Efficient Query Evaluation over Temporally Correlated Probabilistic Streams

Bhargav Kanagal [#1], Amol Deshpande [#2]

University of Maryland

[1] bhargav@cs.umd.edu

[2] amol@cs.umd.edu

February 24, 2009

Abstract

Many real world applications such as sensor networks and other monitoring applications naturally generate probabilistic streams that are highly correlated in both time and space. Query processing over such streaming data must be cognizant of these correlations, since they significantly alter the final query results. Several prior works have suggested approaches to handling correlations in probabilistic databases. However those approaches are either unable to represent the types of correlations that probabilistic streams exhibit, or can not be applied directly because of their complexity. In this paper, we develop a system for managing and querying such streams by exploiting the fact that most real-world probabilistic streams exhibit highly structured Markovian correlations. Our approach is based on the previously proposed framework of viewing probabilistic query evaluation as inference over graphical models; we show how to efficiently construct graphical models for the common stream processing operators, and how to efficiently perform inference over them in an incremental fashion. We also present an algorithm for operator ordering that judiciously rearranges the query operators to make the query evaluation tractable, if possible for the query (under certain assumptions on the data). Our extensive experimental evaluation illustrates the advantages of exploiting the structured nature of correlations in probabilistic streams.

1 Introduction

Probabilistic streams are increasingly being generated by a variety of data sources such as sensor networks [1] and other monitoring applications [2] that produce enormous amounts of uncertain data owing to low cost measurement infrastructures. Increasing use of machine learning techniques to process large amounts of real-world data also generates streams annotated with probabilities and confidence values, e.g. stock prediction models [3], habitat monitoring [4], activity recognition [5] and information extraction [6] applications. The above applications need to be able process these streams in real time for extracting vital information from them. It is therefore imperative to develop systems that can efficiently perform query processing over such streams.

We motivate our system using a habitat monitoring application that uses sensor networks to monitor the nesting environment of birds [4, 7]. In this application, cameras and other sensors capture large amounts of images, video and other valuable data which is used for studying the characteristics of birds (Figure 1). Due to the enormity of the data collected, automated computer

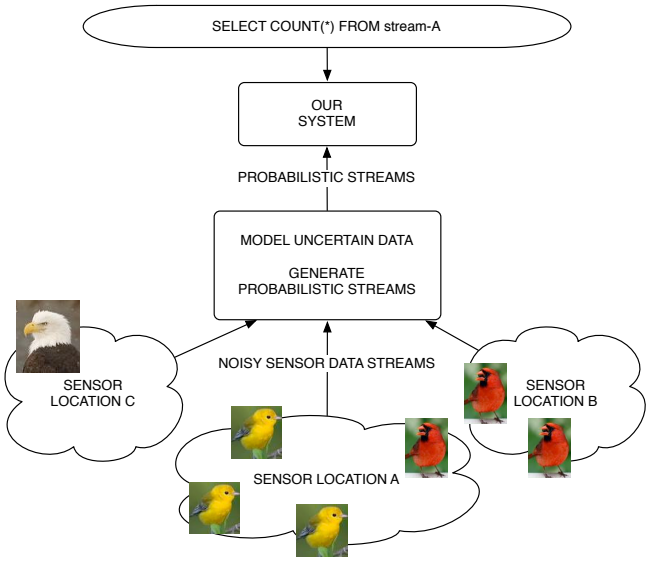


Figure 1: Probabilistic streams are generated from noisy sensor data using a combination of statistical modeling and probabilistic inferencing methods. Our system allows users to perform query processing over such streams.

vision tools are used for identifying the lists of birds from the video and image data. However, the state-of-art approaches in computer vision are not accurate enough [8] to correctly detect and identify the birds; hence most of those tools would output only a likelihood of having made an observation. This would generate a *probabilistic stream* containing birds that were observed at each location, which would then be subject to query processing and analysis by ornithologists. Query processing over such probabilistic streams, however, is challenging for several reasons.

First, the probabilistic streams generated are highly correlated both temporally (across time) and spatially (across streams from adjacent sensors). These correlations must be taken into account during query evaluation, since they can significantly alter the final query answers. For instance, consider the following query that studies the territorial characteristics of birds. Suppose we want to know the likelihood that sensor location A has more birds than another sensor location B over the last seven days. Imagine that both the locations are equally likely to have more number of birds on any given day. If we ignore the temporal correlations between the daily bird count values and assume independence between different days, then the answer for this query would be computed by multiplying the probabilities that the condition holds for every single day, with the result being $(0.5)^7 \approx 0$. However, the bird counts are highly positively correlated in a week's time (perhaps due to availability of food or the presence of predators), hence the true answer should be closer to the value 0.5.

Second, we need to be able to handle high-rate data streams efficiently, and produce accurate answers to continuous queries in real-time. Although there has been much work on data stream processing in recent years, the probabilistic nature of the streams makes this a much harder problem, since query evaluation on probabilistic data may become #P-hard even for simple queries [9].

Third, query semantics become ambiguous since we are dealing with *sequences* of tuples and not *sets* of tuples [10]. For instance, if a `SELECT *` query is posed on sensor A's data stream, we could either return (a) for each time instant, the bird that has the highest probability of being

spotted (set semantics) or (b) the sequence of birds that have the highest overall probability of being spotted (called Viterbi decoding [11]); both of these are valid interpretations for different applications.

In recent years, several approaches have been developed for managing uncertain data and for answering queries over it (see e.g., [9, 12, 13, 14, 15, 16]). Although some of these works have also proposed techniques for capturing and reasoning about the correlations, the types of correlations that can be captured is usually limited. Sen et al. [16] propose a general approach that can capture arbitrary correlations, using concepts from the *graphical models* literature. However that approach, or the other general-purpose approaches to handling correlations [17], cannot be directly applied to probabilistic streams because of their complexity; further these algorithms are not designed to be incremental, and hence cannot handle continuous queries over probabilistic streams.

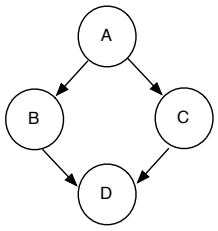
In this paper, we address the problem of efficient query evaluation over probabilistic streams. We observe that although probabilistic streams tend to be strongly correlated in space and time, the correlations are usually quite *structured*, with the same set of dependences and independences repeated across time. Furthermore, most real-world probabilistic streams are *Markovian* in nature, with the state at time "t+1" being independent of the states at previous times given the state at time "t" (in some cases, the state at time "t+1" may depend on a fixed number of states in the recent past [18]). Usually this is a result of the underlying physical process itself being Markovian in nature. For instance, if we know that a bird was present at a sensor location at time "t", then knowing that it was also present at time "t-1" doesn't give any additional information about its presence at time "t+1". In most applications, this is already encoded in the mechanism that generates the probabilistic streams; in our habitat monitoring example, the probabilistic stream would typically be generated by the application of dynamic Bayesian networks [19, 20] to sensor data, which by their nature generate Markovian and structured correlations.

We show how to exploit the knowledge of such structured correlations to efficiently evaluate queries over probabilistic streams. Our high-level approach follows the previously proposed framework of viewing probabilistic databases as graphical models, and probabilistic query evaluation as inference over graphical models [21, 22]. However, our main objective is to build a system for supporting continuous evaluation of queries against high-rate probabilistic streams, and we heavily exploit the structured, Markovian nature of the correlations for this purpose. We show how to compactly encode the correlations by decoupling the correlation structure (the set of dependencies) from the probability values. We develop techniques for incrementally building graphical models as the data arrives for a variety of query operators; our operators are designed to adhere to the *iterator* (get_next()) interface. By judiciously ordering operators, we develop polynomial time query plans for queries whenever possible for the query (under our assumptions). We also identify the queries that result in the worst-case behavior (exponential complexity), and provide polynomial time approximation strategies. Our primary research contributions can be summarized as follows:

1. We present an algebra for operating on probabilistic sequences (using the possible worlds semantics) and introduce the notion of Markov sequences.

2. We develop efficient data structures for representing Markov sequences and develop query processing techniques that exploit the repeated correlation structure.

3. We develop incremental algorithms for the query processing operators based on the get_next() framework to efficiently support streaming data.



(a) A simple DGM

A	p
0	0.3
1	0.7

B	A	p
0	0	0.5
1	0	0.5
0	1	0.2
1	1	0.8

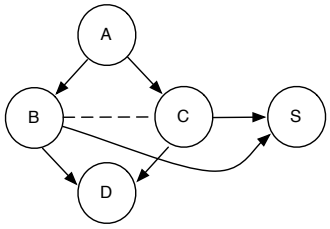
D	B	C	p
0	0	0	0.2
1	0	0	0.8
0	0	1	0.1
1	0	1	0.9
0	1	0	0.4
1	1	0	0.6
0	1	1	0.3
1	1	1	0.7

C	A	p
0	0	0.4
1	0	0.6
0	1	0.1
1	1	0.9

(b) CPDs for the DGM

gid	X
g_2	**A**
g_1	**B**
g_1	**C**
g_3	**D**

(c) Probabilistic database $\mathcal{D}$



(d) DGM for a query

S	B	C	p
0	0	0	1
1	0	0	0
0	0	1	0
1	0	1	1
0	1	0	0
1	1	0	1
0	1	1	0
1	1	1	0
2	1	1	1

(e) $p(S|B, C)$

Figure 2: Query processing over probabilistic databases using graphical models: (a) a graphical model over 4 attributes; (b) an example set of CPDs for the graphical model (bold-faced variables indicate the child nodes); (c) A probabilistic database with 4 uncertain attribute values, with correlations captured by the graphical model in (a); (d, e) to execute a query over the probabilistic database, we add new variables to the DGM and introduce additional CPDs.

4. We characterize queries that have NP-hard data complexity and provide approximation algorithms for them. Our query optimization techniques guarantee that we can find a polynomially computable query plan if one exists.

5. We have built a working system that can parse queries over probabilistic streams, and execute them efficiently. We present an extensive experimental evaluation using our system to illustrate the necessity of correlation-aware query processing over probabilistic streams, and the advantages of exploiting the Markovian nature of most probabilistic streams.

The rest of the paper is organized as follows. Section 2 provides a background on graphical models and inference. In Section 3, we specify our query semantics using a probabilistic sequence algebra and introduce Markov sequences. In Section 4, we provide algorithms for our query processing operators. Section 5 describes our query language and query optimization issues. We conclude with experiments in Section 7.

2 Background

In this section, we provide a brief overview of probabilistic graphical models and query processing in probabilistic databases using graphical models.

2.1 Graphical models

A graphical model is a space-efficient way of representing the joint probability distribution (pdf) of a set of random variables by exploiting the conditional independences that exist among them. In this paper, we focus on directed graphical models (DGM) although most of our techniques are applicable to undirected models as well. A DGM is represented using a directed graphical structure. The nodes in the graph represent random variables and the edges represent the dependencies/correlations between the random variables. Figure 2 depicts a DGM on four binary valued random variables A, B, C and D. According to this model, the value of the random variable B depends directly on the value of A. Similarly, the value of the random variable D depends *directly* on the values of B and C, but only *indirectly* on A. The dependencies among the random variables are quantified by conditional probability distribution functions (CPDs), which capture how the value of a random variable depends on the values of its parents in the graph. Thus there is a CPD for each node X in the graph, denoted by $P(X|Pa(X))$, where $Pa(X)$ denotes the parents of node X. The CPDs for our example graphical model are shown in Figure 2(b). The joint pdf across all the random variables can be computed by multiplying the CPDs of all nodes.

2.2 Probabilistic databases as DGMs

Probabilistic databases may exhibit two types of uncertainties: *tuple existence uncertainty*, which captures the uncertainty about whether a tuple exists in the database, and *attribute value uncertainty*, which captures the uncertainty about the value of an attribute. Both these types of uncertainties can be captured in a uniform manner using probabilistic graphical models [22] as follows:

- *Tuple existence uncertainty* can be modeled by using a binary random variable for each tuple, that takes value 1 if the tuple belongs to the database and 0 otherwise.
- *Attribute value uncertainty* can be modeled by using a random variable to represent the value of the attribute; a pdf on the random variable can be used to capture any form of uncertainty associated with the attribute.

Consider an example probabilistic database $\mathcal{D}$ shown in Figure 2(c) with four tuples, with uncertain attribute values (for attribute X) represented using the random variables A, B, C and D. Further, suppose the dependencies between these attributes are captured using the DGM shown in Figure 2(a). To evaluate a query over such a database, Sen et al. [21] propose adding new random variables to the graphical model to capture the intermediate result tuples generated during query processing. Consider the query `SELECT SUM(X) FROM` $\mathcal{D}$ `WHERE gid =` g_1. The query result is a single attribute, whose value is the sum of the random variables B and C (since only those tuples satisfy the selection predicate). To compute the sum, we introduce a new random variable S in the DGM that represents the sum of B and C, and we add edges from B and C to S since they determine the value of S. The exact dependence itself is captured using the CPD $p(S|B, C)$ shown in Figure 2(e); the CPD encodes the fact that S is the sum of B and C. Now, the query evaluation problem is reduced to the computation of the *marginal distribution* of the random variable S. As shown by Sen et al. [21], this is equivalent to the possible world semantics [9].

Computing the marginal distribution is a well studied problem on DGMs called *inference*. There are a number of algorithms for performing inference on DGMs such as variable elimination [23], belief propagation [24] etc. We illustrate variable elimination using the above example. To determine the marginal distribution of node S, in essence we need to perform the following computation:

$$p(S) = \sum_{A,B,C,D} p(A, B, C, D, S)$$

From the joint distribution of the random variables, we need to *sum out* (eliminate) the variables that we do not require, in this case A, B, C and D. The variable elimination algorithm takes in the order of elimination as input and sums out the variables in the order specified. The first two steps of the elimination (to eliminate A) are as follows:

$$\begin{aligned} p(S) &= \sum_{B,C,D} p(D|B,C)p(S|B,C) \underbrace{\sum_{A} p(A)p(B|A)p(C|A)}_{f(B,C)} \\ &= \sum_{B,C,D} p(D|B,C)p(S|B,C) \end{aligned}$$

The order of elimination affects the complexity of the computation. A bad ordering can potentially result in an exponential computation (in the number of nodes in the graph) while a good ordering can make inference polynomial-time computable. Some DGMs may not have a good ordering at all, in which case, the inference problem is *#P-hard*. We can visualize the reasons for such scenarios by observing the changes that occur to the DGM while eliminating variables. In the above example, when A is summed out from the expression, a new dependency between B and C is created, which is quantified by the function f in the equation. In other words, eliminating A introduces an *edge* between B and C in the graph. In general, edges are introduced between *every pair of neighbors* of the node that is being eliminated. If during the elimination process, a node gets connected to a large fraction of the other nodes in the graph, it would result in the creation of a very large joint probability distribution (possibly exponential in the number of nodes in the graph) since the dependencies between all the nodes needs to be captured.

3 Formal Query Semantics & Markov Sequences

A probabilistic sequence is defined over a set of named attributes $\mathbf{S} = \{V_1, V_2, \cdots, V_k\}$, also called its *schema*, where each V_i is a discrete probabilistic attribute with domain $dom(V_i)$. An instance of a probabilistic sequence with schema S is denoted by $S^p = \mathbf{S^1}, \mathbf{S^2}, \cdots, \mathbf{S^t}, \cdots$, where each $\mathbf{S^t} = \{v_1^t, v_2^t, \ldots, v_k^t\}$ is a set of random variables v_i^t. We use t to indicate the *time* or the *position* of $\mathbf{S^t}$ in the sequence. We abuse the notation somewhat and call $\mathbf{S^t}$ the t^{th} tuple in the sequence S^p.

We use $p(v_i^t)$ to denote the marginal probability distribution over v_i^t, $p(\mathbf{S^t})$ to denote the joint probability distribution over all variables at time t, and $p(S^p)$ to denote the joint pdf over all random variables comprising the sequence.

A probabilistic sequence is equivalent to *a set of deterministic sequences* (called *possible sequences*), where each deterministic sequence is obtained by an assignment of values to the random variables in the sequence. There are an exponential number of such possible sequences, and for each possible sequence PS_i, we associate a probability p_i that is equal to the probability that the random variables in S^p take the values given by PS_i; in essence p_i represents the likelihood that the S^p takes the value PS_i. Each deterministic sequence can be visualized as a relation with the schema $\mathbf{S}$.

3.1 Probabilistic sequence algebra

The probabilistic sequence algebra underlying our system is a probabilistic extension of the sequence algebra model proposed by Seshadri et al. [10] (with some minor changes). The result of the application of an operator on a probabilistic sequence is equivalent to applying the operator to each of the possible sequences separately, and then adding the result sequences to a *result set*. If two sequences return the same result, then we just add up the probabilities of the sequences together. The *result set* is also a set of possible sequences and is therefore a probabilistic sequence. Formally, applying operator *op* to probabilistic sequence S^p results in a probabilistic sequence $R^p = op(S^p)$ where,

$$Prob\,(R^p = x) = \sum_{PS_i \in S^p | op(PS_i) = x} p_i$$

We use this definition to extend the sequence algebra operators such as *project*, *set union*, *aggregates* to probabilistic sequences. However, two of the operators deserve further discussion:

- **Selection:** The selection operator for a sequence is different from relational selection because we cannot drop tuples in our deterministic sequence model [10]. If we drop tuples, then the sequence loses the property that the t^{th} tuple corresponds to the set $\mathbf{S^t}$. If a tuple in a deterministic sequence does not satisfy a predicate, rather than deleting the tuple, we make note that tuple does not exist by creating a new binary valued attribute A^t_P, where P is the selection predicate. A^t_P is assigned a value 1 if the tuple $\mathbf{S^t}$ satisfies the predicate and 0 otherwise. If the selection predicate is over a probabilistic attribute, then A^t_P itself would be a probabilistic attribute. We discuss this further when we present our operator algorithms.
- **Join:** We currently restrict our implementation to equi-joins on time. To join two probabilistic sequences, S^p and T^p, we compute the results of join between every possible sequence PS_i of S^p and every possible sequence PT_j of T^p; the probability of the result is the product of the probabilities of PS_i and PT_j.

Along with the standard sequence operators, we introduce two new operators specific to probabilistic streams. Both these operators take probabilistic sequences as input and return deterministic sequences as output.

1. **MAP:** The MAP operator returns the sequence in the set of possible sequences that has the highest probability.
   $MAP(S^p) = \{PS_i \in S^p | \forall PS_j \in S^p, p(PS_i) \geq p(PS_j)\}$
2. **ML:** The ML operator constructs a new deterministic sequence whose t^{th} tuple is the most likely t^{th} tuple over all the possible sequences. Suppose we denote the t^{th} tuple in the deterministic sequence D by D^t. Then, formally, $ML(S^p) = D$, where:
   $D^t = \underset{x \in X_t}{\mathbf{argmax}}\, f^t(x)$,
   where $X_t = \bigcup_{PS_i \in S^p} PS^t_i$ and $f^t(x) = \sum_{i | PS^t_i = x} p_i$

We note that the selection operator commutes with both the MAP and ML operators (Lemma 9, .3). Similarly, the join operator commutes with both selection and projection operators. A formal proof of the above is presented in the Appendix. These properties help us in designing more efficient query plans for queries. We also notice that in general, the projection operator does not

commute with the MAP and ML operators. However, for the restricted case of *Markov sequences* which we describe next, we can still establish the commutativity of the projection operator with the aggregation and the windowing operators, which is very crucial for query optimization.

3.2 Markov Sequences

Definition A *Markov Sequence*, $S^p = \mathbf{S^1}, \mathbf{S^2}, \ldots$, is a probabilistic sequence that satisfies the Markov property, i.e., the set of random variables $\mathbf{S^{t+1}}$ is conditionally independent of $\mathbf{S^{t-1}}$ given the values of the random variables in $\mathbf{S^t}$ (denoted $\mathbf{S^{t-1}} \perp\!\!\!\perp \mathbf{S^{t+1}} | \mathbf{S^t}$).

Because it obeys the Markov property, any Markov sequence is completely determined by the joint probability distributions between successive sets of random variables, $p(\mathbf{S^t}, \mathbf{S^{t+1}}), \forall t$. Therefore, we can represent a Markovian sequence as a sequence of joint probability distributions. $p(\mathbf{S^1}, \mathbf{S^2})$, $p(\mathbf{S^2}, \mathbf{S^3})$, $\ldots$, $p(\mathbf{S^t}, \mathbf{S^{t+1}})$.

Most real-world sequences obey further structure than just the Markov property. More specifically, the joint distributions between successive time slices (e.g. $p(\mathbf{S^1}, \mathbf{S^2})$ and $p(\mathbf{S^2}, \mathbf{S^3})$) typically exhibit identical conditional independences. Consider a Markov sequence S^p with schema $\{X, Y\}$: $(X^1, Y^1), (X^2, Y^2), (X^3, Y^3), \ldots, (X^t, Y^t), \ldots$. The Markov property tells us that $\{X^1, Y^1\} \perp\!\!\!\perp \{X^3, Y^3\} | \{X^2, Y^2\}$. Suppose that we have a further conditional independence: $X^1 \perp\!\!\!\perp Y^2 | \{X^2, Y^1\}$ (in other words, Y^2 is conditionally independent of X^1 given X^2 and Y^1). Because of the underlying mechanism using which the probabilistic streams are generated, it is likely that this conditional independence is exhibited at all positions (e.g. $X^2 \perp\!\!\!\perp Y^3 | \{X^3, Y^2\}$).

We can visualize such structure in a Markov sequence, if known, by representing it as a directed graphical model. Figure 3(a) shows the DGM representation of the example sequence S^p. Note that if there were no such conditional independence between the variables (i.e., if $X^1 \not\perp\!\!\!\perp Y^2 | \{X^2, Y^1\}$), then we would also have the edge $X^1 \rightarrow Y^2$ and $Y^1 \rightarrow X^2$ in the graphical representation.

3.3 Representing Markov Sequences

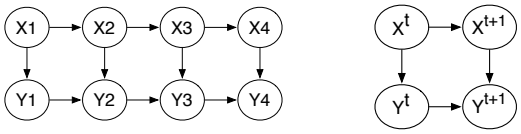
The repeating structure in the Markov sequences can be captured using a combination of two components:

- The first component, called the *schema graph*, is the DGM representation of the two step joint distribution that repeats continuously throughout the sequence.
- The second component, called *clique list*, is the set of *direct* dependencies that are present in the sequence between two successive sets of random variables.

The schema graph and the clique list of the example Markov sequence discussed above are shown in Figures 3(b,c). An assumption we make in the rest of the paper is that the number of nodes in the schema graph is a constant $2c$, i.e., c nodes per slice of the DGM.

This repeating structure also allows us to compactly represent a Markov sequence as a sequence of tuples, each of which is an ordered list of CPDs corresponding to the clique list. Furthermore, since the CPDs for each time instant and the domains of the random variables are known in advance, we can also remove the schema information and simply transmit the numbers comprising the CPDs.

For the example shown above, the list of CPDs at time t is: $\{p(X^t), p(Y^t|X^t), p(X^{t+1}|X^t), p(Y^{t+1}|X^{t+1}, Y^t)\}$. Assuming all variables are binary, we instead represent these CPDs as an array of numbers:



(a) Markov sequence S^p (b) Schema Graph

$\{(X^t), (X^t, Y^t), (X^t, X^{t+1}), (Y^t, X^{t+1}, Y^{t+1})\}$

(c) Clique List

t	$p(X^t)$
1	[0.5,0.5]
...	...

t	$p(X^{t+1}\|X^t)$
1	[0.3,0.7,0.1,0.9]
.	...

t	$p(Y^t\|X^t)$
1	[0.2, 0.8, 0.5, 0.5]
.	...

t	$p(Y^{t+1}\|X^{t+1}, Y^t)$
1	[0.7, 0.3, 0.6, 0.4 0.4, 0.6, 0.9, 0.1]

(d) Data in the relations

Figure 3: (a) Example of a Markov sequence S^p on attributes X and Y; (b) Schema graph and (c) clique list of S^p; (d) Representing S^p using one relation per CPD.

$\{p(X^t = 0), p(X^t = 1), p(Y^t = 0|X^t = 0), p(Y^t = 1|X^t = 0), \ldots, p(Y^{t+1} = 1|X^{t+1} = 1, Y^t = 1)\}$ (total 18 numbers). This allows us to efficiently transfer the tuples between the operators, and minimize the memory requirements of our operators.

Note that such a sequence of numbers can only be interpreted by entities that have the schema information about the relation to which the tuple belongs. In our system, each operator is instantiated with the schema of the tuple that it is going to receive, and only such arrays of numbers are passed around between operators in successive get_next() calls, resulting in efficient query processing over Markovian streams.

This representation also allows us to efficiently store a Markov sequence using a relational database, where we can use one relation per CPD as shown in Figure 3 (d).

3.4 Operating on Markov sequences

Since Markov sequences are a special case of probabilistic sequences, the operators defined in Section 3.1 can be used to operate upon Markov sequences (Lemma .1). However, Markov sequences are not closed under that set of operators. Several of the operators take Markov sequences as input and return non-Markovian sequences as output depending on the input schema. We formalize this observation by defining the notion of a *safe operator-input* pair.

Definition A *safe operator-input pair* is a combination of an operator and an input sequence schema such that the operator returns a Markov sequence as output when applied to a Markov sequence with the specified input schema.

Identifying safe operator-input pairs is crucial because we can evaluate such operators very efficiently; on the other hand, if an operator is not safe for an input schema, then we may have to resort to approximation schemes. Safe operator-input pairs can also be chained together with other safe pairs in a sequence to form polynomial-time query plans for complex queries. In our

query optimization framework (Section 5.2), we design an operator ordering algorithm that avoids non-safe operators as much as possible.

We remark that despite the similarity of this concept to *safe plans* [9], there are several differences between the two concepts. Safe plans were developed for probabilistic databases with only independent tuple-level uncertainty; whereas Markov sequences exhibit high degrees of correlations. Further, safe plans require global reasoning over the database schema and the query, whereas safe operator-input pairs are defined locally without any global consideration. We also note that query processing over Markov sequences is intractable even if we don't allow joins, in contrast to independent tuple databases where single relation queries are trivially answerable.

4 Operator Algorithms

In this section, we present detailed description of our algorithms for operating on Markov sequences in accordance with the semantics defined in the previous section. Our system also supports *sliding window* variants of the aggregate operators, a *pattern* operator that identifies user specified patterns in the stream, and we present the details of that as well. Our operators are designed to be incremental and treat the Markov sequence tuples as a data stream, operating on one tuple (corresponding to a joint distribution between the variables at two consecutive positions in the Markov sequence) at a time. If the operator-input pair is safe, then the output is also produced in the same fashion (a tuple at a time). If an operator-input pair is not safe, then we resort to approximations.

Each operator that we have designed implements two high-level routines: (1) a *schema routine*, which is invoked when the operator is instantiated, examines the schema of the input sequence and deduces the schema of the output sequence (to be fed to the input of the next operator); (2) a *get_next() routine* that is invoked every time a tuple is routed through this operator.

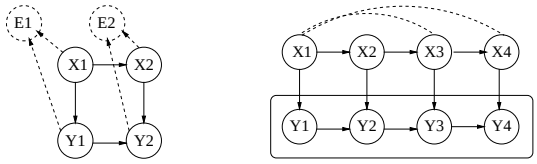
4.1 Selection

In the schema routine, we first start with the DGM corresponding to the input schema. Then we add a new node corresponding to the *exists* variable (A_P) to both time steps of the DGM. We connect this node to the variables that are part of the selection predicate through directed edges. In addition, we update the clique list of the schema to include the newly created dependencies. An illustration of this operation is shown in Figure 4(a). Here, we have as input, the Markov sequence S^p shown in Figure 3(a), and a predicate $X > Y$. We add a new node E corresponding to the new exists variable, and directed edges from X and Y to E. Also, we add (E^t, X^t, Y^t) to the clique list. In the get_next() routine, we determine the CPD of the newly created node, add it to the input tuple's CPD list and return the new tuple. A typical example of a CPD for such a case (predicate: $X > Y$) is shown in Figure 5(a).

As we can see, the algorithm does not alter the Markovian property of the sequence, and therefore every sequence can be paired safely with this operator.

4.2 Projection

In this operator, we need to remove the nodes that are not in the projection list. This corresponds to an *elimination* operation on the graphical model. To determine the schema of the output sequence, we need to determine if any new edges need to be added to the schema graph (as a result of the elimination). We do this by performing a dummy inference operation on the input



(a)Selection predicate ($X > Y$) (b) Projection is unsafe

Figure 4: (a) Executing a selection predicate ($X > Y$) entails adding new *exists* variables (E^i); the dotted edges show the changes to the schema. (b) Projection may result in a non-Markov sequence – if Y nodes are eliminated, the resulting X sequence (shown through dotted edges) is not Markov.

X^i	Y^i	E^i	f
0	0	0	1
0	0	1	0
1	0	1	1
1	0	0	0
0	1	0	1
0	1	1	0
1	1	0	1
1	1	1	0

(a) $X > Y$

G^1	X^2	G^2	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
.	.	.	.
1	1	2	1

(b)$G^2 = X^2 + G^1$

G^1	E^2	X^2	G^2	f
0	0	1	0	1
0	1	1	0	0
.	.	.	.	.
1	0	1	2	0
1	0	1	1	1
1	1	1	2	1
1	1	1	1	0
.	.	.	.	.

(c)$G^2 = X^2.E^2 + G^1$

Figure 5: Constructing CPDs for new nodes for (a) selection, (b) aggregate, and (c) aggregate with selection ($dom(X) = dom(Y) = \{0, 1\}$).

schema graph and determine the new edges to be added to the graphical model. We then derive the output sequence schema from the graphical model. In the get_next() routine, we perform the actual variable elimination procedure to eliminate the nodes that are not required.

The projection operator is *not* always *safe* for all input sequences. In certain cases, even if the input is a Markov sequence, after projection, the output may not be a Markov sequence. An example of such a sequence is shown in Figure 4(b). Here, if the nodes denoted by $Y^1, Y^2 \ldots Y^n$ are eliminated from the sequence (i.e., if Y is removed), edges will be introduced between every pair of nodes (X^i, X^j) in the graph, which results in a non-Markov sequence.

We characterize the schema of the input probabilistic sequence which results in unsafe projection as follows. Consider the connected subgraph G of the schema that contains the set of nodes E being eliminated. If there is an edge between the set of nodes E^t and E^{t+1} and there exists node $x \in vertices(G) \setminus E$, then after projection, the resulting sequence will not be Markov and the projection operation is unsafe. In Section 5.2, we present an algorithm to identify such scenarios and to postpone the projection operation if it is unsafe.

4.3 Joins

In the schema routine, we concatenate the schemas of the two sequences in order to determine the resulting output schema, i.e., we combine the schema graphs, and concatenate the clique lists. Similarly, in the get_next() routine, we concatenate the CPD lists of the two tuples, whenever both tuples have the same time value. Thus, a join can be computed incrementally, and is always safe.

4.4 Aggregation

We support decomposable aggregates, `SUM, AVG, MAX, MIN, COUNT`, in our system. When computing aggregates, the output schema is just a single attribute corresponding to the aggregate (note

(a) Case 1 (no selection predicates) (b) Case 2 (with selection predicates)

Figure 6: Illustrating aggregate computation. $G^i = Agg(X^1, X^2, \ldots, X^i)$. In each get_next() call, dotted variables are added to the DGM, and the boxed nodes are eliminated, continuously maintaining state $p(X^i, G^i)$ and $p(X^i, G^i, E^i)$ respectively. Also, note the dependence of G^i on E^i in (b).

that these are *not* sliding window aggregates, but rather one-time aggregates). The more complicated routine is the get_next() routine for which we have developed incremental algorithms. We consider two cases based on the query, the first when there are no selection predicates in the query, the second when there are selection predicates. We illustrate both cases with examples.

Case 1: No selection predicates

Consider a single attribute Markov sequence $X^1, X^2, X^3 \ldots$, and say we wish to determine the `SUM` of all the variables in the sequence, in an online fashion. Let G_k denote the sum of all the X^i's from 1 to k. The trick we use here, is to incrementally compute the distribution $p(X^i, G^i)$ as input tuples arrive. $p(X^{i+1}, G^{i+1})$ can be incrementally computed from $p(X^i, G^i)$ as follows:

$$p(X^{i+1}, G^{i+1}) = \sum_{X^i, G^i} p(X^i, G^i) p(X^{i+1}|X^i) p(G^{i+1}|G^i, X^{i+1})$$

At the end of the sequence, we get $p(X^n, G^n)$, from which we can obtain $p(G^n)$ by eliminating X^n. The DGM corresponding to this operator is shown in Figure 6 (a). The CPD $p(G^{i+1}|G^i, X^{i+1})$ is determined based on the nature of the aggregate (Figure 5(b) shows a `SUM` CPD).

Case 2: With selection predicates

When selection predicates are present, the DGM that we construct is slightly more complex. An example is shown in Figure 6(b). This is because of the presence of the E^i (exists) attributes: a value X^i contributes to the aggregate only if E^i is 1 and not otherwise. This information is added to the CPD of the aggregate node, an example of which is shown in Figure 5(c). In this case, we maintain the distribution $p(X^i, G^i, E^i)$ for all time instants i and determine the $p(X^{i+1}, G^{i+1}, E^{i+1})$ from $p(X^i, G^i, E^i)$ using a similar operation as described earlier. The DGM for doing this is shown in Figure 6(b).

In general, we have to maintain the distribution of all random variables in one time instance to enable incremental computation of aggregates. For `AVG`, we maintain the joint distribution of `SUM` and the `COUNT` aggregates for each time instant, and determine the distribution of `AVG` based on this.

The time complexity of the aggregate operator is $O(D^3)$ for `MIN` and `MAX` aggregates and $O(nD^3)$ for `SUM`, `COUNT` and `AVG` aggregates, where $D = |dom(X^i)|$ and n is the length of the sequence. This is because the domains of the G^i variables for `SUM` and `COUNT` increase as i increases ($|dom(G^n)| = nD$). Hence the CPD sizes increase resulting in high per tuple processing time as we receive more and more tuples. In order to keep the per tuple processing time for `SUM` and `COUNT` small, we use constant-time approximation algorithms for domains larger than a threshold parameter. We discuss these strategies in Section 5.3.

In addition, we also support entity based aggregates [13], for example, to determine the time instances at which a variable value was maximized. For these we compute and store the distribution of (MAX, id) at each time instant, where MAX corresponds to the maximum value and id corresponds to the time instant at which the MAX was achieved. Again, we can show that we can update this distribution incrementally.

4.5 Sliding Window Aggregates

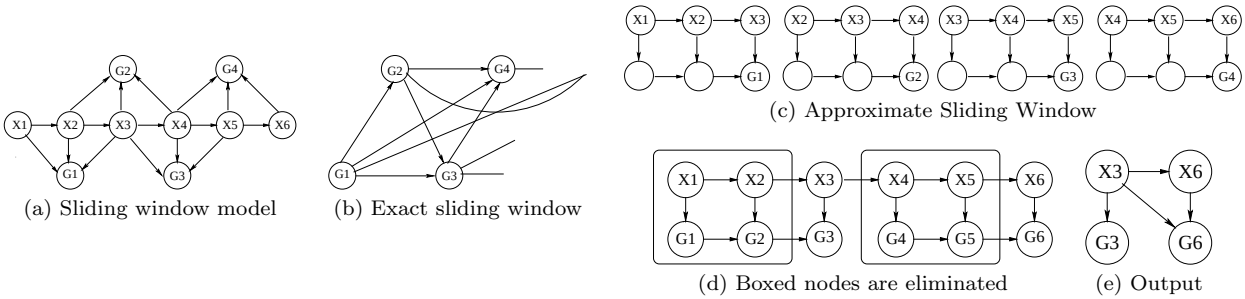
A sliding window aggregate query asks to compute the aggregate values over a window that shifts over the stream. It is characterized by the *length* of the window, the desired *shift*, and the type of aggregate. Sliding window operator is unsafe for all input sequences, since the output of the operator is always non-Markovian (illustrated in Figure 7(a) and (b), formal proof in the Appendix). This is because the aggregate value for a sliding window influences the aggregates for all of the future windows in the stream. Therefore, the exact answer to the sliding window aggregate query has exponential data complexity, which forces us to use approximations.

One approach to handling this, that we adopt, is to ignore the dependencies between the aggregate values produced at different time instances, and to compute the distribution over each aggregate value independently. We achieve this by splitting the sliding window DGM into separate graphical models (one for each window), run inference on each of them separately and compute the results. Figure 7(c) shows a simple illustration of the operation that computes the marginal probability distributions of each of the nodes G^1, G^2, G^3, G^4. The unmarked nodes in the figure denote the intermediate sums (we have used the decomposability property of our aggregates here).

However, for the special case of *tumbling windows*, where the length of the sliding window is equal to its shift, we can compute exact answers in a few cases. We use a similar trick that we used for aggregates, i.e., we maintain the distribution of all the random variables in the last step of each window. By doing so, we can guarantee that the output sequence is Markovian. However, this still requires a final unsafe projection operation; we postpone that for as long as possible, and resort to approximation when the projection must be done. As shown in Figure 7(d), we eliminate only the boxed nodes and end up with a Markovian sequence with schema shown in Figure 7(e). Eliminating X^3 and X^6 is postponed to a later projection step. When `ML` aggregates are required, the projection can be performed accurately, however, if `MAP` aggregates are required, then we resort to approximation as shown in Section 5.3.

4.6 Pattern operator

A *pattern* is a list of predicates on the attribute values, with the ordering of the predicates defining a temporal order on the sequence. For instance, $(A = 3, B > 5, A < 3)$ is a pattern that looks for a sequence of time instants such that the value of attribute A is 3 in the first instant, the next B has value more than 5, and the following A has value less than 3. We currently only handle consecutive patterns. To compute the probability of a consecutive pattern, we need to compute the product of the corresponding conditional distribution functions. If the user specifies a threshold parameter, we can prune out those time steps that do not contribute to the result. For instance if we want a pattern with probability greater than 0.7, then each of the contributing CPDs must be at least 0.7. In future, we plan to extend our system to incorporate techniques from [25] for supporting complex non-consecutive patterns.



(a) Sliding window model (b) Exact sliding window (c) Approximate Sliding Window (d) Boxed nodes are eliminated (e) Output

Figure 7: (a) DGM for sliding window aggregate. G^i's denote the aggregates that we have to compute. (b) After eliminating the X^i variables, we obtain a clique on the G^i variables, which is #P-hard. (c) Hence, we split the DGM into components as shown. Unmarked nodes are intermediate aggregates. (d) For tumbling window aggregates, we only eliminate boxed nodes to obtain the Markov sequence shown in (e). Removing nodes X^3 and X^6 is postponed to a later projection.

4.7 MAP operator

The MAP operator takes in a Markov sequence and returns a deterministic sequence. It is usually the last operator in the query plan, and hence it does not have a schema routine. The get_next() routine uses the dynamic programming based approach of Viterbi's algorithm [11] on Markov sequences to determine the sequence that has the maximum probability. We have designed and implemented an incremental version of this algorithm by maintaining appropriate state in memory. For each value in the domain, we maintain the best sequence that ends in that value. After receiving the CPD list of the new tuple, we extend each of the sequences that we have maintained in memory, by concatenating one additional value to it and computing its probability. After this, we update our memory state by recomputing the best sequences. We store these sequences in memory using a circular list of finite size. When the size of the sequence exceeds the length of the list, we remove the head of the list and continue our algorithm using the part of the sequence present in the list.

4.8 Most Likely operator

In order to determine the most likely value of a variable at each time step, we first compute the marginal probability distribution for each time instant from each tuple. Based on this, we eliminate the variables that are not required and determine the most likely values for the variables.

5 Query Evaluation

We begin with a brief discussion of our query language, and then present our overall query processing and optimization algorithms for evaluating queries over Markov sequences.

5.1 Query Syntax

In our system, queries can be specified either in an SQL-style language or it can be specified using the probabilistic sequence algebra described in Section 3. The SQL-style syntax is as follows:

```
<SELECT-MAP/ML> <Agg<attrs>>
 FROM <tables>, ..., <tname>[size,shift]
 WHERE <predicates>, <attr> like <pattern> (p)
```

The main extensions to SQL that we support are: (1) the user has the choice between using MAP or ML operators for converting the final probabilistic answer to a deterministic answer; (2) support for specifying sliding window parameters, and (3) support for pattern queries (including specifying the threshold probability).

5.2 Query Planning and Optimization

The key challenge in designing a query plan for a given query is avoiding unsafe operators. The two operators that are potentially unsafe (among the operators described in the previous section) are projection and the window aggregate operators. As we discussed above, the sliding window aggregates are always unsafe (since the output itself is of exponential size) and we only compute an approximate answer to those queries (by not computing the correlations in the output sequence). For the tumbling window operator, we separate the final projection step (which may be unsafe) into a separate projection operator. Because of this, the projection operator is the only unsafe operator in our system, and the query planning reduces to determining the correct position for the projection operators in the query plan. Next we present a sketch of our query planning algorithm.

For a given query, we first convert it to a probabilistic sequence algebra expression. We then construct the query plan by instantiating each of the operators with the schemas of their input sequences. Each operator then executes its *schema routine* and computes its output schema, which is used as the input schema for the next operator in the chain. While doing this, we also check the input to the projection, and determine if the projection operator is safe (see Section 4.2). If a projection-input pair is not safe, we pull up the projection operator through the aggregate and the windowing operators and continue with the rest of the query plan. If the operator we find after the projection is `ML`, then we can determine the exact answer, however if we find a `MAP` operator, we replace both the projection and the `MAP` operator with the approximate-MAP operator (Section 5.3) and notify the user that a safe plan cannot be found for the query. After generating a safe query plan as shown here, we optimize it in the next step.

Example Suppose that the user issues the query Q_0 : `SELECT_MAP MAX(X) FROM SEQ WHERE Y < 20` on the Markov sequence shown in Figure 3(a). The PSA expression for this query can be written as $MAP(G^p(\Pi^p_X(\sigma^p_{Y<20}SEQ)))$. While running the query planning algorithm on this query, we see that the projection operator immediately after the selection predicate is not safe (illustration is shown in Figure 4(b)). Hence, we postpone the projection and execute it after the aggregate operator, to obtain the new plan $MAP(\Pi^p_{MAX(A)}(G^p(\sigma^p_{B>2}SEQ)))$, which is now safe, because the aggregate operator returns a single value.

Our query planning algorithm is both *sound* and *complete*, i.e., we guarantee that the above procedure returns a safe plan if it exists for the query. This is trivial to see because the only reason for not finding safe plans is when the data complexity of the output sequence is *#P-hard* (which happens with unsafe projections and sliding windows). A formal proof of this fact is provided in Lemma .6 in the Appendix.

We optimize the query plan generated above by applying various rules to rearrange operators and to simplify the DGMs generated during query processing:

1. *Projection push-down:* If possible (i.e., safe), we push the projections down the query plan. For instance, if the input data streams have no temporal correlations, we can safely execute projections early on.

2. *Exploiting operator commutativity:* If we drop the probabilities (CPDs) in the tuples early, we can reduce the memory cost incurred in storing and routing tuples through the query plan; so we try to push the MAP and the ML operator down the query plan as much as possible. This is in contrast to a traditional database, where we try to push the selection predicate as far down the query plan as possible. Recall that the selection operator commutes with both MAP and ML operators (Section 3); hence we can push it down the query plan without affecting the correctness.

3. *Dropping correlations when ML values are requested:* When only the ML values are requested by the user, then we only need to determine marginal distributions for every time instant. Hence, we can drop certain edges in the DGMs of operators that will not influence query results. Suppose that a most likely sequence of the tumbling window aggregate is required by the user. In this case, we can drop edges that exist in the DGM between the first window and the second window because the most likely value sequence is not affected by these edges. For instance, in Figure 7(d), if ML values are required, we can drop the dotted edges in the Figure.

5.3 Approximation Strategies

To execute unsafe operators and to improve the throughput of the aggregate operators, we employ the following two approximation strategies:

5.3.1 Approximate MAP operator

We use the Mini-Bucket elimination algorithm of Dechter et al. [26] to approximate the MAP operator. The main idea here is to bound both the dimensionality of the CPDs and the number of CPDs generated during inference. Using this algorithm, we can bound the complexity of a potentially exponential MAP task to be polynomial in the number of tuples. We present results from using this approximation in the Section 7.

5.3.2 Approximate Aggregates

As described earlier, when the domains of the SUM and COUNT aggregates become large, the throughput of the aggregate operator falls. To counter this, we perform simple approximations beyond a threshold domain size. One such approximation for aggregates/sliding window aggregates is based on simply computing expected values. Using the *linearity of expectation*, we can compute the expected value of SUM and COUNT of aggregates in just $O(1)$ time. Our system currently does not support approximating AVG aggregate. In future, we plan to use the generating function technique of Jayram et al. [27] and techniques based on Central Limit theorem for this purpose.

6 Implementation

We have implemented a prototype system that supports querying over Markov sequences; we briefly discuss some of the salient points of the implementation. The system is implemented using Java (JDK 1.5), and we use the JavaCC parser to parse the user queries.

We store CPDs using an array of double values. To represent a distribution such as $P(X_1, X_2, X_3)$, we use an array of size $D_1 \times D_2 \times D_3$, where D_i's are the domain sizes of the variables. The value of the probability $p(X_1 = i, X_2 = j, X_3 = k)$ is given by the value in array$[iD_2D_3 + jD_3 + k]$.

Our get_next() routines are implemented to fully exploit the structured nature of Markov sequences. For this purpose, we implement a *template* data structure that stores the set of data items that each operator needs and an *instruction list*, which is a list of instructions to execute over the data structure. This data structure is created in the schema routine of the operator, with empty spaces to store the data that would be obtained from the tuple. The list of instructions to operate on these data items is also written in the schema routine. Whenever the operator receives a new tuple, these empty spaces are filled with the CPDs from the tuple and then the operator performs the execution using the instruction list.

We illustrate this using a simple example. Consider the aggregate operator with the sequence shown in Figure 6(a). In the schema routine of the aggregate operator, we realize that the DGM we need to construct has three CPDs, one CPD previously computed and stored in memory $p(X^i, G^i)$, one CPD from the input tuple i.e., $p(X^{i+1}|X^i)$ and an additional new CPD that we compute and introduce $p(G^{i+1}|G^i, X^i)$. Our template is an array of size four, the first position already filled and second and third positions will be filled when we receive the new tuple, the fourth is for storing an intermediate result.

template[0]	template[1]	template[2]	template[3]
$p(X^i\|G^i)$	New tuple	$p(G^{i+1}\|G^i, X^i)$	...

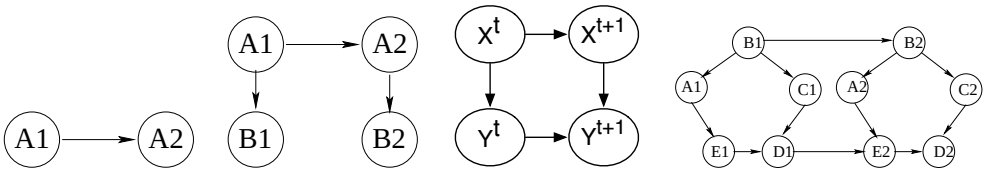
The set of instructions to perform are given as follows:

1. Multiply template[0], template[1]; Sum-out variable[0]
2. Store in template[3]
3. Multiply template[2], template[3]; Sum-out variable[1]
4. Store in template[0]

Note that on each get_next() call, the template is filled out and the same sequence of execution occurs. This way, we have effectively utilized the structure in the sequence to minimize the query processing time.

7 Experiments

In this section, we present an experimental evaluation over our prototype system. Our experimental results demonstrate that probabilistic stream query processors must incorporate support for temporal correlations, otherwise the query results can be highly inaccurate. We illustrate the effectiveness of our query processing and optimization algorithms, especially for evaluating aggregate queries over probabilistic streams. We also discuss the trade-offs between accuracy and performance for our approximate MAP operator.



(i)S_1 (ii) S_2 (iii) S_3 (iv)S_4

(a) Different schemas used in the experiments

```
Q1:  SELECT_MAP Agg(A) FROM S;
Q2:  SELECT_MAP Agg(A) FROM S WHERE B > 1;
Q3:  SELECT_MAP MAX(A) FROM S[size,size]
Q4:  SELECT_MAP MAX(A) FROM S[size,1]
Q5:  SELECT_ML A FROM S2
Q6:  SELECT_MAP X FROM S1[size], S3[size]
     WHERE S1.A > S3.Y
Q7:  SELECT_ML Agg(A) FROM S[10,10]
Q8:  SELECT_MAP X FROM S3
```

(b) Queries

Figure 8: The set of queries and the schemas used in the experiments.

7.1 Experimental Setup

Markov sequence generator:

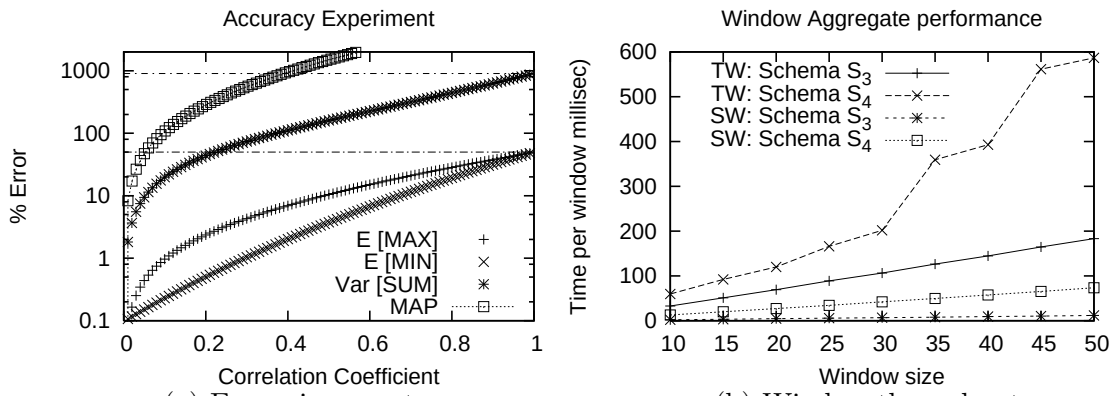
We implemented a Markov sequence generator that generates Markov sequences for a given input schema. Figure 8(a) shows the four schemas that we use in our experiments. Consider the third schema (iii) shown in Figure 8(a). For this schema, the generator starts by creating random CPDs for the first time instant - $p(X^1)$, $p(X^1, Y^1)$, $p(X^1, X^2)$ and $p(X^1, X^2, Y^2)$. For each time instant t after that, it recursively computes $p(X^t)$ and $p(X^t, Y^t)$ using the previous CPDs and then randomly generates new CPDs $p(X^t, X^{t+1})$ and $p(X^t, X^{t+1}, Y^{t+1})$ for the current time instant. By doing this, we ensure that continuity is maintained for all time instants, i.e., marginals over overlapping random variables in successive joint distributions match up. We also control the amount of spatial and temporal correlations in the sequence using a *correlation coefficient parameter*, which is input to the generator. The domains of the random variables we considered ranged from 3 ($\{0,1,2\}$) to 10 ($\{0, 1, \ldots, 9\}$).

We generated such sequences for all schemas shown in Figure 8(a). We constructed schema (iv) specifically to denote the sequence generated by our habitat monitoring application with 5 sensor locations and with complex spatial and temporal correlations. We use the notation S_i to denote the sequence generated from the corresponding schema.

Figure 8(b) shows the 8 queries that we use in our experimental evaluation.

7.2 Experimental Results

Query Processing that is aware of temporal correlations is important We generate Markov sequences based on the schema shown in Figure 8(ii) for different values of the correlation coefficient parameter f ranging from 0 to 1. We ensured that the marginal probability of variable A is $\{0.5, 0.5\}$ for all time instants. We measure the amount of error for each of our operators when



(a) Errors in operators (b) Window throughput

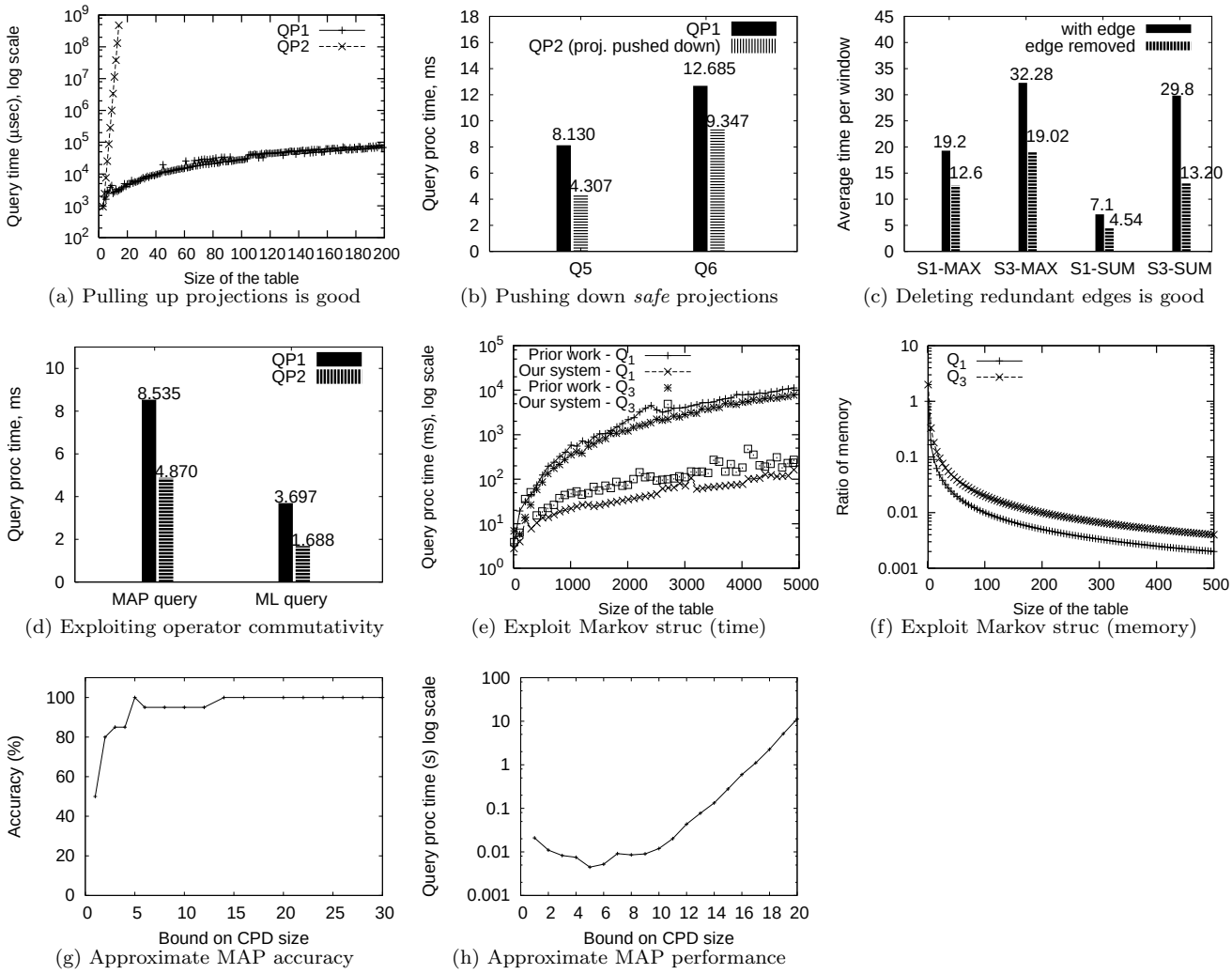
Operator	S_3		S_4	
	Q_1	Q_2	Q_1	Q_2
agg_max	2762	1201	509	220
agg_min	2802	1271	559	260
agg_sum	34.5	30.3	15.7	15.6

(c) Aggregate throughput (tuples per second)

Figure 9: (a) We plot the % error in query processing for various operators when temporal correlations are ignored, (b) We show performance (throughput) of the windowing operators, (c) We show the throughput of aggregate operators for different cases.

temporal correlations are ignored. For the MAP operator, we measure the difference between the probability of the answer returned and the correct probability (the probabilities match only when $f = 0$, i.e., when there are no correlations). We plot the error as a function of the correlation coefficient in Figure 9(a). We also measured the errors encountered for various aggregates. We plot the error in the expected values of MIN and MAX and also the errors in the variance of the SUM aggregate in the figure (expected value in SUM had 0 error as expected). The Pattern operator also suffers lot of error if temporal correlations are ignored (not shown here). As shown in the figure, for correlation coefficients beyond 0.2, the amount of error for all operators is very large if the temporal correlations are ignored.

Study of Streaming Performance of Aggregates Here, we measure the throughput of our aggregate and windowing operators. We execute queries Q_1 & Q_2 for all aggregates and measure the time take to process each new tuple completely. From this, we estimate the throughput of our aggregate function. For MAX and MIN aggregates, the amount of time taken to process new tuples is constant (as expected), however the per tuple processing time increases continuously for SUM and COUNT as the domains of intermediate results keep increasing. For SUM and COUNT, we switch to returning expected values (Section 5.3) when the domain size exceeds 200. We measure the throughput for both sequences S_3 and S_4. The results of this experiment are tabulated in Figure 9(c). As we can see from the table, our system can support up to 500 tuples/second even with our habitat monitoring schema, which is quite impressive for a system that handles temporal correlations. The value shown for SUM is the *lowest* throughput we encountered in the experiment. We can improve this number by switching to approximations earlier.



(a) Pulling up projections is good
(b) Pushing down *safe* projections
(c) Deleting redundant edges is good
(d) Exploiting operator commutativity
(e) Exploit Markov struc (time)
(f) Exploit Markov struc (memory)
(g) Approximate MAP accuracy
(h) Approximate MAP performance

Figure 10: Figures (a),(b),(c),(d) illustrate query optimization. (a),(b) show the need for determining the correct location for the projection operator. (c) demonstrates gains made by deleting redundant edges in the model. Note that this is not drawn to scale, only used for comparison. (e),(f) demonstrate advantages of our system over previous approaches. Notice that the y-axis is in the log scale, so our gains are substantial. (g),(h) describe accuracy and performance for the approximate map operator

We also measured the throughput of our tumbling window and sliding window operators. We executed queries Q_3 and Q_4 for each of sequences S_3 and S_4 as a function of the size of the window and measured the time taken to process a window of tuples. The results are shown in Figure 9(b). As we can see, the processing time increases linearly as a function of the window size. Sliding window processing takes much less time as we perform approximations (Section 5.3).

Query Optimization Strategies With this set of experiments, we demonstrate the need for query optimization and effectiveness of our query optimization strategies in choosing efficient query plans.

- **Projections:** Determining the correct position for the projection operator in the query plan is very critical to the query performance. We run query Q_0 shown in Section 5.2 with two query plans, QP1, which is obtained by our query optimizer - $\Pi_{MAX(A)}(Agg_A(TBL))$ (projections pulled up) and the naïve query plan, QP2 - $Agg_A(\Pi_A(TBL))$ (projections pushed down, as in a standard query optimizer). As Figure 10(a) shows, QP2 is extremely inefficient when compared to QP1. In fact, QP2 runs out of memory (1GB) for just 20 tuples. Also note that QP1 is incremental and works well for streams while QP2 requires all the input data at once. Pushing down *safe* projections also helps significantly improve performance (reduced data flow among operators) in many queries. We run queries Q_5 and Q_6 with query plans QP_1 and QP_2 (projections pushed down – for Q_6 we push the projection below join) and compute the total query processing time, which are plotted in the Figure 10(b).
- **Dropping edges for ML values:** As we illustrated in Section 5.2, we can drop redundant edges in the DGM to improve query performance. We run the ML query Q_7 for `SUM`, `MAX` aggregate using two query plans for each query - first one with all edges, the second with edges removed. For analysis, we use both the sequences S_1 and S_3. The query processing times are plotted in the bar chart in Figure 10(c). (We scaled down the query proc. times for `SUM` by 10 to fit the figure). As we can see, we can reduce the query processing time to about half of its value. We observed that even with the habitat monitoring sequence S_4, we reduced the query processing time by half.
- **Exploiting commutativity of operators:** We examine the amount of savings that we obtain by exploiting the operator commutativity discussed in Section 3. We execute query plans QP_1 - $ML(\sigma^p(PS))$ and QP_2 - $\sigma(ML(PS))$ and determine the amount of savings in this case. Similarly, we execute query plans $MAP(\sigma^p(PS))$ and $\sigma(MAP(PS))$. First, we verified that the answers returned by both the query plans is indeed the same. Second, we observe about 40% saving in the query processing time if we execute the ML/MAP operator first, as shown in Figure 10(d).

Comparison with previous techniques (Sen et al. [21]) We illustrate that the get_next() query processing framework that we have developed is much more efficient than previous approaches. By performing incremental operations, we can not only reduce amount of memory consumed, but also the query processing time. The advantage is magnified while computing aggregates, which create large DGMs. We use queries Q_1 and Q_3 from Figure 8 for our experiments. We run the incremental version using our system and then use techniques from Sen et al. [21] to fully construct the DGM for the query. We plot the query processing times, as a function of the table size in Figure 10(e). As we can see from the figure, the incremental algorithm is comparable with the previous approaches for small tables, but does better as the size of the table increases. We also have an order of magnitude reduction in the total memory consumed (10(f)).

Approximation Performance and Accuracy Here, we describe the trade-offs between accuracy and performance provided by the approximate MAP operator (Section 5.3). For this experiment, we use Query Q_8 and fix the correlation coefficient to be 0.5. We run the exact version of the query plan first, without any approximations. Since this is exponential, the algorithm could only handle 25 tuples before running out of memory. Then we run the query plan with the approximate-MAP operator for different bounds on the CPD size. We measure the accuracy of the results by comparing the resulting sequence against the exact MAP sequence computed earlier. The accuracy and the performance benefits of the approx-MAP operator is shown in Figure 10(g) and (h). As we can see from the plots, the approximate-MAP operator can be used to obtain fairly accurate query answers. In future, we plan to test this operator for more complex queries.

8 Related Work

Probabilistic Databases: The area of research most closely related to our work is probabilistic databases. Research work such as Mystiq [9], Trio [12], Orion [13] and others [14, 16] have developed techniques for handling tuple existence and attribute value uncertainties. Although we have adopted some techniques from this prior work, our work differs significantly from it in that we focus on efficient evaluation of continuous queries over probabilistic streams. In recent work Sen et al. [22] discuss how to exploit *shared correlation factors* to improve query processing performance. Here, the correlation factors have to be identical (including the probability values) to be able to compress the graphical model generated. This work is complementary to ours, since we only require that the *structure* of the correlations be identical, but allow for the probability values to be different.
Aggregates on Probabilistic Streams: Jayram et al. [27] and Cormode et al. [28] present algorithms for computing expected values of aggregates such as MIN, MAX, AVG etc. Our work differs from this in two aspects. First, our focus is on computing the exact probability distribution of the aggregates rather than just the expectations. We resort to approximations for the SUM and COUNT aggregates only when their domains become very large. Second, our techniques can handle the strong spatial and temporal correlations present in real-world probabilistic data streams, which this prior work ignores. In recent work, Re et al. [25] present an approach to evaluating pattern queries over correlated streams, under a correlation model quite similar to ours; however in contrast to them, we focus on evaluating general continuous queries.

9 Conclusions

There is an increasing need for building streaming databases that can efficiently query probabilistic streams that exhibit complex spatial and temporal correlations. In this paper we presented a query processing system for efficiently handling Markovian streams, which constitute a large class of naturally occurring correlated streams. We show how to exploit the structured nature of correlations in such sequences, which enables us to build an efficient query processing architecture. We also developed incremental query operator algorithms that can reuse the previous computation during query processing. Our experimental evaluation shows promising results, especially with regards to the throughputs of the operators. In future, we plan to improve the scalability of our system further by resorting to approximations with guarantees.

References

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, 2002.

[2] E. Hoke, J. Sun, and C. Faloutsos, "Intemon: Intelligent system monitoring on large clusters." in *VLDB*, 2006.

[3] P. Yoo, M. Kim, and T. Jan, "Machine learning techniques and use of event information for stock market prediction: A survey and evaluation," in *CIMCA*, 2005.

[4] A. M. Mainwaring, D. E. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring." in *WSNA*, 2002.

[5] D. Patterson, L. Liao, D. Fox, and H. Kautz, "Inferring high level behavior from low level sensors," in *UBICOMP*, 2003.

[6] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu, "Avatar information extraction system," *IEEE Data Eng. Bull.*, vol. 29, no. 1, pp. 40–48, 2006.

[7] A. Cerpa, J. Elson, D.Estrin, L. Girod, M. Hamilton, and J. Zhao, "Habitat monitoring: Application driver for wireless communications technology," in *Proceedings of ACM SIGCOMM 2001 Workshop on Data Communications in Latin America and the Caribbean.*

[8] D. Forsyth and J. Ponce, *Computer Vision.* Prentice Hall, 2003.

[9] N. N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," in *VLDB*, 2004.

[10] P. Seshadri, M. Livny, and R. Ramakrishnan, "Seq: A model for sequence databases," in *ICDE*, 1995.

[11] L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," vol. 77, pp. 257–286, 1989.

[12] J. Widom, "Trio: A system for integrated management of data, accuracy, and lineage," in *CIDR*, 2005.

[13] R. Cheng, D. V. Kalashnikov, and S. Prabhakar, "Evaluating probabilistic queries over imprecise data," in *SIGMOD*, 2003.

[14] D. Barbara, H. Garcia-Molina, and D. Porter, "The management of probabilistic data," *IEEE TKDE*, vol. 4, no. 5, pp. 487–502, 1992.

[15] P. Andritsos, A. Fuxman, and R. J. Miller, "Clean answers over dirty databases," in *ICDE*, 2006.

[16] P. Sen and A. Deshpande, "Representing and querying correlated tuples in probabilistic databases," in *ICDE*, 2007.

[17] L. Antova, C. Koch, and D. Olteanu, "From complete to incomplete information and back," in *SIGMOD*, 2007.

[18] M. Collins, "A new statistical parser based on bigram lexical dependencies," in *ACL*, 1996, pp. 184–191.

[19] B. Kanagal and A. Deshpande, "Online filtering, smoothing and probabilistic modeling of streaming data," in *ICDE*, 2008, pp. 1160–1169.

[20] K. Murphy, "Dynamic bayesian networks: Representation, inference and learnig," Ph.D. dissertation, UC Berkeley, 2002.

[21] P. Sen and A. Deshpande, "Representing and querying correlated tuples in probabilistic databases," in *ICDE*, 2007, pp. 596–605.

[22] P. Sen, A. Deshpande, and L. Getoor, "Exploiting shared correlations in probabilistic databases," in *VLDB*, 2008.

[23] N. L. Zhang and D. Poole, "Exploiting causal independence in bayesian network inference," *J. Artif. Intell. Res. (JAIR)*, vol. 5, 1996.

[24] J. Finn and J. Frank, "Optimal junction trees," in *UAI*, 1994.

[25] C. Re, J. Letchner, M. Balazinska, and D. Suciu, "Event queries on correlated probabilistic streams," in *SIGMOD*, 2008.

[26] R. Dechter and I. Rish, "Mini-buckets: A general scheme for bounded inference," *J. ACM*, vol. 50, no. 2, pp. 107–153, 2003.

[27] T. S. Jayram, A. McGregor, S. Muthukrishnan, and E. Vee, "Estimating statistical aggregates on probabilistic data streams," in *PODS*, 2007, pp. 243–252.

[28] G. Cormode and M. N. Garofalakis, "Sketching probabilistic data streams," in *SIGMOD Conference*, 2007, pp. 281–292.

APPENDIX

Lemma .1 *Every Markov sequence is also a probabilistic sequence*

Proof We prove that every Markov sequence can also be expressed as a Probabilistic Sequence. Consider a Markov sequence of length L, given by the sequence of joint probability distributions as shown below.
$\{p(X_1, X_2), p(X_2, X_3), \ldots p(X_{i=1}, X_i), \ldots p(X_{L-1}, X_L)\}$.
If D_i is the domain of the random variable X_i, construct the crossproduct $D = D_1 \times D_2 \times D_3 \ldots D_L$. For each element $(d_1, d_2, \ldots, d_L) \in D$, we determine its probability using the expression given below.

$$\begin{aligned} p(d_1, d_2, d_3, \ldots, d_L) &= P(X_1 = d_1, X_2 = d_2, \ldots, X_L = d_L) \\ &= P(X_1 = d_1) \prod_{i=1}^{L-1} \frac{P(X_i = d_i, X_{i+1} = d_{i+1})}{P(X_i = d_i)} \end{aligned}$$

Then, we construct a deterministic sequence along with the above probability for each element in D. The set of all such deterministic sequences form a probabilistic sequence. Hence, every Markov sequence is also a probabilistic sequence.

Lemma .2 $\sigma(MAP(S^p)) = MAP(\sigma^p(S^p))$

Proof We start with a probabilistic sequence

$$S^p = \{(\mathcal{R}_1, p_1), (\mathcal{R}_2, p_2), \ldots (\mathcal{R}_n, p_n)\}$$

Without loss of generality, assume that p_1 is the highest among the above n probabilities.
Suppose we first apply the selection predicate and then apply the MAP operator. In that case, we first obtain the probabilistic sequence $\{(\mathcal{R}'_1, p_1), (\mathcal{R}'_2, p_2), \ldots (\mathcal{R}'_n, p_n)\}$, where each relation $\mathcal{R}'_i$ has an additional binary attribute (when compared with $\mathcal{R}_i$) that denotes whether the tuple satisfies the predicate. Note here that the probability of the deterministic sequence $\mathcal{R}'_i$ is same as its probability before the application of the selection operator, which is p_i. The MAP operator then selects the element in this set with highest probability, i.e, $\mathcal{R}'_1$ in our case. Now suppose that we apply the MAP operator first, in this case, it selects the deterministic sequence ($\mathcal{R}_1$). Now, the selection predicate is applied to this sequence, which results in the sequence $\mathcal{R}'_1$, which is the same as the answer we obtained before. Hence, the commutativity holds.

Lemma .3 $\sigma(ML(S^p)) = ML(\sigma^p(S^p))$

Proof We start with a probabilistic sequence

$$S^p = \{(\mathcal{R}_1, p_1), (\mathcal{R}_2, p_2), \ldots (\mathcal{R}_n, p_n)\}$$

We will compute the RHS and LHS of the lemma and show their equivalence as before. Suppose we first apply the ML operator on the above sequence. We obtain a deterministic sequence $(\mathcal{R}, 1)$ as defined in Section 3 – computes the most likely tuple for each time instant in the sequence. Suppose the most likely tuple for the ith time instant is t_i. Now suppose that we apply the selection predicate on this sequence, we obtain a new deterministic sequence given by $(\mathcal{R}', 1)$. Where R' has an additional binary attribute in its schema. The value of this attribute for each tuple is 1 if the tuple satisfies the selection predicate and 0 otherwise, i.e., the final output will be tuples of the form $(t_i, \sigma(t_i))$.

Now suppose that we apply the selection predicate first, and then apply the ML operator. In this case, we get a new probabilistic sequence $(S')^P = (\mathcal{R}''_i, p_i)$ which can be represented by the set $\{(\mathcal{R}''_1, p_1), (\mathcal{R}''_2, p_2), \ldots (\mathcal{R}''_n, p_n)\}$. For every tuple t in the original probabilistic sequence (more specifically, in the deterministic sequences of S^p), we have a tuple $(t, \sigma(t))$ in S'^p, i.e., either $(t, 1)$ if t satisfies the selection predicate or $(t, 0)$ otherwise. Note that the same tuple (either $(t, 0)$ or $(t, 1)$) appears in all the possible sequences of S'^p. Also note that the probabilities of the possible sequences remain the same even after the application of the selection operator. Therefore, after applying the ML operator, the probability of tuple $(t_i, \sigma(t_i))$ is same as the probability of the tuple t_i for all values of i. Hence, the most likely tuple for the ith instant is $(t_i, \sigma(t_i))$, which is same as the previous answer. Hence, the commutativity holds.

Lemma .4 *Joins commute with selection and projection*

Proof We prove for projections in part (a) and for selections in part (b).
(a)
Suppose we have probabilistic sequences R^p and S^p, also suppose we project attributes $A \subset S$.

Then we show that $\Pi_A^p(R^p \bowtie^p S^p) = R^p \bowtie^p \Pi_A^p(S^p)$. Suppose that $R^p \bowtie^p S^p = T^p$. We prove the above result in two parts. We first show that the possible sequences of both the LHS and the RHS are the same. Then we show that the probability of a given possible sequence is same for both LHS and RHS.
Suppose we have a possible sequence $X \in \Pi_A^p(T^p)$. This means that for some $PR_i \in R^p$ and for some $PS_j \in S^p$, we have $X = \Pi_A(PR_i \bowtie PS_j)$, where the Π and $\bowtie$ (equijoin) are relational operators. We can rewrite this expression as $PR_i \bowtie \Pi_A(PS_j)$. Clearly, the rewritten expression belongs to the RHS. Hence, X also belongs to the RHS. We can similarly prove the argument in the other direction also.
We now show that the probability of X is the same in both the RHS and LHS.

$$\begin{aligned} prob^{LHS}(X) &= \sum_{\substack{PR_i \in R^p, PS_j \in S^p \\ \Pi_A(PR_i \bowtie PS_j) = X}} p_i p_j \\ &= \sum_{PR_i \in R^p} p_i \sum_{\substack{PS_j \in S^p \\ \Pi_A(PS_j) = \Pi_A(X)}} p_j \\ &= prob^{RHS}(X) \end{aligned}$$

(b)
For selections to commute with joins, clearly the selection predicate should be over only one of the relations. We show the following result. $\sigma_P^p(R^p \bowtie^p S^p) = R^p \bowtie^p \sigma_P^p(S^p)$. Just as in part (a), given a sequence that belongs to LHS, it must belong to the RHS and vice versa. Also, as the application of the selection operator does not change the probabilities of the possible sequences, a given sequence must have the same probability in both the LHS and the RHS. Hence, both the probabilistic sequences are the same.

Lemma .5 *The output of a sliding window aggregate has exponential data complexity*

Proof We illustrate this proof with a simple example and then use these techniques to prove the general case. Consider a probabilistic sequence on a single variable X_1, X_2, ..., X_n. Suppose we have a sliding window of size 2. Denote the sliding window aggregates using S_1, S_2, ..., S_{n-1}, $S_i = X_i + X_{i+1}$.

We illustrate that for all i, j where $j > i + 1$, $S_i \not\perp\!\!\!\perp S_j | S_{i+1}, S_{i+2}, \ldots, S_{j-1}$. Consider S_1 and S_3. A necessary and sufficient statistics to determine the value of S_3 is any of X_2 or X_3, (Because the distribution on X_3 and X_4 can be computed exactly from the knowledge of either X_2 or X_3, from which we can determine the distribution on S_3) Given the knowledge of S_2 which is equal to $X_2 + X_3$, we cannot determine the value of either X_2 or X_3 which would help us to determine the value of the window aggregate S_3. Hence, S_1 would still influence the value of S_3. Now, consider any arbitrary S_i and S_j such that $j > i + 1$. Given the values of every window between them, i.e., S_{i+1} to S_{j-1}, we get a total of $j - i - i$ linear equations in $j - i$ variables, of the form $S_k = X_k + X_{k+1}$ from which we cannot obtain a sufficient statistic for S_j. Hence, $S_i \not\perp\!\!\!\perp S_j$ for all values of $j > i + 1$. Even for a sliding window of length $L > 2$, we observe the same scenario, where the number of equations is fewer than the number of variables, In order to represent the dependencies that exist between all the n sliding window aggregates exactly, we need a distribution of size that is exponential in n.

Lemma .6 *Assuming that the number of variables per slice is a constant, our query planning algorithm is sound and complete*

Proof We will first prove soundness, followed by a proof for completeness.
Soundness: The query planning algorithm only returns polynomially computable plans. Since we verify that every operator-input pair is *safe* in the query plan returned by the algorithm, we only return polynomially computable query plans.
Completeness: The query planning algorithm returns a polynomially computable plan if one exists for the query.
Here, we are given that a polynomially computable plan exists for a query and we need to prove that our algorithm computes such a plan. We will prove the contrapositive of the statement, i.e., if our query planning algorithm cannot find an algorithm for a query, then solving such a query is NP-hard. If our query planning algorithm fails to find a polynomial plan for a query then we can conclude that there exists a projection operator which is unsafe for the query. (It could also be the case that the query was a sliding window query, which was already shown to be NP-hard in Lemma x). Also, every safe projection is polynomially computable since the number of variables per slice is a constant, the worst case complexity for a safe projection is exponential in this constant. However, from Lemma y, we see that if the projection is unsafe, then the data complexity of the output is necessarily exponential – equivalently, the query is NP-hard to be solved exactly. Hence, the result follows.