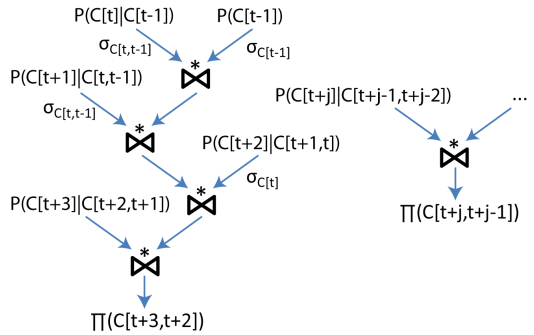

Figure 4: An execution tree for calculating the probability $P(cpu[t + 3]|cpu[t], cpu[t - 1])$ is given on the left (C is used to denote CPU). On the right is a template query tree for the described iterative operation in generating plans for arbitrary point-based prediction queries.

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 3, 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 Section 2.4, 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 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 $P(cpu[t+k]|cpu[t], cpu[t-1])$ is actually built on the execution tree of the query $P(cpu[t+k-1]|cpu[t], cpu[t-1])$. Hence, the most basic optimization is to share the computation across these queries by using a combined execution plan as shown in Figure 5.

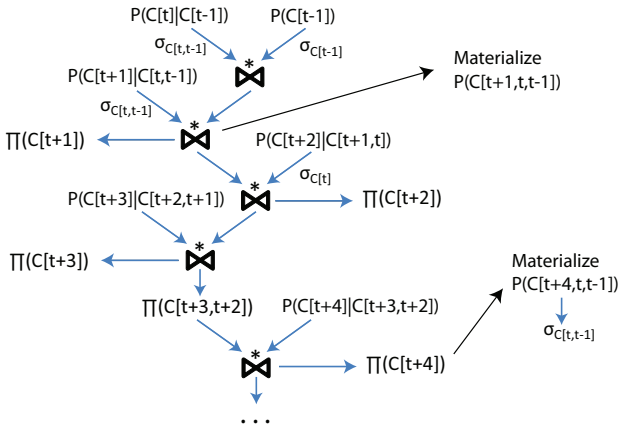

Figure 5: An execution tree for the range-based query computing $P(cpu[t+k]|cpu[t], cpu[t-1])$ for a range of k values is given for the first 4 time points (C is used to denote CPU). The marginalization operations containing only the observed variables are not shown.

The query execution tree in Figure 5 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 5, 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 as 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.

2.3 Plan Selection

2.3.1 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 systems [23]. For storage costs, which we need for estimating the memory requirements of an execution plan, we assume uniform space requirements for all tuples. The details of our cost model are described in appendix §C.

2.3.2 Generating CPQ Execution Plans

Next, we first modify the Selinger-style Dynamic Programming (DP) algorithm given for query optimization in [2, 11, 12], 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 DP 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** $r_j, S_j : S_j \subseteq S \wedge |S_j| = l, Q' = S_j \cup r_j$ **do**
11. $p_{Q'}.add(product\_join(optplans(S_j), r_j))$
12. $p_{Q'}.add(product\_join(GroupBy(optplans(S_j)), r_j))$

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, 11 and 12, 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. The size of the plan space is discussed in appendix §D.

Plan generation for DBN-based CPQs:
For the **point-based queries** on DBNs, the DP algorithm discussed for the BN case can be used with minor changes. First, the DBN has to be unrolled as shown in Figure 3, 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 ap-

ply 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.

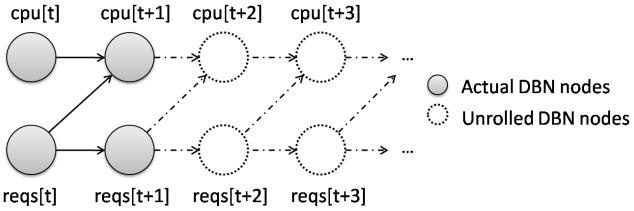

Figure 6: A two-slice DBN consisting of the variables CPU and REQS. CPU represents the CPU usage and REQS is the number of requests received by the web server in the DyMon application.

As described in Section 2.2.2, 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 6 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 7. 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.

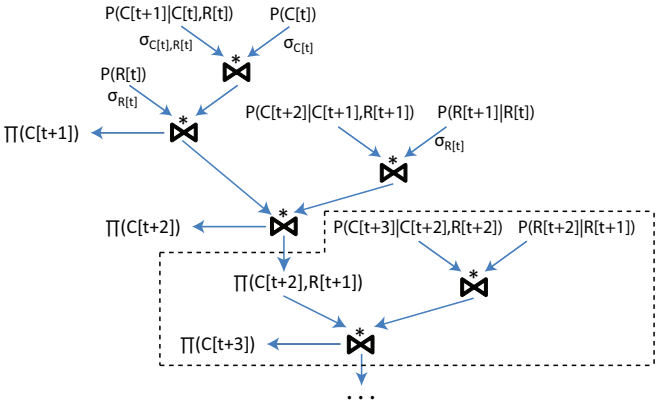

Figure 7: A query execution plan for the range-based query predicting the CPU value based on the DBN shown in Figure 6. C is used to denote the CPU variable and R is used for the REQS variable. Finally, the highlighted area is the instantiation of the template plan for time 3.

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 part of the plan highlighted with the dashed rectangle in Figure 7 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.

2.4 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 2, 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:

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

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 2 using the same technique and reduce computation without introducing errors in the query results. 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 [24, 25]. For instance, in the DyMon application one could specify a top-k query to generate only the top-k predic-

tions 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:

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

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.

3. EXPERIMENTS

3.1 Experiment Setup

Prototype Implementation: We implemented the described algorithms for continuous prediction queries in Java by modifying H2 [21], 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.

The NIDS experiments are in §3.2 and the DyMon experiments are in §3.3. In §3.4 we provide a comparison of our system with traditional non-DB inference methods. Further experimental results are given in appendix §E.

3.2 Network Intrusion Detection Results

The NIDS dataset was obtained from the 1999 KDD Intrusion Detection Contest [9]. 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 1. 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 8. The tradeoff between materialization (memory usage) and computation time, as discussed in §2.2.1, 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 (appendix §C), 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.

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 8: 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.

Number of Variables vs. Query Execution Time: In Figure 9, 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 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.

The results in Figure 9 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 8. 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 §2.4 for eliminating the low probability events from

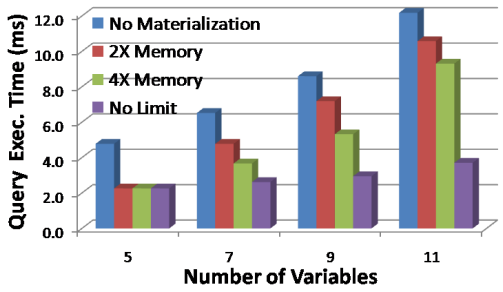

Figure 9: The avg. query execution time vs the number of variables in the NIDS BN presented for different levels of available memory.

the materialized distributions. The results are shown in Figure 8, 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.

3.3 Software Performance Monitoring Results

Experiments on the DyMon application were performed using the setup described in §2.1. 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, $P(cpu[t+k]|cpu[t], cpu[t-1])$, given in §2.2.2, 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 10. 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.

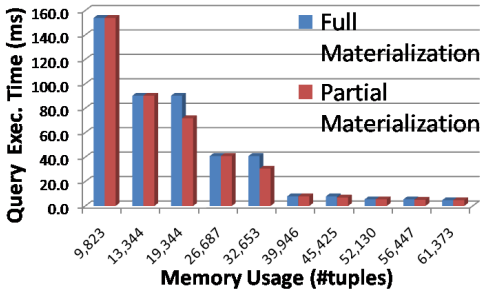

Figure 10: Memory Usage vs Computation tradeoff for the DyMon scenario using the DP algorithm with different materialization options.

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. 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. Finally, if we do not place

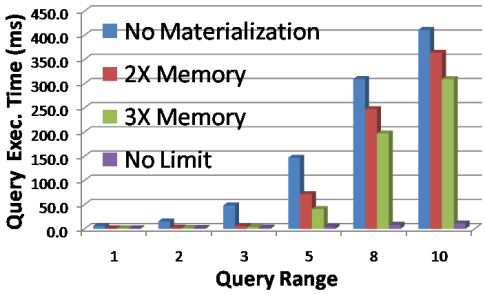

Figure 11: Average query execution time for increasing query ranges and under different memory budgets.

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 (§2.4) 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 12, 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*.

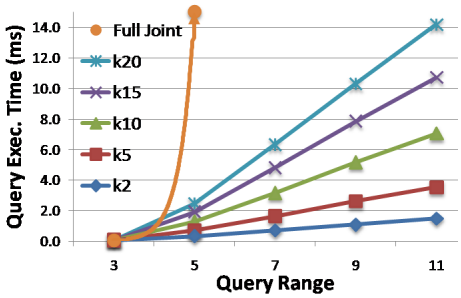

Figure 12: Average execution times for the top-k queries predicting the k most likely CPU sequences for different range and k values.

3.4 Black-box vs. White-box Inference

A key premise of this work is that an in-database (white-box) approach for inference can offer substantial benefits over the off-database (black-box) approaches, which we quantitatively demonstrate in this section. As a representative black-box system, we use the open-source "Weka" software (ver. 3.6.1) [19], 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 (§3.2), we obtained 4.5 ms execution time (per incoming tuple) with Weka's BayesNet classifier. Recall from Figure 8 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 (§3.3). As Weka does not support DBNs, there was no straightforward way to use it for this task. Our initial 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 (§3.3) 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.

4. RELATED WORK

Probabilistic inference on BNs is discussed in [1, 4]. There are two main classes of inference algorithms: exact and approximate. Variable elimination (VE) is an example of exact inference algorithms. In this paper, we have exclusively focused on exact probabilistic inference. While some approximate methods may be more efficient and better suited for certain applications, it is well-known that both exact and approximate inference are NP-hard.

Wong et al. [7, 8] discussed how to implement the probabilistic inference methods within the database using an extended relational model. The product-join operator was also introduced in the same study. More recent work discussed how to support and optimize inference queries inside a traditional database [2]. In [6], access methods for probabilistic data streams are presented. One of the techniques, called the Markov Chain Index, is based on an idea similar to our materialization approach (albeit considered for on-disk materialization), as it provides efficient access to precomputed joint distributions. Our techniques borrow the core relational BN representation and inference model and extends them to address continuous predictions over data streams. Our memory-aware materialization and sharing-based optimization techniques and support for DBNs are not considered by prior work.

Forecasting queries have been discussed in the context of the Fa system [3]. Fa considers a collection of forecasting models such as Bayesian Networks (BNs), Support Vector Machines (SVMs), and Multivariate Linear Regression (MLR). Authors consider an incremental approach to building models in which more variables are added to the model in successive iterations. There is also an extension to processing continuous foreasting queries in the study. However, Fa's approach is less tightly integrated to the database system and does not consider the optimizations we describe.

Other recent studies (e.g., [5]) used BNs in conjunction with databases for inference on streaming data. However, these are black-box approaches, as the probabilistic models are usually handled outside the DB. On the other hand, our work follows the white-box approach by providing low-level support for probabilistic inference and CPQs at the core database level.

5. CONCLUSIONS

In this paper, we discussed how real-time predictive analytics over data streams can benefit from database support. Our system seamlessly integrates a Bayesian Network-based predictive model into its relational continuous query execution and cost-based optimization framework. Our prototype-based results on real-world data and applications verify and quantify the benefits of our system, and, in particular, the various optimizations we proposed.

6. REFERENCES

[1] Jensen, F. V. Bayesian Networks and Decision Graphs. Springer-Verlag, 2001.

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

[3] Duan, S. and Babu, S. Processing forecasting queries. VLDB 2007.

[4] Pearl, J. Probabilistic Reasoning in intelligent systems: networks of plausible inference. Morgan Kaufmann, 1988.

[5] Kanagal B., Deshpande A. Online Filtering, Smoothing and Probabilistic Modeling of Streaming data. ICDE 2008.

[6] Letchner J. et al. Access Methods for Markovian Streams. ICDE 2009.

[7] Wu D., Wong M.: Global Propagation in Bayesian Networks Vs Semijoin Programs in Relational Databases. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 13(5), 2005.

[8] Wong S.K.M., Butz C.J., and Xiang Y. A method for implementing a probabilistic model as a relational database. UAI, 556-564, Montreal, 1995.

[9] Hettich, S. and Bay, S. D. The UCI KDD Archive [http://kdd.ics.uci.edu]. Irvine, CA: University of California, Department of Information and Computer Science 1999.

[10] Zhang N.L. and Poole D. Exploiting causal independence in Bayesian networks inference, JAIR 5, 1996.

[11] Chaudhuri S. and Shim K. Including Group-By in Query Optimization. VLDB'94.

[12] Chaudhuri S. and Shim K. Optimizing queries with aggregate views. In EDBT'96.

[13] Reiss, S. P. Dynamic detection and visualization of software phases. WODA '05.

[14] Reiss, S. P. Visual representations of executing programs. Journal of Visual Languages and Computing 18, 2, 2007.

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

[16] Murphy K. "Dynamic Bayesian Networks: Representation, Inference and Learning". PhD Thesis. UC Berkeley, 2002.

[17] Ghahramani Z. Learning Dynamic Bayesian Networks. Adaptive Processing of Sequences and Data Structures. Lecture Notes in Artificial Intelligence, 1387, 168-197, 1998.

[18] Arlitt M. and Jin T., "1998 World Cup Web Site Access Logs", August 1998. www.acm.org/sigcomm/ITA.

[19] Witten I.H., Frank E. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufman, 2005.

[20] Jetty, open source web server. http://www.mortbay.org/jetty/

[21] H2 Database Engine. www.h2database.com

[22] Zhang N. L. and Poole D. A simple approach to Bayesian network computations. Tenth Canadian Conference on Artificial Intelligence, 171-178, 1994.

[23] Boncz P., et al. Database architecture optimized for the new bottleneck: Memory access. VLDB 1999.

[24] Guo, L., et al. Efficient top-k processing over query-dependent functions. PVLDB 2008.

[25] Soliman M. A. and Ilyas I.F. Top-k query processing in uncertain databases. ICDE 2007.

APPENDIX

A. BACKGROUND

A.1 Bayesian Networks

Bayesian Networks (BNs) [1, 4] 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 $x = \{x_1, x_2, .., x_N\}$, 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 [2, 7, 8]. A functional relation R has the schema $\{A_1, A_2, .., A_n, f\}$ where f is called the measure attribute and the functional dependency $A_1, A_2, ..., A_n \rightarrow f$ 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 13 together with the functional relations for each CPD.

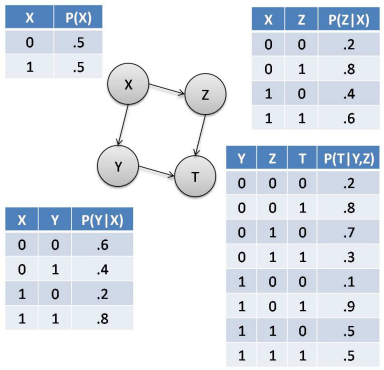

Figure 13: An example Bayesian Network, defined over the binary attributes X, Y, Z and T, representing the joint distribution $P(X, Y, Z, T) = P(X)\ P(Y|X)\ P(Z|X)\ P(T|Y, Z)$.

Dynamic Bayesian Networks (DBNs) [16, 17] are a natural extension of the Bayesian Networks (BNs) for modeling dynamic systems, i.e., those that evolve with time. Some basic continuous inference queries that the users would like to pose in such systems could include both 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)
`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 with 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 the query `2`. However, in this case we will need to introduce 10 variables, 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 as the number of variables grows large. 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 by adding additional nodes.

DBNs represent a similar approach except that there are certain restrictions which help reduce the network size and thereby make inference more efficient. The restrictions 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, and
3. *Identical distributions:* Conditional probability distributions for the same attribute at different time steps are the same.

The DBN shown in Figure 14 can be used to answer both queries `1` and `2` (inference with BNs and DBNs is discussed in Section A.2). 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.

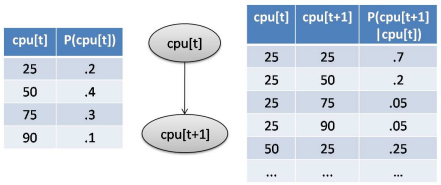

Figure 14: An example DBN consisting of two time slices in which the current CPU value depends only on the previous value.

A.2 Inference with Bayesian Networks

The inference problem in BNs is the problem of computing the posterior distribution of a set of variables given a set of observed variables. There are a variety of inference algorithms for BNs [1], most of which try to exploit the BN structure, which encodes the conditional independences, to efficiently compute the posterior distributions. In our work, we use variable elimination (VE) [10, 22], one of the most popular inference algorithms.

The VE algorithm, like many other inference algorithms, is based on a set of operations for manipulating probability distributions. These basic operations on probability distributions have been incorporated into the relational systems by means of extended relational algebras [2, 8]. Such extended relational algebras generally contain two main operations: `marginalization` and `product-join`, which we describe below using the BN shown in Figure 13.

- **The product-join operation** ($\overset{*}{\bowtie}$) is defined on two functional relations s and r as follows:
  $s \overset{*}{\bowtie} r = \pi_{s.a \cup r.a, s.f * r.f}(s \bowtie_{s.a \cap r.a} r)$
  where $s.a$ and $r.a$ represent the non-measure attributes of relations s and r. For instance, in Figure 15 we apply the product-join operation on the variables X, Y and Z of the BN given in Figure 13 to get the joint distribution $P(X, Y, Z)$.

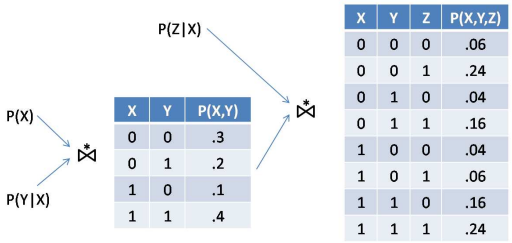

Figure 15: Calculating $P(X, Y, Z)$ with the product-join operation.

- **The marginalization operation** is defined over a joint distribution. For instance, let r represent the distribution encoded by the BN in Figure 13 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. First, note that the product-join operation is both associative and commutative. Now, to marginalize over a subset A of the joint distribution variables, we eliminate all variables not in A by applying:

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

In Figure 16, we calculate the marginal distribution $P(Y, Z)$ by eliminating the variable X from the joint distribution $P(X, Y, Z)$ in Figure 15. 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 [1] 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 16: Calculating $P(Y, Z)$ using marginalization on $P(X, Y, Z)$.

We can now 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 [2]. However, there is further opportunity for optimization of inference queries using domain specific information as discussed in this paper.

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 in Section A.1 using the DBN given in Figure 14. 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.

A.3 Learning Bayesian Networks

While the learning of network parameters (i.e., the conditional distributions) and structure is beyond the scope of this paper, we briefly outline how this process is performed, in general and in our study, for completeness and repeatability.

Learning the BN Structure: The structure learning problem is to find a BN structure that closely matches a given dataset. It is known that the space of all BN structures grows more than exponentially with the number of nodes [1]. Therefore, enumerating all BN structures is not a viable solution. In addition, one has to avoid overfitting and find relatively simple structures describing the data accurately.

The most common approaches to structure learning are the score-based methods, which are generally based on score functions that measure the goodness of fit of a BN to the data. Score-based methods often perform a hill-climbing search: they start out with an initial network structure and iteratively modify it until no further improvements are possible. At each iteration, the model is modified either by adding, deleting or reversing a link. Score-based methods differ in the score functions they use. In our work, we use the popular Bayesian Information Criterion (BIC) score [1].

Structure learning for BNs is a fundamental issue for predictive applications. However, this issue is orthogonal to our contributions in this paper: any solution for finding BN structures can be utilized in our system and all of our optimizations would still be valid.

Learning the CPDs: Given a BN structure and a complete training dataset (i.e., without missing values), the maximum likelihood principle states that the model parameters should be chosen as to maximize the likelihood of the model given the data. In this case, finding the parameters corresponds to frequency calculations on the data. For instance, for $P(T = t|Y = y, Z = z)$ we need to calculate $\frac{\#(T=t,Y=y,Z=z)}{\#(Y=y,Z=z)}$ where $\#(\cdot)$ refers to the number of instances satisfying the argument in the data. If the dataset is incomplete, the EM algorithm [1] can be used to find the parameter values. In this study, we do not consider incomplete datasets.

B. THE DYMON APPLICATION

The high-level setup for the dynamic software performance monitoring application (DyMon) is shown in Figure 17. 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 requests and the average time of processing a request).

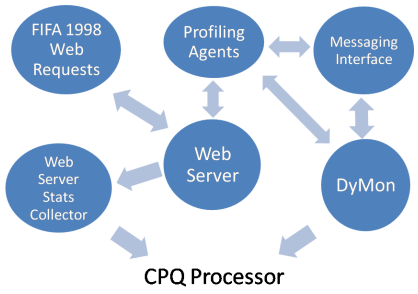

Figure 17: The setup for the DyMon application. The web server is under a load of web requests replayed from the FIFA 1998 web logs. The CPQ Processing system receives its information both from DyMon and a custom monitoring software.

C. COST MODELING FOR CPQS

In this section, we discuss the computation and storage costs for selection, product-join, and marginalization operations.

Consider an intermediate relation X 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 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 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. We do not provide it here for the interest of space and instead discuss an alternative technique below.

This method is based on the fact that the result of both product-join and the marginalization operations are distributions over a set of variables V. Given a BN for a variable set $U \supseteq V$, we can utilize the BN to estimate the size of a distribution over V with less independence assumptions. Moreover, due to the BN information the independence assumptions used in this estimation correspond to conditional independences in the data and as such are more accurate than arbitrary independence assumptions. If the variables in V are found in a single distribution in the BN, then we can find the true size of the distribution over V with a simple count operation. This technique relies on the fact that any type of distribution over a given set of variables will have the same size, e.g., $|P(X,Y)| = |P(X|Y)| = |P(Y|X)|$. If the variables in V are not found in a single distribution, then we can use counts from multiple distributions that together form V to obtain a size estimate.

D. SIZE OF THE QUERY PLAN SPACE

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.

E. ADDITIONAL EXPERIMENTAL RESULTS

Size Estimation for Intermediate Relations: Accurate size estimation for the intermediate relations is important when making materialization decisions. For the NIDS experiment, when we assumed attribute independence during size estimation, our estimation results were off up to a factor of 5.46 and 3.2 times the true value on average. When we used the counts obtained from the BN as discussed in appendix §C, however, our estimation results were always within a factor of 2.6 and within 1.73 times the true value on average. While neither method provides high accuracy, we believe this is an intrinsic problem with this domain, as one of the main reasons for using a BN is the infeasibility of computing/storing the complete joint distribution. Also observe that generally the size estimate for an intermediate relation will be worse if it has a large number of variables. However, this problem is somewhat alleviated by the fact that we do not need to have good estimates for the large joint distributions if it is already not feasible to fully materialize them due to their large size.

Computation Sharing for Range-based CPQs: For the range-based prediction queries, $P(cpu[t+k]|cpu[t], cpu[t-1])$, given in Section 2.2.2, we compared the average execution times of independent point-wise execution of the range query and the incremental execution based on the DP algorithm without materialization. The results are given in Figure 18 for different range values.

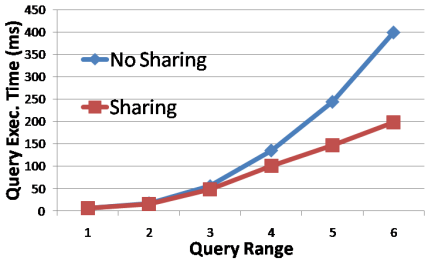

Figure 18: Average query execution times of range queries for independent execution and shared execution.

When the range of the query is short, the required computation for query execution is little hence sharing does not have much effect. However, as the range value of the query increases, the quadratic behavior of the average query execution time for independent execution is revealed. On the other hand, when the computation is shared across time points in the query range, the average query execution time increases only linearly. In the experiments presented in this paper, we always use shared computation for range queries.