Lineage for Markovian Stream Event Queries

Julie Letchner
Microsoft
jletch@microsoft.com

Magdalena Balazinska
University of Washington
magda@cs.washington.edu

ABSTRACT

Imprecise, sequential data, such as location sequences inferred from RFID/GPS, are often represented as Markovian (probabilistic, temporally-correlated) streams. Event queries, which detect instances of specific patterns in these streams, have become the standard tool for analysis of these streams; however, many data mining applications require richer information such as *how* a pattern is matched, *how long* the match is, or *what* stream elements matched specific pattern predicates. Such queries can dramatically increase the power of applications, but they cannot be answered by existing tools.

In this paper, we present novel techniques for processing the above queries on Markovian streams. Central to our approach are algorithms for computing and manipulating the *lineage* of Markovian stream event queries. We provide formal definitions and linear-time algorithms for computing lineage, which may be exponentially-sized in the length of the input stream. We additionally demonstrate the importance of flexible lineage projections, and provide definitions of, and two efficient algorithms for, these projections. We evaluate all algorithms on two real-world data sets (location from RFID and words from spoken audio), and demonstrate that lineage can greatly increase the analytical power of applications while incurring small processing overhead.

1. INTRODUCTION

Much of the digital information recorded daily by businesses, scientists, and individuals takes the form of sequences. Mobile sensors such as RFID [1] and GPS [2] are common sources of this data. Such sequence data also extends beyond location data to include smart home or environmental sensor readings, audio, video, *etc.* This data is recorded for its value to applications, from activity recognition and monitoring in smart homes [15], to RFID-based business operation optimization in factories [1], retail [3] or hospitals [17], to online multimedia search/retrieval [22].

To perform effective analysis, applications cannot query raw sensor data directly. Instead, they require access to higher-level information exposed through post-processing: location sequences inferred from GPS or RFID, activity sequences inferred from smart home readings, sentence-level transcripts inferred from raw audio, *etc.* Due to noise and/or ambiguity, this information is often imprecise—a person's location at a given time, or the phrase spoken at a given instant, may be represented as one of several possibilities [6, 20, 22]. An increasingly-common approach to exposing such imprecise sequences to applications is to model them as Markovian streams [8,13,16]. Markovian streams are probabilistic, temporally-correlated data streams. Each instant in a Markovian stream represents a distribution over possible values (e.g. which locations a person may have been in) and correlations between the values at the current and next instant (e.g. constraints relating a person's location at times t and $t + 1$, which reflect limitations on speed, physical constraints imposed by building layouts, *etc.*).

A key tool for Markovian stream processing is the *event query*, which detects instances of query patterns in a stream. An example event query is, *[Q1]: "Find all times Dr. Bob moved from his office to an exam room."* The output of an event query is a set of probabilities, one for each instant of the input stream, identifying the probability with which the query pattern is satisfied at each instant. The output is imprecise because of uncertainty in the input Markovian stream.

Although event queries are powerful, they identify only the instants at which a query pattern is matched. Many applications require additional information about *how* a pattern was matched, *when* the match began, or *which* sequence elements matched the pattern. For example, which exam room did Dr. Bob visit? How long did he take to get there? What path did he take through the hallways? We call such how/when/which queries *lineage queries*. They are vastly more powerful than standard event queries but they are not supported by existing systems. In this paper, we introduce the first techniques for processing lineage queries.

To process lineage queries on Markovian streams, we propose techniques for computing and manipulating event query *lineage*. In relational systems, lineage is an expression that maps each result tuple to the set of input tuples responsible for its existence; it may also include an explanation of the operators used to create the mapping [2]. Event query lineage, the stream equivalent of relational lineage, has been recently shown by Kimelfeld & Ré to be intractable to compute in general on Markovian streams [10]. In this paper, however, we identify a class of event queries that are common in practice and lend themselves to efficient lineage processing. We call these *unambiguous, DFA* event queries. Even for these queries, however, lineage is challenging to compute because its size is potentially exponential in the length of an input stream.

We define the lineage of an event query on a Markovian stream as (1) all subsequences in the stream that yield a pattern match

[1]Radio Frequency IDentification
[2]Global Positioning System

with non-zero probability and (2) a mapping of each subsequence element to the part of the pattern that it matches. For example, the lineage for query *Q1* would include both Dr. Bob's location at each moment from the time just before he left his office to the time he entered an exam room, and also a mapping from each location to the part of the query expression (introduced in Section 2.2) satisfied by the location. The answers to lineage queries can be read from this lineage. For example, the exam rooms that Dr. Bob visited appear directly in this lineage. We address the challenge of potentially-exponential lineage sizes by developing a compact encoding for lineage which we call a *lineage graph*, and a linear-time algorithm for constructing the graph. Enumerating the potentially-exponential number of lineage explanations is an additional challenge that we address using a standard top-k approach [4].

In addition to enumerating lineage explanations, most applications are also interested in projecting these explanations. For example, the result of a "which" query such as, *"Which room did Bob start in?"* comprises only a small piece of lineage—the identity of a particular room—with the rest of the lineage projected away. Similarly, the query *"How long did Bob take to move between locations X and Y?"* requires that all locations be projected out of the answer, leaving only a single number (duration) as output.

Importantly, projection must be applied *before* enumeration of the top-k answers because, as we demonstrate in Section 4, projection on only the top-k lineage explanations can yield dramatically incorrect results. Thus, projection must be a part of query processing (it cannot be left to applications), and it must be performed on *all* lineage explanations, which as we have already noted can number exponentially in the length of the input stream. In this paper we formally define projection and present two algorithms for performing projection efficiently by directly manipulating our compact lineage encoding, reducing the complexity from exponential to quadratic in the stream length. These algorithms extend the recent work of Kimelfeld & Ré by supporting projections on arbitrarily-placed lineage elements (prior work is limited to projections of prefixes and suffixes of lineage expressions), and by providing algorithms and evaluation for all constructs.

We demonstrate the utility and efficiency of our algorithms on real data from two domains: location (from RFID) and English sentences (from spoken audio podcasts). We show that, empirically, lineage processing adds an overhead of 25-280% percent (but usually less than 100%) to standard event query processing time—a small price to pay for the huge increase in the expressive power of lineage queries over standard event queries. We further demonstrate that judiciously choosing the correct projection algorithm can reduce projection overhead by a factor of 3.5.

Summary of Contributions

This paper is the first to demonstrate query processing techniques for lineage queries on Markovian streams. We present formal definitions, algorithms, and empirical evaluations of Markovian stream lineage computations, including flexible projections, to support these queries. Concretely, we provide:

1. A formal definition of lineage for event queries on Markovian streams, and a compact *lineage graph* representation (Sections 3.1 and 3.2).
2. Algorithms for constructing lineage graphs and enumerating the top-k lineage sequences for a query match, in time linear in the stream length (Sections 3.3 and 3.4).
3. A definition of lineage projection, and two algorithms (one basic, one optimized) for performing projection (Section 4).
4. An experimental evaluation of all algorithms on real-world streams inferred from RFID and audio data (Section 5).

2. PRELIMINARIES

In this section we review the definition of Markovian streams and event queries, and we define a new subclass of *unambiguous, DFA* event queries for which our lineage algorithms are designed (in this context, DFAs are deterministic finite automata).

2.1 Data Model: Markovian Streams

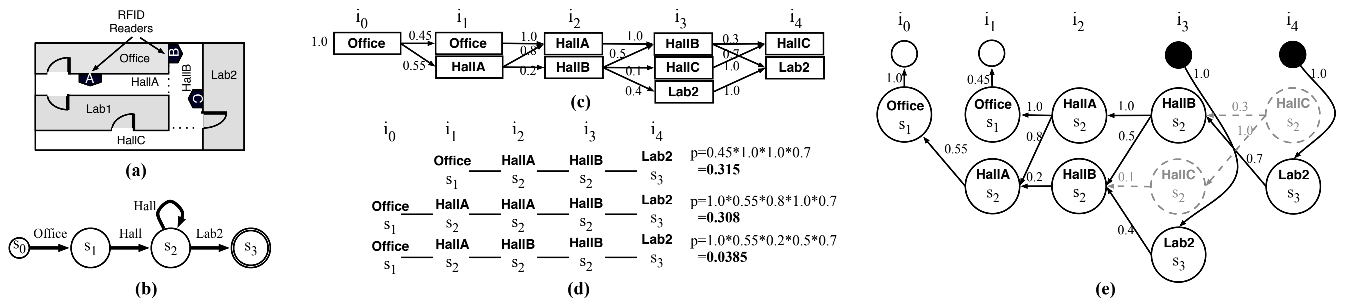
Here we present a brief overview of Markovian streams; more detailed descriptions can be found in existing work [13, 16]. A Markovian stream is an imprecise, temporally-correlated sequence. For simplicity, we consider in this paper only streams with a single dimension D (e.g. location), although generalization to multi-dimensional streams is straightforward. Each instant i in a Markovian stream defines a probability distribution p_i over the true (but unknown) value of the stream at instant i, as well as correlations (conditional probability distributions) c_i relating the stream values at instants i and $i+1$. Consider as an example Figure 1(c), which shows a Markovian stream over a location domain (i.e. derived from Bob's RFID readings). The marginal probability distribution over Bob's location at each instant is shown in boxes: for example, at instant i_0 Bob was in the Office with 100% certainty. The correlations between Bob's location at consecutive timesteps are drawn as edges: for example, *if* Bob was in the Office at i_1, then he moved to HallA at instant i_2 with 100% certainty; however, if Bob was instead in HallA at time i_1, then he moved to either HallA *or* HallB, with probability 0.8 and 0.2, respectively. Correlations are important for representing physical constraints (i.e. to avoid representing paths that involve teleporting or walking through walls).

Markovian streams are compact representations of probability distributions over an exponential number of deterministic sequences. They implement a standard possible worlds semantics in which each stream-length sequence $x \in D^N$ is a possible world [8,16]. The probability of a particular deterministic sequence is computed as $p_0 c_1 \ldots c_n$, where p_0 is the probability of the first element in the sequence and c_i is the conditional probability of the i^{th} sequence element, given the value of the previous element. For example, in Figure 1(c), the probability of the sequence Office-Office-HallA-HallB-Lab2 is $1.0 * 0.45 * 1.0 * 1.0 * 0.7 = 0.315$.

2.2 Query Model

A common tool used in Markovian stream analysis is the *event query* [16, 24], a sequence query expressed as a regular expression (equivalently, an NFA—nondeterministic finite automaton) in which alphabet symbols are replaced with predicates over the domain of an input stream. An example event query over a location domain is shown in Figure 1(b). The output of an event query Q executed on a single Markovian stream M of length n is a sequence of pairs $((i_0, p_{i_0}), \ldots, (i_n, p_{i_n}))$ indicating the probability p_i that one or more instances of the query pattern are detected in the stream, terminating at instant i. We define the probability $p_i = \sum_{x \in D^n} (x|Q@i = True)$ as the sum of the probabilities of all deterministic sequences x encoded in the input stream, in which the query pattern is satisfied at instant i. In this paper, we consider only event queries executed on a single Markovian stream.

Kimelfeld and Ré demonstrate in recent work that computing Markovian stream lineage is intractable for arbitrary event query NFAs [10]. Here, we identify a subclass of *unambiguous, DFA* event queries that are common in practice. The key property of these queries is that, on a *deterministic* input stream, they produce at most one lineage sequence ending at each instant. We exploit this property to construct the compact lineage representation and manipulations described in Sections 3 and 4.


RFID Readers
Office
HallA
HallB
Lab1
Lab2
HallC
(a)
Office
Hall
Hall
Lab2
(b)
i0 i1 i2 i3 i4
1.0 Office 0.45 Office 1.0 HallA 1.0 HallB 0.3 HallC
0.55 HallA 0.8 0.2 HallB 0.5 0.1 HallC 0.7 1.0 Lab2
0.4 Lab2 1.0
(c)
Office s1 HallA s2 HallB s2 Lab2 s3 p=0.45*1.0*1.0*0.7 =0.315
Office s1 HallA s2 HallA s2 HallB s2 Lab2 s3 p=1.0*0.55*0.8*1.0*0.7 =0.308
Office s1 HallA s2 HallB s2 HallB s2 Lab2 s3 p=1.0*0.55*0.2*0.5*0.7 =0.0385
(d)
(e)
Figure 1: (a) Sample RFID deployment. (b) Unambiguous DFA query Q. (c) Markovian stream M. (d) The set of lineage sequences describing the lineage of Q on M. for instant i_4. (e) The lineage graph describing the lineage of Q on M, for all instants.

DFA queries are those whose automaton representations are deterministic. We define a DFA query to be unambiguous in conjunction with a *deterministic* input stream as follows: a query is unambiguous on a particular deterministic input stream if the longest and shortest substrings that match the query at any given instant are the same. A query is *universally unambiguous* if is unambiguous on all possible deterministic input sequences. As an example, the query pattern Q:(RoomA, (¬RoomB)*, RoomB) is unambiguous on the input 'ab', but is ambiguous on the input stream 'aaab', in which the longest and shortest matching substrings ending at instant 3 are 'aaab' and 'ab', respectively. The ambiguity of a particular query/input pair can be detected straightforwardly at query time.

In its most simple form, a lineage query is simply an unambiguous, DFA event query. While the event query returns a single probability value for each instant in the input stream, the lineage query result contains, for each instant, a set of *lineage sequences*, which we define formally in Section 3.1 and which contain the segments of the input stream that correspond to each query match. Together, the unambiguous and DFA properties guarantee that multiple lineage sequences matching a given instant are disjoint: that is, they each correspond to a different deterministic sequence encoded in the input Markovian stream We leverage the disjointness of lineage sequences in later sections to develop a compact, Markovian structure for representing lineage.

A lineage query can optionally include a set of *projections* to be applied to the lineage query results (e.g. to identify only the identity of a particular room without regard to the hallways used to reach it). We define projection formally in Section 4. Figures 2(a-d) shows a set of unambiguous, DFA lineage queries used here as running examples, and which we also execute on real-world data as part of our experimental evaluation. We discuss the lineage shown in these diagrams, as well as Figures 2(e-h), in Sections 3 and 4, respectively. We note from experience that many real-world event queries are naturally expressed as unambiguous DFAs [23]; furthermore, most queries that do not have both properties can be easily rewritten into DFA queries with similar semantics and no ambiguity by removing repeated sequences at the beginning of the query. The ambiguous query Q, for example, can be rewritten into a similar, universally-unambiguous query Q':(RoomA, (¬(RoomA ∨ RoomB))*, RoomB) that yields the unambiguous match 'ab' on the input 'aaab'.

3. EVENT QUERY LINEAGE

In this section, we introduce Markovian stream event query lineage, which is a critical tool for answering lineage queries on Markovian streams. Current state-of-the art techniques require that these queries be answered without accessing lineage, which requires writing a separate, grounded event query to represent each possible sequence of elements—of every length, using every possible combination of predicate groundings to concrete domain elements—that can match the original event query. This approach is clearly intractable, requiring a number of queries exponential in both the input stream length and the size of the stream domain. By contrast, the lineage we define in this section admits tractable algorithms for lineage query processing.

Intuitively, we define the *lineage* of an event query on a Markovian stream as a set of lineage sequences. Each lineage sequence indicates a unique subsequence of elements in the input Markovian stream that satisfy the query, along with the path of transitions through the query DFA that are triggered by this sequence. The set of lineage sequences ending at a particular instant i together indicate the set of all possible ways in which the event query is satisfied on the input stream at instant i. For example, Figure 1(d) shows the set of lineage sequences that comprise the lineage for instant i_4 on the query and input stream shown in Figures 1(b) and (c), respectively. We revisit this Figure throughout this section.

The lineage definition we propose here is analogous to "how" lineage in a relational setting [2] because it identifies not only the set of input stream subsequences that match a query, but also the operations (DFA edge transitions) used to create a query result from each subsequence. This definition of lineage is equivalent to the output of an "indexed s-projector" as defined in recent work by Kimelfeld and Ré [10], who identify this lineage format as the only tractable type of the many alternatives they explore.

3.1 Formal Lineage Definition

We first define Markovian stream lineage formally in a deterministic setting, and then broaden the definition to cover imprecise (specifically, Markovian) streams. In both settings, we define lineage in terms of a single stream instant i. In a deterministic setting, the input stream is a single sequence of domain elements $(d_0, \ldots d_N)$. The lineage $L_Q^M(i)$ for input stream M, unambiguous DFA event query Q, and instant i is either empty, or is a single, contiguous sequence $l = (i_s, e_{i-n+1}, \ldots, e_i)$ comprising a list of n ⟨ instant, domain-element, DFA-state ⟩ triples e preceded by a single element i_s indicating the instant at which the sequence begins. The instant i_s is redundant with the instant contained in the element e_{i-n+1}, but is required for performing projection (Section 4). The number n is the number of stream instants contributing to the lineage sequence, which has length $n + 1$ due to the inclusion of i_s.

In a Markovian stream setting, the lineage $L_Q^M(i)$ for instant i has two parts: a *set* of j lineage sequences $\{l_0 \ldots l_j\}$, and a set of probability assignments $\{p(l_0), \ldots p(l_j)\}$ defining a probability $p(l)$

for each sequence l in the set. Each probability $p(l)$ is the sum of the probabilities of all deterministic sequences in the input Markovian stream that generate lineage sequence l (by definition, each deterministic input sequence can yield at most one lineage sequence ending at each instant; however, multiple deterministic input sequences can yield the same lineage sequence, derived from a shared subsequence with length shorter than the full input stream). Each individual lineage sequence l is defined as in the deterministic case. Figure 1(d) shows the set of lineage sequences that represent the lineage of the query and stream in Figures 1(b) and (c).

The lineage sequences in a set have three important properties: First, the sum of their probabilities equals the probability that the query is satisfied at instant i in the input stream. Second, they are disjoint (mutually exclusive). Finally, the lineage sequences in a set are also Markovian—that is, they can be compactly represented in a Markovian structure, which we discuss next.

3.2 Lineage Graphs

The lineage for instant i may contain an exponential number of sequences, which prohibits efficient enumeration. In this section we introduce a structure called the lineage graph, which compactly encodes the potentially-exponential set of lineage sequences—not just for a single instant i, but for all instants i simultaneously. The lineage graph provides a foundation for the development of tractable algorithms for computing and manipulating lineage.

Formally, the lineage graph of query Q on input stream M is a directed acyclic graph (DAG) $\langle V_Q^M, E_Q^M \rangle$, comprising vertices $v \in V_Q^M$ and edges $e \in E_Q^M$. There exists a vertex $v_d^s(i) \in V_Q^M$ for each element $e = \langle i, d, s \rangle$ appearing anywhere in the set of lineage sequences for any instant. The lineage graph additionally contains an edge $(v_d^s(i) \leftarrow v_{d'}^{s'}(i+1))$ for each pair of elements $e = \langle i, d, s \rangle$ and $e' = \langle i+1, d', s' \rangle$ that appear consecutively in *any* lineage sequence in a set. Each edge is assigned the probability of the corresponding edge in the input Markovian stream. Finally, the lineage graph contains additional "start" nodes $v_{start}(i)$ and "final" nodes $v_{final}(i)$ for each instant i at which *any* lineage sequence begins or ends. The final node at instant i is connected to any nodes at instant i associated with an accepting (final) DFA state, with probability 1.0. The start node at instant i is connected to all nodes at instant i that are associated with DFA states reachable from the start state (e.g. state s_1 for the DFA in Figure 1(b)). The probability assigned to edge $(v_{start}(i) \leftarrow v_d^s(i)$ is the marginal probability that domain element d is true at instant i. A given input stream and query uniquely determine a single lineage graph representation–thus, two different query/data pairs yield the same lineage graph if and only if they yield exactly the same lineage sequences.

Figure 1(e) shows the lineage graph for the query and input stream shown in Figures 1(b) and (c), respectively. Dashed vertices and edges in this figure are not a part of the lineage and will be addressed in Section 3.3. To illustrate lineage graph nodes, consider the node $v_{Office}^{s_1}(0)$. This node corresponds to the shared first element $e = \langle i_0, \text{Office}, s_1 \rangle$ which begins *both* lineage sequences starting at instant i_0 in Figure 1(d). To illustrate lineage graph edges, consider the edge from node $v_{HallA}^{s_2}(1)$ to $v_{Office}^{s_1}(0)$ in the example graph. This edge corresponds to the last two lineage sequences in Figure 1(d). It has probability 0.55 because 0.55 is the probability of the edge connecting the Office element at instant i_0 to the HallA element at instant i_1 in the input stream (Figure 1(c)). Edges in the lineage graph point backwards in time for convenience in enumerating lineage; their directionality is not fundamental to the graph definition.

More globally, each path in the lineage graph of Figure 1(e) that starts at a final (black) node and ends at a start (white) node represents a unique lineage sequence: thus, the lineage graph in Figure 1(e) contains non-empty lineage for instants i_3 and i_4 of the input stream. The lineage $L_Q^M(3)$ is the set of all paths/sequences that begin at the final node at instant 3. In this case there is only one such path, which spans instants 0 through 4, and has probability (1.0*0.4*0.2*0.4*1.0) = 0.032. The lineage $L_Q^M(4)$ includes three sequences, two of which end at instant 0 and one of which ends at instant 1.

For simplicity, all examples in this paper yield lineage graphs that are subsets of the input stream; however, lineage graphs are often *not* simple subsets—more sophisticated examples can be found in Letchner's doctoral thesis on Markovian stream processing [12].

Through our work with real-world lineage, we have identified and define here several characteristics of lineage graphs that have a significant impact on query performance and quality. We demonstrate the effects of these characteristics in Section 5.

Size: We define the size of a lineage graph as the number of nodes it contains. Larger lineage graphs are produced by queries that are more frequently satisfied, and/or are matched by longer subsequences of the input stream; these graphs incur higher total I/O and CPU costs during construction.

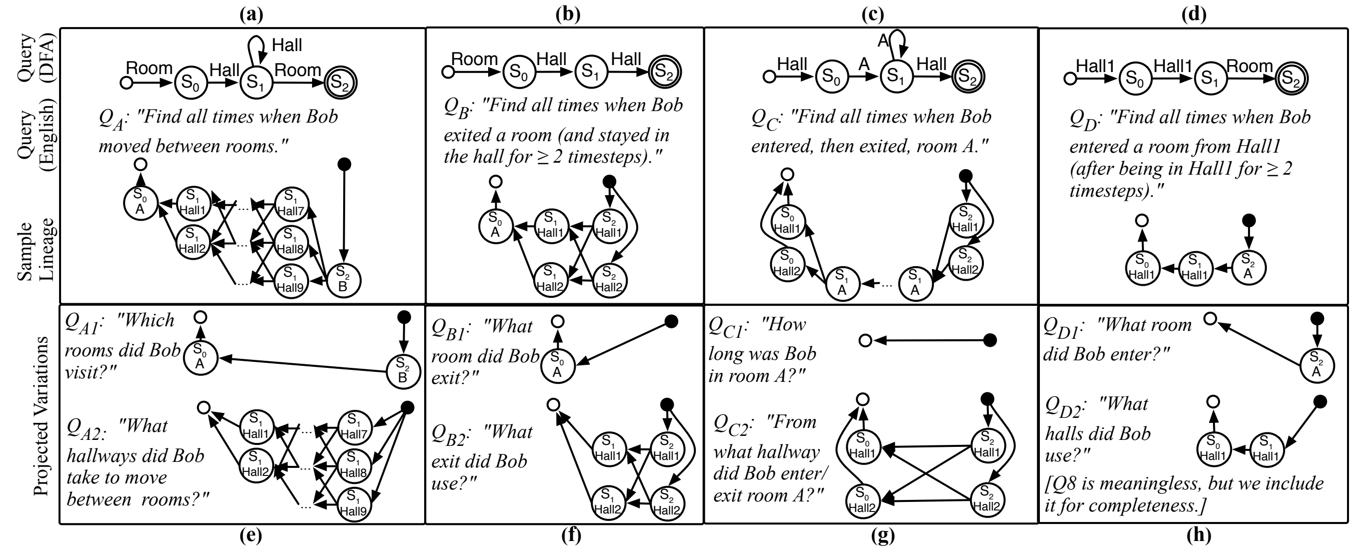
Connectivity: We define the connectivity of a lineage graph as the average degree of each of its nodes. Here, degree is the total number of edges—either incoming or outgoing—associated with a node. Highly-connected lineage graphs are generally produced by sequences or loops of unselective predicates.

Skew: We define skew in terms of a lineage graph together with a top-k value (an integer k). Skew measures the (weighted) fraction of lineage sequences that are captured in the top-k set for a given query match; thus, skew is defined *relative* to the probability of a given query match, and can be high even when the probability of the query match is low. A skew of 1.0 indicates that all lineage paths are contained in the top-k set (for each instant), while decreasing values indicate that smaller fractions of the lineage mass are captured. Skew is a measure of the utility of a particular value of k for a given stream and query.

3.3 Lineage Graph Construction

Construction of the lineage graph $\langle V_Q^M, E_Q^M \rangle$ occurs in two phases. In the first phase, the input Markovian stream M is scanned from beginning to end. As instant i of the input stream is read, a superset of the lineage nodes $v(i)$ and lineage edges $(v(i-1), v(i))$ are added to the lineage graph, according to the set of DFA transitions activated by the Markovian stream instant being processed. We call the resulting graph the pre-lineage graph, denoted $\langle pV_Q^M, pE_Q^M \rangle$. The pre-lineage graph contains the lineage graph in its entirety, but may also contain additional nodes and/or edges that are not part of the final lineage graph. These additional nodes and edges are the result of partial query matches that produce lineage histories for only part of a DFA, but never reach an accepting (final) state. They are drawn as dashed elements in Figure 1(e).

The second phase of lineage graph construction is a backward pass over the pre-lineage graph constructed in the first phase, in which the "dead-end" branches of the lineage graph, created from partial query matches, are identified and removed. These dead-end elements are identifiable as branches not reachable from any "final" graph node. A straightforward, single-pass marking algorithm is sufficient for identifying and removing these elements. Note that this backward pass requires only the pre-lineage graph as input; it does not read or write the input Markovian stream. Furthermore, both passes of lineage graph construction are memory-efficient because they must keep only two instants' worth of the pre-lineage graph (and input Markovian stream) in memory at any given time.



(a)
(b)
(c)
(d)
Query (DFA)
Query (English)
Sample Lineage
Projected Variations
Q_A: "Find all times when Bob moved between rooms."
Q_B: "Find all times when Bob exited a room (and stayed in the hall for ≥ 2 timesteps)."
Q_C: "Find all times when Bob entered, then exited, room A."
Q_D: "Find all times when Bob entered a room from Hall1 (after being in Hall1 for ≥ 2 timesteps)."
Q_{A1}: "Which rooms did Bob visit?"
Q_{A2}: "What hallways did Bob take to move between rooms?"
Q_{B1}: "What room did Bob exit?"
Q_{B2}: "What exit did Bob use?"
Q_{C1}: "How long was Bob in room A?"
Q_{C2}: "From what hallway did Bob enter/ exit room A?"
Q_{D1}: "What room did Bob enter?"
Q_{D2}: "What halls did Bob use?"
[Q8 is meaningless, but we include it for completeness.]
(e)
(f)
(g)
(h)

Figure 2: (a-d) Example DFA event queries, English translations, and sample lineage of these queries on a location-based (RFID) input stream (not pictured). (e-h) Lineage queries corresponding to the DFA event queries in (a-d), respectively, but now including projection. Each box in the lower portion of the figure contains two different, projected variants of the query in the corresponding box in the upper portion. Projected lineage is shown along with the English translation of the projection query.

3.4 Top-K Lineage

Our strategy for answering lineage queries is a basic top-k lineage sequence enumeration approach, analogous to the top-k approaches supported in other database systems [4]. In this case, applications specify a value for k. For each instant in which lineage is requested, we return only the k lineage paths with the highest probabilities, thereby avoiding enumeration of a potentially-exponential number of lineage sequences.

In order to support top-k lineage queries, we convert the lineage graph into a top-k lineage graph, from which the top k paths for each query match can be easily enumerated. Our top-k conversion algorithm is a single-pass dynamic programming algorithm, and is equivalent to a known variant of the Viterbi algorithm which computes the k most likely paths (the standard Viterbi algorithm is defined for k=1) [7]. This algorithm begins at the first (earliest/lowest) instant in the lineage graph and moves to the last, identifying the k most likely paths out of each node at the given instant, in turn. This information can then be used to traverse/enumerate the k most likely paths out of any node; for a "final" node $v_{final}(i)$, these k paths are precisely the top-k lineage sequences for instant i. Additional details of this well-established algorithm can be found in Letchner's doctoral thesis [12].

4. LINEAGE PROJECTION

Although some queries can be answered by enumerating ranked lineage sequences directly, many queries require that parts of these sequences be *projected*, and the resulting sequences deduplicated, before such enumeration is useful. Figures 2(e-h) list examples of projection-based variations of the lineage queries in Figures 2(a-d), respectively, along with lineage graphs projected accordingly. Consider query Q_{A1} (Figure 2(e)), in which an application is interested in knowing the identities of the rooms visited by Bob. The application is *not* interested in the path Bob took through the hallways, so lineage graph elements corresponding to hallways can be projected away. In this example, lineage elements are projected based on their association with particular domain elements–in this case, hallways. In other cases, projection of lineage elements may be based on their association with particular DFA states or edges (e.g. when an application is interested in knowing only the *last* room Bob visited, but is uninterested in the identities of other rooms that appear earlier in the lineage sequences). We define these projections formally in Section 4.1.

To demonstrate the importance of projection, we executed query Q_{A1} on a real-world Markovian stream derived from RFID data, and compared the quality of the top k results obtained before versus after projection (the former is equivalent to allowing an application to perform projection as a post-processing step on the k most likely paths returned by the lineage algorithms). The top ten paths computed *after* projection covered 79% of the lineage (i.e. skew was 0.79 for k=10). The top ten paths computed *before* projection, however, covered only 1.17E-6% of the lineage (i.e. the unprojected lineage graph has poor skew), which is not enough coverage to produce meaningful query results even if an application chooses to perform its own projection on these top ten paths. To underscore this point, we note that *all* ten of the most likely paths enumerated from the unprojected lineage indicated that Bob began his journey in room A; by contrast, the top ten paths computed on the projected graph, which together cover 79% of the lineage, indicate that Bob started in room B with probability 0.57, beginning in room A with a probability of only 0.43.

We informally distinguish three types of projection queries: (1) *which*-style queries (e.g. Q_{A1}, Q_{B1}, Q_{D1}, Q_{B2} and Q_{C2}) whose answers identify individual domain elements (here, locations); (2) *when*-style queries (e.g. Q_{C1}) whose answers are a time interval; and (3) *how*-style queries (e.g. Q_{A2} and Q_{D2}) whose answers are paths or subsets of paths through the lineage graph. In this section, we present an approach to projection that enables processing of all three types of projection query by supporting projection at the level of pairs of consecutive lineage elements in a sequence.

4.1 Formal Definition

We formally capture the examples of projection introduced in the previous section with the following definition: Projection on lineage L_Q^M is a transformation that removes a given element e'

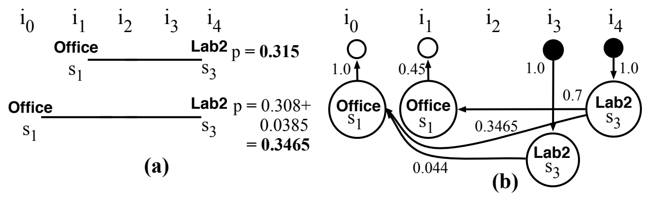


Figure 3: (a) and (b) represent the lineage sequences and graph from Figures 1(d) and (e), respectively, in which Hall elements have been projected away.

from all lineage sequences in which e' is directly preceded by a given element e: that is, projection on the pair (e, e') transforms all sequences $(\dots, s, e, e', f \dots)$ into $(\dots, s, e, f, \dots)$ by removing the element e'. The sequence suffix $(f, \dots)$ and prefix $(\dots, s)$ may be empty, but element e cannot (consequently, the start element i_s of each lineage sequence can not be projected away; we revisit this point shortly). The probability of each lineage sequence is unaltered by this transformation; however, when projection transformations yield multiple identical sequences, the sum of their probabilities is assigned to a single, deduplicated result.

The definition of projection presented here is independent of any particular representation of lineage sequences. Importantly, though, such projection can be efficiently implemented directly on a lineage graph representation, because the projection is *Markovian*: the decision about whether to project out an element e' can be made using only information about e' and its immediate predecessor, e. Markovian projections can be applied directly on a lineage graph using a quadratic-time algorithm (Section 4.2). Non-Markovian projections (e.g. projecting elements based on preceding sequences of more than one element, or based on arbitrary regular expressions) cannot be applied directly on the lineage graph.

4.2 Applying Projection

Markovian projections can be applied directly to a lineage graph because each projection transform specified by a pair of lineage elements (e, e') identifies a unique edge $(e \leftarrow e')$ in the lineage graph, to be projected out. As each projected edge is removed, connectivity of the graph is maintained using a simple, quadratic-time closure algorithm that adds edges to the graph to link together any nodes whose connectivity is disrupted by removal of a projected edge. The algorithm is thus a simple graph closure procedure.

Conceptually, lineage graph edges can be projected away in any order. However, in practice, projecting away edges starting at the beginning ($i = 0$) of the graph and moving toward the end ($i = N$) can improve performance by increasing memory locality. Because the projection process only adds edges that start at the instant *after* the edge being projected, projection in this scheme can be performed keeping only two instants of the lineage graph in memory at a time: instant i from which edges are being projected away in step one, and instant $i + 1$ to/from which edges are added/removed. All experiments in Section 5 use this sliding-window, single-pass approach to projection.

4.2.1 Optimizing Projection

Recall from Section 3.3 that construction of a lineage graph occurs in two phases: the first phase constructs a superset of the graph (the pre-lineage graph), while the second phase prunes away dead-end graph branches corresponding to partial query matches. As an alternative to applying projection to the completed lineage graph, projection can also be applied *before* the pruning phase, simultaneously with graph construction. In this approach, two instants' worth of newly-constructed graph elements are held in memory and projection is applied to them before they are written to disk (and before pruning is applied in a separate, later pass over the resulting graph). For queries where many elements are projected away, this approach can improve performance significantly by avoiding manipulation of elements that are eventually removed. Of course, the combined construction-plus-projection approach is not always superior; when significant pruning occurs, the two-phase approach is preferable since time is otherwise wasted projecting elements that will eventually be pruned away. We explore the tradeoffs of these two approaches to projection in Section 5.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our algorithms on Lahar, a prototype Markovian stream warehouse written in Java and running on a 2.0GHz Linux machine with 16GB of RAM.

5.1 Experimental Setup

We evaluate the performance of Markovian stream lineage queries on two real-world data sets, both of which are publicly released and described in additional detail on the Lahar project website [21]. The first data set comprises five 12-minute RFID traces. We show performance results for queries on one of these traces, which includes 714 instants of an individual's location, is 8.2MB in size, reflects an average of 10.6 possible locations per instant, and reflects an individual walking through an office building visiting several different offices for approximately one minute each. We test performance on this data using the queries in Figure 2.

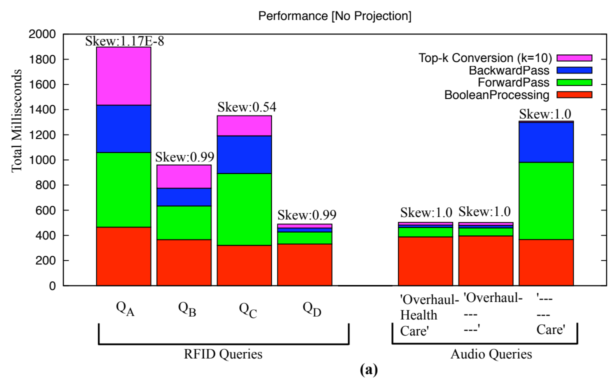
The second data set comprises four 5-minute NPR newscasts, converted into Markovian streams over spoken words. The newscast that we use to demonstrate performance here comprises 915 instants, 1.3MB, and lists, on average, 4.9 words per instant with non-zero probability. The three audio queries we use to evaluate performance are each three-word phrases: "overhaul health care", "overhaul * *", and "* * care", where the symbol "*" represents a wildcard predicate satisfied by any word. We selected these particular phrases for this evaluation because we knew them to be satisfied at least three times each in our real-world audio stream.

We selected these queries to demonstrate a range of performance costs and lineage characteristics. Although we performed our experiments on a wider range of queries and input streams, we show only a limited number of representative results here.

5.2 Performance: Enumerating Lineage

We begin by studying the performance of lineage generation and enumeration without projection. Recall that, in this case, three sequential passes over the lineage graph are required: one to generate the pre-lineage graph, one to prune dead-ends, and a final pass to compute the top k sequences. The total time required for each of these passes is shown in Figure 4 for four real-world RFID queries (left cluster) and three real-world audio queries (right cluster). Here, top-k computation was performed for k=10. For comparison, the time required to process these queries as Boolean event detection queries, without any lineage, is also shown.

Total Cost of Lineage Is Moderate: In both domains, *each* lineage-related pass takes less than twice the time of the single-pass Boolean processing algorithm, and in many cases each lineage-related pass requires only a fraction of the Boolean processing time. All queries completed in under two seconds. The total overhead of lineage processing can be as low as 10%, but is 4X in the worst case. This is true even when the lineage has size nearly equal to that of the input stream, as is the case for Q_A, or when the pre-lineage graph is larger than the input stream but is then pruned down to a small final lineage, as is the case for Q_C (lineage and input stream



	Q_A	Q_B	Q_C	Q_D
K=1	1.3E-9	0.63	0.16	0.99
K=3	3.8E-9	0.98	0.27	0.99
K=5	6.2E-9	0.99	0.36	0.99
K=10	1.17E-8	0.99	0.54	0.99

Skew

(b)

	"OHC"	"O**"	"**C"
K=1	1.0	0.88	1.0
K=3	1.0	1.0	1.0
K=5	1.0	1.0	1.0
K=10	1.0	1.0	1.0

Skew

	Q_A	Q_B	Q_C	Q_D
Pre-lineage	7625	1373	9168	237
Lineage	7455	1373	1885	135
Input stream	7563	7563	7563	7563

Graph or stream size
(number of nodes)

(c)

	"OHC"	"O**"	"**C"
Pre-lineage	20	24	9920
Lineage	20	24	20
Input stream	4526	4526	4526

Graph or stream size
(number of nodes)

Figure 4: (a) Performance of lineage processing for real-world queries on a location domain (left), and an audio domain (right). (b) Skew values computed using k=10, and (c) lineage graph sizes for these queries.

sizes are shown in Figure 4(c)). Nearly every instant of the input stream contributes one or more pre-lineage graph nodes during processing of queries Q_A and Q_C, creating maximum lineage processing overhead: these two queries are in the range of worst-case scenarios for lineage performance on queries of this length (DFAs with more states can of course incur additional overhead).

Lineage Construction Time Is Proportional To Size: The time required for the forward and backward lineage passes are approximately proportional to the size of the *pre*-lineage graph, generated by the forward pass and pruned in the backward pass. This cost can be (relatively) large even when the size of the *final* lineage is small, as can be seen in Figure 4 for queries Q_C and the "* * care" audio query, on which many nodes are generated in the forward pass only to be pruned later (actual graph sizes are shown in Figure 4(c)).

Top-K Time Is Small: The cost of the top-k computation is proportional to both the value of k and the size of the final lineage, with the size of the lineage dominating performance. Figure 4 clearly shows the correlation between lineage size and cost of top-k conversion costs: Q_A, with the largest lineage, incurs the highest cost; the mid-sized lineages of Q_B and Q_C incur moderate top-k costs; and the remaining queries with small lineage incur negligible top-k overhead. Indeed, even when k is increased to 100 (not shown), the top-k costs for queries generating mid-sized and small lineage remain similar because frequently there are fewer than 100 paths to examine per node. The only query in Figure 4(a) for which top-k costs more than double when k is increased to 100 is Q_A, whose top-k costs are 560 milliseconds for k=10 and 2833 milliseconds for k=100. This worst-case scaling of cost still increases only five-fold for a ten-fold increase in k; the scaling is sub-linear because some nodes do not have k outgoing paths and thus incur additional costs that scale less than linearly with k. We do not expect applications to use values of k near to or greater than 100.

Skew Varies By Query & Domain: Skew values for each query

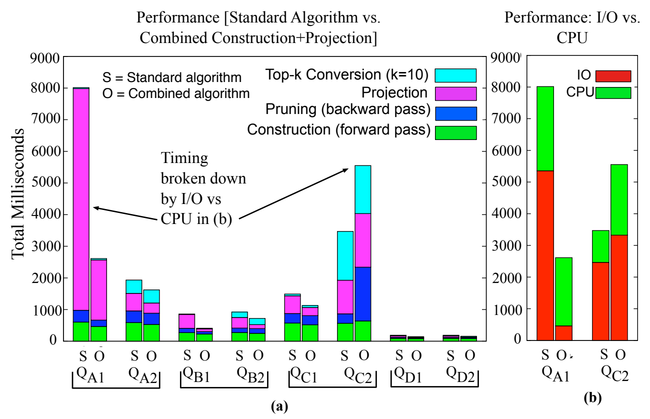


Figure 5: (a) Performance of lineage queries including projection on RFID data. (b) Performance of queries Q_{A1} and Q_{C2} broken down into I/O and CPU components.

(for k=10) are shown above the performance bars in Figure 4, and skew values for varying k are shown in Figure 4(b). In the audio domain, skew is nearly always 1.0 on our sample queries because the lineage of these queries is small and contains a minimal number (1-3) of unique paths. In the RFID domain, the queries with high skew (Q_B and Q_D) have lineage that contains more than k (=10) paths, but only a few of the paths have high probability due to skew in the input Markovian stream. By contrast, queries such as Q_A and Q_C, which include loop predicates, display poor skew. In the case of Q_A, this low skew is due to the sheer number and length of paths in the lineage. The low skew of Q_C, on the other hand, is due almost entirely to uncertainty about the starting time of each match.

Projection must be performed before top-k enumeration: Consider the skew of query Q_A in Figure 4(b): even for k=10, the skew of this query is very poor. Accurate evaluation of projection-based versions of this query (e.g. queries Q_{A1} or Q_{A2}) require that projection be performed *before* top-k enumeration, and not as a post-processing step on the top k sequences enumerated before projection. In the next Section, we demonstrate that by performing projection as part of query processing (and thus before top-k enumeration), the skew of query Q_{A1} rises to 0.79 for k=10.

5.3 Performance: Projecting Lineage

Figure 5(a) shows the performance of eight representative lineage queries that include projection (these are the queries from Figures 2(e-h)). The left bar of each pair shows query performance when separate passes are used to construct, then project, the lineage graph. The right bar of each pair shows performance when the lineage graph is constructed and projected in a single pass. We use this plot to highlight the following key points regarding projection:

Projection is practical: Figure 5(a) shows that the overhead of performing projection is manageable in practice, ranging from negligible (less than one millisecond) in the best case to a worst case of 8 seconds (without appropriate optimizations). Of the eight representative projection queries, all completed in under 8 seconds—a time which includes standard event query costs and lineage construction—and all finished in under 4 seconds using the faster of the two projection algorithms (the better choice of algorithm varies by query; we discuss this choice shortly). In theory, the quadratic complexity of projection could add a significant overhead to lineage processing. In practice, there is virtually no overhead for projected variants of queries Q_B and Q_D because their lineage is very small even before projection, and we see a significant overhead for projection on only one query, Q_{A1}. However, the benefit applications gain from incurring this overhead is a set of top-10 lineage paths that cover 79% of the lineage results, instead of the 1.2E-7% of results that can be obtained by performing projection after top-k enumeration.

Combined lineage construction+projection usually improves performance (due to I/O cost reduction): On seven out of eight of the queries in Figure 5(a), the combined construction+projection algorithm yields better performance than the multi-pass algorithm. In the case of query Q_{A1} the difference is dramatic, reducing query completion time from 8 seconds to 2.5. The performance of query Q_{A1} is also shown in the first pair of bars in Figure 5(b), where it is clear that the cost savings stems from an order-of-magnitude reduction in I/O time. A key feature of query Q_{A1} is that its lineage graph is very large before projection and very small afterward. Performing projection as the graph is constructed, in memory, avoids the necessity of writing most lineage nodes to disk (and also the cost of reading them back into memory later).

Combined lineage construction+projection can decrease performance: As the performance of query Q_{C2} shows, the combined construction+projection algorithm does not always outperform the multi-pass algorithm. The primary reason for its poor performance on query Q_{C2} is visible in Figure 5(a): a dramatic increase in the cost of the backward pass (blue bar), which is performed *after* projection in the optimized algorithm, but before projection in the standard algorithm. The optimized costs grow so much here for the combined algorithm because the projected lineage graph is very highly connected. The backward pruning pass scans a graph of approximately the same size in both algorithms (9168 vs. 9059 nodes), but the average degree of each node is 55 in the graph constructed using combined construction+projection, and only 2.15 in the unprojected graph. This 25x increase in connectivity greatly increases the CPU costs associated with pruning (Figure 5(b)).

Top-k costs are unaffected by choice of projection algorithm: This is no surprise, since top-k construction and enumeration algorithms are applied to the projected lineage graph, which is the same regardless of the algorithm used to obtain it.

6. RELATED WORK

Relational Lineage Support: Much existing work, such as the WHIPS warehouse [5] and the ProQL language [9] focuses on *deterministic, relational* lineage manipulation. Such work is not directly applicable to Markovian streams, although ProQL conceptualizes lineage as a graph, in a spirit similar to that of this paper. Probabilistic databases that support lineage include Trio [18] and MayBMS [11], but these systems do not support sequential data.

Sequence Processing: Many approaches have recently emerged for processing *imprecise* sequences, including those with Markovian correlations [8, 20, 22]. However, none of these systems track event lineage. Separately, lineage-like concepts have been explored in a number of *deterministic* stream processing systems, including SASE [24] and ZStream [14], but of course these systems are inadequate for processing Markovian or other imprecise streams.

Lineage for Imprecise Stream Processing: The work most closely related to this paper is that of Shen et al. [19] and Kimelfeld & Ré [10], who define subsequence matches of event queries on imprecise streams but do not provide algorithms or support projections beyond removal of prefixes/suffixes. Shen et al. consider only *independent* sequences, however, and Kimelfeld & Ré propose a theoretical Markov sequence transducer whose output under specific conditions (those of an "indexed s-projector") is equivalent to the lineage graph of Section 3.2, but they do not propose or evaluate algorithms for constructing or manipulating the graph to answer lineage queries. The *prDB* system proposed by Kanagal & Deshpande [8] generates lineage for *conjunctive* queries on Markov sequences, but does not consider lineage of event queries.

7. CONCLUSION

We introduced a set of Markovian stream lineage queries that provide information about *how* a query is matched in an input stream. We formally defined a lineage graph to capture this information, and provided single-pass algorithms for constructing and projecting the graph and for enumerating the top-k matching sequences. Our experiments demonstrate the practicality of this approach, which is an important step in building more powerful mobile applications and also extends to other domains.

Acknowledgements

This work was partially suported by NSF grant IIS-0713123; by gifts from Intel Research; and by Magda Balazinska's Microsoft Research Faculty Fellowship.

8. REFERENCES

[1] J. Brusey, M. Harrison, C. Floerkemeier, and M. Fletcher. Reasoning about uncertainty in location identification with rfid. In *Proc. of the 18th IJCAI Conf.*, 2003.

[2] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[3] Computerworld. Procter & Gamble: Wal-Mart RFID effort effective. `http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=284160`, Feb. 2007.

[4] G. Cormode, F. Li, and K. Yi. Semantics of ranking queries for probabilistic data and expected ranks. In *Proc. of the 25th ICDE Conf.*, 2009.

[5] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *Proc. of the 29th VLDB Conf.*, 12(1):41–58, 2003.

[6] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-based approximate querying in sensor networks. *Proc. of the 31st VLDB Conf.*, 14(4):417–443, 2005.

[7] J. Forney, G.D. The viterbi algorithm. *Proceedings of the IEEE*, 61(3), 1973.

[8] B. Kangal and A. Deshpande. Lineage processing over correlated probabilistic databases. In *Proc. of the SIGMOD Conf.*, 2010.

[9] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *Proc. of the SIGMOD Conf.*, 2010.

[10] B. Kimelfeld and C. Ré. Transducing markov sequences. In *Proc. of the 29th PODS Conf.*, pages 15–26, New York, NY, USA, 2010. ACM.

[11] C. Koch. MayBMS: A system for managing large uncertain and probabilistic databases. In *Managing and Mining Uncertain Data*. Springer-Verlag, 2009.

[12] J. Letchner. *Lahar: Warehousing Markovian Streams*. Ph.d thesis, University of Washington, 2010.

[13] J. Letchner, C. Ré, M. Balazinska, and M. Philipose. Access methods for markovian streams. In *Proc. of the 25th ICDE Conf.*, pages 246–257, 2009.

[14] Y. Mei and S. Madden. ZStream: A cost-based query processor for adaptively detecting composite events. In *Proc. of the SIGMOD Conf.*, 2009.

[15] D. J. Patterson, D. Fox, H. A. Kautz, and M. Philipose. Fine-grained activity recognition by aggregating abstract object usage. In *ISWC*, pages 44–51, 2005.

[16] C. Ré, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *Proc. of the SIGMOD Conf.*, pages 715–728, 2008.

[17] RFID Journal. Hospital gets ultra-wideband RFID. `http://www.rfidjournal.com/article/view/1088/1/1`, Aug. 2004.

[18] A. D. Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *Proc. of the 24th ICDE Conf.*, pages 1023–1032, 2008.

[19] Z. Shen, H. Kawashima, and H. Kitagawa. Probabilistic event stream processing with lineage. In *Proc. of the Data Engineering Workshop*, 2008.

[20] T. Tran, L. Peng, B. Li, Y. Diao, and A. Liu. Pods: A new model and processing algorithms for uncertain data streams. In *Proc. of the SIGMOD Conf.*, 2010.

[21] University of Washington. The Lahar Project. `http://lahar.cs.washington.edu/`.

[22] D. Wang, E. Michelakis, M. Franklin, M. Garofalakis, and J. Hellerstein. Declarative information extraction in a probabilistic database system. In *Proc. of the 26th ICDE Conf.*, 2010.

[23] E. Welbourne, M. Balazinska, G. Borriello, and J. Fogarty. Specification and verification of complex location events with panoramic. In *Pervasive*, 2010.

[24] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. of the SIGMOD Conf.*, pages 407–418, 2006.