Efficient Query Evaluation over Temporally Correlated Probabilistic Streams

Bhargav Kanagal , Amol Deshpande
University of Maryland
{bhargav,amol}@cs.umd.edu

I. Introduction

Continuous streams of probabilistic data naturally arise in many application domains including sensor networks, distributed measurement infrastructures, information extraction, and pattern or event detection. Furthermore, in most cases, the probabilistic data streams are highly correlated in both time and space; such correlations can significantly impact the final query results, and hence must be taken into account during query execution. We illustrate this through an example.

Consider a *habitat monitoring application* that uses *camera sensor networks* to study *nesting environments of birds*. Computer vision and machine learning tools can be used to analyze the captured images in real-time, and subsequently determine if a bird was detected at a given nest location. However the image processing algorithms are typically not 100% accurate, and hence the result is best modeled as an uncertain, probabilistic stream. The stream naturally exhibits high *spatial* and *temporal* correlations, making query processing a challenge. Consider a query that asks us to compute the likelihood that a nest was occupied for all 7 days in a week. Also suppose that the probability of detecting a bird on any given day is approximately 0.5. If we ignore the temporal correlations 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 presence of a bird at a nest is highly positively correlated over a week's period (perhaps due to availability of food or the presence of predators), hence the true answer should be closer to 0.5.

In addition, since we are dealing with a *sequence* of tuples and not a set of tuples, SQL query semantics become ambiguous. If we pose a "`SELECT *`" query over a sensor's data stream, we could either return (a) for each time instant, the state of the nest (*occupied*, or *not occupied*) that has the highest probability or (b) the sequence of nest occupancy states that have the highest joint probability, (called Viterbi decoding [1]), both of which are valid and useful semantics.

In recent years, several approaches have been developed for managing uncertain data and for answering queries over them (see e.g. [2], [3], [4], [5]). In particular, Sen et al. [6] propose a general approach that can capture arbitrary correlations among the tuples. However, these general-purpose approaches are typically complex, and further are not designed to handle continuous queries over probabilistic streams.

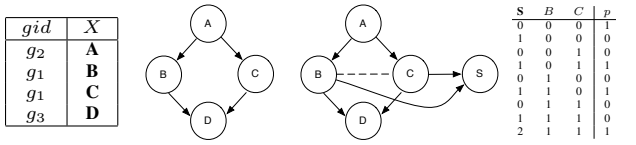
[1]This work was supported by NSF grants CNS-0509220 and IIS-0546136.

In this paper, we address the problem of efficient query evaluation over highly correlated probabilistic streams. We observe that although probabilistic streams tend to be strongly correlated in space and time, the correlations are usually quite *structured* (i.e., the same set of dependencies and independences repeat across time) and *Markovian* (i.e., the state at time "t+1" is independent of the states at previous times given the state at time "t"). We exploit this observation to compactly encode probabilistic streams by decoupling the correlation structure (the set of dependencies) from the actual probability values. We develop novel stream processing operators that can efficiently and incrementally process new data items; our operators are based on the previously proposed framework of viewing probabilistic query evaluation as inference over probabilistic graphical models (PGMs) [6]. We develop a query planning algorithm that constructs efficient query plans that are executable in polynomial-time whenever possible, and we characterize queries for which such plans are not possible. Finally we conduct an extensive experimental evaluation that illustrates the advantages of exploiting the structured nature of correlations in probabilistic streams.

II. Background

We begin with a brief description of the equivalence between probabilistic databases and PGMs. Consider the probabilistic database shown in Fig. 1(a), in which the attribute X is uncertain. Let A, B, C and D be the correlated random variables that denote the attribute values of the 4 tuples. Such a probabilistic database, along with the correlations present in it, can be naturally captured using a PGM [6]. A PGM is represented by a directed graphical structure as shown in Fig. 1(b). The nodes in the graph denote the random variables (in our case, the attribute values or the tuple existence variables) and the edges correspond to the dependencies between the random variables. The dependencies are quantified by a conditional probability distribution (CPD) function for each node that describes how the value of that node depends on the values of its parents. Fig. 1(b) represents one possible set of dependencies among the variables where B and C directly depend on A's value and directly influence the value of D. The corresponding CPDs are: $p(A), p(B|A), p(C|A)$ and $p(D|B,C)$.

To evaluate queries over such a probabilistic database, we add new random variables to the graphical model to denote the intermediate tuples that will be generated during query processing [6]. For instance, consider the query `SELECT SUM(X) FROM` $\mathcal{D}$ `WHERE gid =` g_1 which requires us to compute the



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

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

(a) ProbDB $\mathcal{D}^p$ (b) A simple PGM (c) PGM for a query (d) $P(S|B,C)$

Fig. 1. Query processing over probabilistic databases using graphical models: (a) A probabilistic database with 4 uncertain attribute values, with correlations captured by the graphical model shown in (b). (c) To execute a query over the probabilistic database, we add a new variable to the PGM and introduce an additional CPD $p(S|B,C)$ shown in (d).

sum of random variables B and C. For this, a new random variable S is introduced as shown in Fig. 1(c), that represents this sum. In addition, we introduce a CPD for S that captures how S is related to B and C (e.g., if $B = 1 = C$, then $S = 2$ with probability 1.0). The query evaluation problem is now equivalent to the computation of the *marginal distribution* of S, i.e., $p(S)$. This operation is called *inference* and we describe a simple algorithm (*variable elimination*) for it below.

To determine $p(S)$, we need to *sum out* the variables A, B, C, D from the joint distribution over the variables, i.e.:

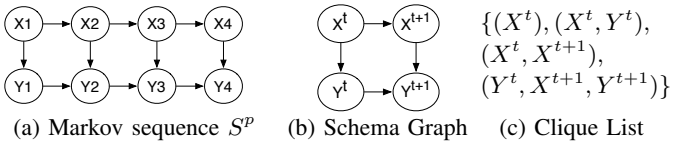
$$p(S) = \Sigma_{A,B,C,D}\ p(A,B,C,D,S)$$
$$= \Sigma_{A,B,C,D}\ p(A)p(B|A)p(C|A)P(D|B,C)(S|B,C)$$

Based on the order in which we sum-out the variables, different intermediate terms are created in the summation. For instance, if we first eliminate A, then we need to multiply the first three CPDs and sum-out A, which results in a function on B and C (not coincidentally, its neighbors). The intermediate terms can be visualized from the graph directly: eliminating a node results in creation of a function over all of its neighbors (which can be visualized as drawing edges between all pairs of neighbors, see dashed edge in Fig. 1(c)). The ordering of variables significantly affects computational complexity – a bad ordering could potentially result in an exponential-sized intermediate term (a large clique in the graph), while a good ordering may result in polynomial-time inference. There are some graphical models for which there is no good ordering, and exact inference may be infeasible. Furthermore, finding the optimal elimination order is also known to be NP-Hard.

III. Query Semantics & Markov Sequences

Definition A *probabilistic sequence* $\mathcal{S}^p$, over a set of named attributes $\mathbf{S} = \{V_1, V_2, V_3 \ldots V_k\}$ is defined as the sequence of sets $\mathbf{S^1}, \mathbf{S^2}, \mathbf{S^3} \ldots \mathbf{S^t} \ldots$, where each $\mathbf{S^t}$ is a set of random variables of the form V_i^t. In parlance with the commonly used terminology, the set $\mathbf{S}$ is called the *schema* of the probabilistic sequence, and t is used to indicate the time or the position of $\mathbf{S^t}$ in the probabilistic sequence.

A probabilistic sequence is equivalent to a *set of deterministic sequences* (called *possible sequences*), where each deterministic sequence is obtained by assigning a value to each random variable (from its domain). There are an exponential number of such possible sequences, each of which (PS_i) is associated with a probability p_i that is equal to the probability that the random variables in $\mathcal{S}^p$ take the values given by PS_i. Each possible sequence can be seen as a standard deterministic relation with schema $\mathbf{S}$. We establish an algebra of stream processing operators whose semantics are based on possible world semantics, where each possible world corresponds to a possible sequence. In addition, we introduce 2 new operators `MAP` and `ML` which convert a probabilistic sequence into a deterministic sequence. `MAP` returns the possible sequence that has the highest probability and `ML` returns a sequence of items each of which has the highest probability for its time instant. We refer the reader to the full version of the paper [7] for details of the algebra. In the rest of the paper, we focus on a special type of probabilistic sequences called "Markov" sequences. We begin by defining them.



(a) Markov sequence S^p (b) Schema Graph (c) Clique List

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

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

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

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

Fig. 2. (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.

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 $\mathbf{S^{t-1}} \perp\!\!\!\perp \mathbf{S^{t+1}} | \mathbf{S^t}$).

A Markov sequence is completely determined by the joint probability distributions between successive sets of random variables $p(\mathbf{S^t}, \mathbf{S^{t+1}}), \forall t$. Most real-world Markov sequences obey further structure in the joint probability distributions themselves. Specifically, the joint distributions exhibit identical conditional independences across different time instances. For example, consider a Markov sequence on the schema $\{X, Y\}$. We might have that $Y^{t+1} \perp\!\!\!\perp X^t | \{Y^t, X^{t+1}\}$ for all values of t. Such repeating structure can be visualized by representing the Markov sequence as a directed graphical model (Fig. 2(a)) where an identical set of edges appear between any two successive time instances, and can be captured using a combination of:

- **Schema graph:** graphical representation of the *two step joint distribution* that repeats continuously (Figure 2(b)).
- **Clique list:** the set of *direct dependencies* present between successive sets of random variables (Figure 2(c)).

The above components form the *schema* of the Markov sequence. Note that the probability values themselves would typically differ across different time instances. The repeating structure 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. Since we already know

the domains of the random variables, we can just transmit the numbers comprising the CPDs between the operators (see Figure 2(d)). Note that such a sequence of numbers can only be interpreted by entities that know the schema of the relation to which the tuple belongs. In our system, each of the operators instantiated for a query is initialized with the schema of the tuples that it is going to receive, and after that only arrays of numbers are passed around.

IV. Operating on Markov Sequences

Although we can efficiently exploit the structure in the Markov sequences to compactly encode such sequences, Markov sequences are unfortunately not closed under several common stream processing operators. We formalize this concept as follows:

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 given input schema.

It is critical to identify safe operator-input pairs because we can evaluate them incrementally and efficiently. On the other hand, if an operator is not safe for its input schema, then we cannot evaluate it incrementally, and in fact, it may be NP-hard to evaluate it exactly (see *projection* operation below). In our query optimization framework (§ V), we design an operator ordering algorithm so that we avoid unsafe operator-input pairs as long as possible to enable incremental evaluation.

Most of our key stream processing operators are designed to be incremental. They treat the input Markov sequence as a data stream, operating on one tuple (an array of numbers) at a time, and produce the output in the same fashion (which is passed to the next operator in the plan). Each operator implements two high level routines:

(1) A *schema* routine, invoked when the operator is instantiated. This routine examines the schema of the input sequence and constructs the appropriate graphical model structure corresponding to the input schema.

(2) A *get_next()* routine, which is invoked every time a tuple is routed to the operator, and performs the actual inference operation to compute the output tuple.

Next, we briefly describe the operator algorithms; for a comprehensive description, please refer to our technical report [7].

Selection: To ensure correct semantics and to preserve the Markov property for selections, we add a new node to the output schema graph corresponding to a boolean *exists* variable, which indicates whether the selection predicate is satisfied for that time instant. This variable may be uncertain if the attribute(s) involved in the selection are themselves uncertain. In the get_next() routine, we determine the CPD for the new node, add it to the input tuple, and return it.

Projection: In the projection operator, we remove the nodes that are not present in the projection list. To do this, we first determine the structure of the output schema using a dummy variable elimination operation. In the get_next() routine, we perform the actual elimination and determine the new tuple values. Projection is not safe for all input sequences, and in some cases, even if the input is a Markov sequence, the output may not be a Markov sequence. An example of such a sequence is shown in Figure 2(a). Projection on the variable Y involves eliminating the variables X_1, X_2, ..., which induces a clique on all the Y_i nodes; this output can not be represented as a 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.

Joins: We only allow joins on the *time* attribute. In the schema routine, we concatenate the schemas of the two sequences and in the get_next() routine, we simply concatenate the CPD lists of the two tuples. Thus a join can be computed incrementally and is always safe.

Aggregation: We first discuss how to evaluate aggregates that produce a single output tuple (and hence only make sense for finite streams). We support both value-based and entity-based *decomposable* aggregates on probabilistic streams – this includes most of the common aggregates including `SUM`, `COUNT`, `AVG`, `MAX` and `MIN`. When computing aggregates, the output schema is just a single attribute corresponding to the aggregate. We illustrate the get_next() routine using a simple example. Consider a single attribute Markov sequence X^1, X^2, ... and say we wish to compute the sum of all the variables in the sequence. Suppose we denote by G^k, the sum of all the X^i's from 1 to k. The trick we use here, is to incrementally compute the distribution $p(X^{i+1}, G^{i+1})$ as:

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

The probability $p(G^{i+1}|G^i, X^{i+1})$ is determined based on the nature of the aggregate, in this case it is just a truth table for `SUM` (see Fig. 1(d)). At the end of the sequence, we get $p(X^n, G^n)$ from which we obtain the desired distribution $p(G^n)$ by eliminating X^n. For a general Markov sequence with multiple attributes, we maintain the distribution of all random variables in one time instance to enable incremental computation of aggregates. For details about aggregates with predicates and entity-based aggregates, see [7].

Sliding Window aggregate: A sliding window aggregate query asks to compute an aggregate over a window that shifts over the stream; it is characterized by the length of the window, the desired shift, and the aggregate type. A sliding window aggregate is always *unsafe* for any input sequence. To handle this, by default, we approximate the output by ignoring the dependencies between the aggregate values produced at different time instants, and then compute the distribution over each aggregate independently.

For the special case of *tumbling window aggregates* (where the length of window is equal to the 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. In some cases, depending on the final operator in the query, we can do the computation exactly (see below).

MAP & ML: The final operator in a query plan is either the MAP or the ML operator, which takes in a Markov sequence and returns a deterministic sequence as output. The get_next() routine for `MAP` uses the dynamic programming based approach of Viterbi's algorithm [1] on Markov sequences to determine the *sequence* that has the maximum probability, while the `ML` operator returns the most likely value of the output variables for each time instant.

To fully exploit the Markovian structure, we implement our get_next() routines using a *template* data structure. It consists of: (1) an array for storing the input probability values that the operator processes, (2) an *instruction* list, which is the sequence of instructions to execute over the data structure for each tuple (the instruction list is generated in the schema routine). Every new tuple that arrives "fills" the template and the operator executes the instruction list. On each get_next() call, the template is filled out in exactly the same manner and the same sequence of executions occur. This way, we have effectively utilized the structure in the sequence to minimize both query processing time and memory footprint.

V. Query Processing

We allow users to specify queries in an SQL-style language that supports two additional constructs: (1) the user has the choice of using a MAP or an ML operator for converting the final probabilistic answer to a deterministic answer; (2) we support specifying the sliding window parameters, and also threshold probabilities for *pattern queries*.

The key challenge in designing a query plan for a given query is avoiding unsafe operator-input pairs. The two operators that are potentially unsafe (among the operators described in the prior section) are projection and sliding window aggregate operators. However, since we approximate the sliding window aggregates, and postpone the potentially unsafe portion of a tumbling window aggregate to a separate projection operator, projection is the only unsafe operator in our system, and hence query planning reduces to determining the correct position for the projection operators in the query plan.

For a given query, we first convert it to a probabilistic sequence algebra (PSA) 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 § IV). 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 ([7]) and notify the user that a safe plan cannot be found for the query. After generating a query plan, we optimize it by applying rules to: (1) rearrange operators (to exploit the commutativity of `ML` and `MAP` with selection) (2) simplify the PGM generated during query processing (by removing redundant edges from the PGM) and (3) push down safe projections. We provide proofs of validity of these rules in the longer version of our paper [7].

As an example, suppose a user issues the query Q_0 : `SELECT_MAP MAX(X) FROM SEQ WHERE Y < 20` on the Markov sequence shown in Figure 2(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. 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.

VI. Experimental Evaluation

We performed a comprehensive experimental evaluation using our prototype system. We built a Markov sequence generator to generate Markov sequences for a wide variety of complex schema graphs, exhibiting a range of correlations from *independent* to *perfectly correlated*. The query workload included queries with joins, selections, and aggregations. Our key findings were:

- Capturing and reasoning about temporal correlations is critical for obtaining accurate query answers.
- Our incremental *aggregate* and *windowing* operators are very efficient, and can process up to 500 tuples per second, for domain sizes below 200.
- Our query optimization strategy is effective at decreasing the query execution times (up to 50% reduction).
- Our get_next() query processing framework that exploits the structure in Markov sequences is much more efficient that previous generic approaches [6].

We refer the reader to the full version of this paper for the detailed and comprehensive experimental evaluation.

References

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

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

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

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

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

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

[7] B. Kanagal and A. Deshpande, "Efficient query evaluation on temporally correlated probabilistic streams," University of Maryland, Technical Report CS-TR-4916, http://www.cs.umd.edu/∼bhargav/CS-TR-4916.pdf.