Lahar: Warehousing Markovian Streams

Julia Maureen Letchner

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2010

Program Authorized to Offer Degree: Computer Science & Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Julia Maureen Letchner

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

____
Magdalena Balazinska

Reading Committee:

____
Magdalena Balazinska

____
Gaetano Borriello

____
Dan Suciu

Date: ______________________

University of Washington

Abstract

Lahar: Warehousing Markovian Streams

Julia Maureen Letchner

Chair of the Supervisory Committee:
Professor Magdalena Balazinska
Computer Science and Engineering

A huge amount of the world's data is both *sequential* and *low-level*. Many applications consume higher-level information, such as words and sentences, that is *inferred* from low-level sequences such as raw audio signals using a model (e.g., a hidden Markov model). This inference process is typically statistical, resulting in high-level streams that are imprecise. These imprecise streams, once archived, are useful for analytics support including sequence-finding *event queries* (e.g. *"Find all times when the phrase 'Barack Obama...veto' occurs in the NPR news podcast from July 9."*), event query aggregates (e.g. *"How many times do 2008 NPR podcasts use the phrase 'Barack Obama...veto'?"*), and event query *lineage* (e.g. *"What words appeared between the word 'Obama' and 'veto' in the previous query?"*). These queries are difficult to support efficiently because archives can be large, and standard relational warehouses cannot support analytics on the rich semantics of imprecise sequences; however, these analytics are critical for allowing applications to effectively leverage this data.

In this thesis, we introduce Lahar, the first database system for a common type of imprecise, sequential model called a *Markovian stream*. Lahar includes novel algorithms for efficiently processing aggregated event queries, and event query lineage. Lahar accelerates performance and scalability of all queries using several techniques, including a set of novel Markovian stream indices and novel methods for approximating Markovian streams. Through experiments on two real-world datasets (one collected from an office-building RFID deployment and the other collected from audio podcasts) we demonstrate that Lahar is an efficient Markovian stream warehousing system.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This thesis is the product of many years in graduate school, and during that time I have had the luck to work with wonderful colleagues and to be supported by my family and great friends. This thesis would not exist without them.

I owe a large debt of gratitude to my advisor, Magda Balazinska, who was willing to take me on as a student when I came knocking on her door unannounced, and who guided me through several years of work on Lahar. Magda often saw the path of my research long before I did, but had the patience to guide me along gently so that I could learn the research process for myself. She pushed me to refine and improve my ideas, and always gave me the encouragement to get through rough patches by finding something positive to say about any situation.

I am grateful to Dan Suciu for providing valuable and thorough feedback on my work over the past few years, and for always being so very willing to do so. I am also grateful to Dieter Fox for first introducing me to the mysterious ways of academic research, to Gaetano Borriello for serving on my committee and for often providing a good laugh, and to John Krumm for providing kind guidance on my research and job search. Thanks also to Ming-Ting Sun for serving on my committee as a Graduate School representative.

I am lucky to have been a part of the database group at the same time as many other great graduate students, many of whom are also now my friends. Chris Ré in particular was instrumental in helping to get the Lahar project going and in providing a success model for me to follow. I owe a special thanks to fellow graduate students Anna Cavender, Dan Halperin, Nodira Khoussainova, Jon Ko, YongChul Kwon, and Vibhor Rastogi for listening to countless practice talks. Without their input and support, particularly during my job search, I would surely be facing unemployment next year. I also owe thanks to Evan Welbourne, Emad Soroush, and the RFID Ecosystem team for their help in collecting RFID traces, and to Jennifer Wong for building a GUI for Lahar as well as various support tools. Thanks also to Lindsay Michimoto for her emotional and logistical support. Finally, I owe thanks to Benson Limketkai for years of mutual pep talks, and for helping me hone my Boggle skills. Someday he will perhaps win a round.

I owe thanks to Jean-Claude Latombe, Oussama Khatib, and Eve Coste-Maniere for encouraging me to pursue graduate school. Had it not been so rewarding, I might begrudge them my last seven years at a minimum-wage salary.

I would never have reached graduate school, let alone completed it, were it not for the unflagging support of my family. My parents have enjoyed tallying the number of years I have spent in school, watching bemusedly as the count grew past 12 and then beyond 20. Even at 24 years, however, they still proudly post my work on their refrigerator, and seeing it there is the best reward I could receive. I am thankful for the love and support they have always given me generously and without question. I am also grateful to Matt, for being the kind of little brother who would buy his big sister a milkshake when she was too shy to do it herself.

Finally, I am deeply thankful for the love and support of Seth Bridges, who has been by my side both to share my failures and celebrate my triumphs. I would not have made it through graduate school without his encouragement, hugs, advice, humor, loyalty, and excellent cooking.

DEDICATION

To Hubba for my love of science

and to Mo for my love of words.

Chapter 1
INTRODUCTION

People and computers worldwide generate exabytes of audio, video, text, GPS[1], RFID[2], and other types of multimedia and sensor data—and because disk storage is cheap, most of this data is archived for future use [51]. These information-rich archives are poised to revolutionize data-centric applications in diverse areas including patient and asset tracking in hospitals [100], activity monitoring for elder care [85], scientific environment observation [37], e-Learning [123], phone conversation mining, and multimedia search/retrieval.

Such data stream archives are valuable to applications for the information they contain. However, applications are generally interested in high-level information that is implicitly contained in data streams, but is not exposed explicitly in the raw data and must instead be *inferred* using sophisticated post-processing techniques. Audio indexing/search applications, for example, cannot index raw audio signals directly, but instead operate on sequences of spoken words that are inferred from the audio signals. Similarly, location-based applications report location to users in terms of rooms in a building or addresses on a street, but these locations must be inferred via post-processing from raw GPS coordinates or RFID antenna detections.

Many applications also share a need for sophisticated queries on such implicit, high-level information. Such queries include event queries, which detect occurrences of fixed patterns in a single stream [128, 35]. Example event queries include, *"Find all occurrences of the phrase 'launchers ready' in telephone conversations recorded in January 2008,"* and *"Find all times when the crash cart went straight to the ICU after exiting sterilization during the week of May 1."* Applications may also require aggregated event queries to compute simple statistics about the occurrences of events across a set of high-level streams. These queries are important for performing analytics or data mining on stream archives. Example aggregate queries include, *"How many people entered the ICU on*

[1]Global Positioning System

[2]Radio Frequency Identification

March 20, 2009?" and *"How many times does the word 'launchers' appear across all telephone conversations recorded in 2008?"* Finally, applications require access to detailed information about specific event query matches, to filter out results based on these details or to understand unexpected results more fully. For example, consider the event query, *"Find all times when the crash cart went from the ICU to a sterilization unit during the week of May 1,"*. Example questions that require access to the details of intervals matching this query include, *"What path did the cart take from the ICU to the sterilization unit?"* or *"Which sterilization unit did the cart enter?"*

A major challenge to supporting these application queries on high-level streams is that the high-level information inferred from data archives is nearly always *imprecise*, due to noise, ambiguity, or both. For example, in an audio indexing application, background noise or speaker dialect may render a speech processing system unable to determine the difference between "Tom has" and "Thomas". Similarly, language ambiguities like "eight" vs. "ate" can be difficult to resolve. The result is that applications must process queries on high-level information that is presented as a set of possible values (as in Figure 1.2), instead of a single "true" answer.

Because high-level streams are often difficult to extract, and most applications are poorly-equipped to handle the imprecise results of such extraction, today's analytical tools only scratch the surface of the information contained in data archives. Current speech processing tools, for example, can identify audio files containing particular keywords or keyword pairs, but cannot support searches on audio files using keyphrases of arbitrary length, or that include wildcards. Similarly, archives of location data collected using RFID or GPS can be used to answer basic questions about an object or person's location at a given time, but detecting patterns or events of interest (e.g. the movement of equipment from a patient room into an access-controlled surgery unit) on these same archives is difficult.

In cases such as these, extracting utility from raw data sequences is difficult, not because the desired information does not exist in the data, but because applications lack a sufficient framework for performing post-processing and analysis on the data. In the absence of such a framework, three general approaches have emerged for performing sophisticated stream analysis (Figure 1.1). The first approach, shown in Figure 1.1(a), is an ad-hoc approach in which application developers write their own inference and analysis code from scratch. Clearly, this approach is time-consuming and produces brittle solutions that are difficult to adapt to novel data or analyses.

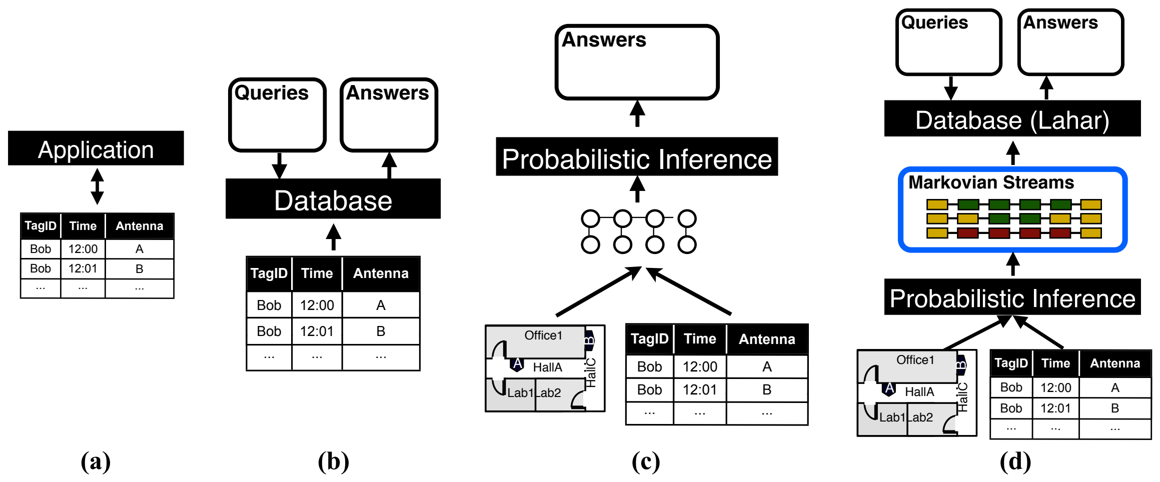


Figure 1.1: Four approaches to processing high-level information in data streams. (a) Ad-hoc approach in which applications perform hand-coded analyses. (b) Database approach in which raw data is placed directly in a database. (c) Artificial intelligence approach in which queries are encoded in a graphical model, and query processing is framed as probabilistic inference. (d) Markovian stream approach in which probabilistic inference is used to generate a Markovian stream, which is then queried using an appropriate database (e.g. Lahar).

The second approach (Figure 1.1(b)) has emerged from the artificial intelligence (AI) community, and encodes all extraction/analysis as a single probabilistic inference problem [61, 58]. This approach can be applied only for the most basic of the example queries outlined above—pattern matching queries—and even when applicable the approach scales poorly because it must begin analysis afresh, from scratch, for each query. However, when applicable, the probabilistic inference techniques used in this approach are robust to noise and produce high-quality results.

The third approach, shown in Figure 1.1(c), has emerged from the database community. In this approach, raw data is placed directly into a database, and analysis is performed using SQL queries [43, 22, 50]. This approach bypasses high-level information entirely, making it difficult to use on many data sets (e.g. RFID queries in this approach are highly sensitive to noise in the data), and impossible to use on others (e.g. audio data in this approach must be analyzed using queries directly on the raw audio signal, with no reference to high-level word information). Of course, when the database approach is applicable, it is fast and scalable because it is based on standard database technology, which also allows queries to be written flexibly in SQL.

1.1 The Markovian Stream Approach

In this thesis, we propose a framework for sophisticated analysis of the high-level information contained in sequential data such as audio or multimedia streams, or sensor streams like RFID, GPS, *etc.*. We call this approach the Markovian stream approach, and it is illustrated in Figure 1.1(d).

The Markovian stream approach to stream analysis has two parts. In the first part, high-level sequences are extracted from raw data using standard probabilistic inference techniques [39, 58]. These high-level sequences are represented as *Markovian streams* [94, 73, 61], which are a specific type of imprecise sequence which we define formally in Chapter 2. Markovian streams use probability distributions to represent uncertainty about the true value of a data sequence (i.e. the words spoken in a sentence, as in Figure 1.2(a), or the true location of an object, as in Figure 1.2(b)). Markovian streams are extremely general, and subsume the types of sequences commonly used in today's applications, including sentences inferred from audio signals, location sequences inferred from RFID/GPS, *etc.*.

In the second part of this approach, the inferred Markovian streams are materialized to disk and queried/analyzed using a database engine. Because standard relational databases cannot support Markovian stream data, a novel database is required to support this half of the Markovian stream approach. Lahar[3], the topic of this thesis, is the first database to support Markovian streams and its design and implementation are motivated by the Markovian stream approach to stream analysis.

The key feature of the Markovian stream approach is that it uses materialized Markovian streams to separate the inference process—that is, Markovian stream generation—from the analysis process—that is, Markovian stream querying. One benefit of this separation is that applications using this approach are free to substitute different probabilistic inference techniques to generate Markovian streams from different types of data. Along the same lines, each application must convert a raw input stream into a Markovian stream only once: the database can query the Markovian stream as often as necessary without ever again touching the raw input stream. Another benefit

[3]A Lahar is a huge mudslide, or stream of dirt and debris, caused by the eruption of a snow-covered volcano. This system is named Lahar for its ability to manage massive streams of "dirty" (imprecise) data.

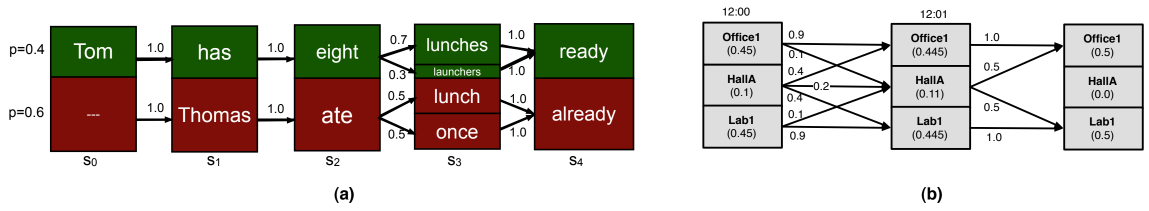


Figure 1.2: (a) Sample Markovian stream over a text (language) domain, inferred from spoken audio. (b) Sample Markovian stream over a location domain, inferred from RFID sensors. Boxes represent the possible stream values at each instant, while arrows represent the conditional probability of two values in sequence. In (a), for example, if the true word spoken at time s_2 is "eight", then the word spoken at time s_3 is "lunches" with probability 0.7 and "launchers" with probability 0.3"

of this approach is that the Markovian stream database (Lahar, in this case) can optimize performance for high-level queries, and need not worry about extracting high-level streams from raw data. This allows the database to support a query language and data model that are independent of any particular data domain.

By separating Markovian stream generation and analysis, the Markovian stream approach achieves the flexibility and scalability of a database-oriented approach, but also maintains the high accuracy and robustness to noise of an AI-oriented approach. The Markovian stream approach is a conceptual contribution of this thesis. The generation of Markovian streams is *not* a contribution of this thesis; many techniques exist for generating these streams, as we outline in our background discussion (Chapter 2). Similarly, the basic algorithms for processing real-time Markovian streams were developed elsewhere [94]. The contributions of this dissertation are algorithms for efficient and powerful querying of markovian stream *archives*. These techniques include (1) indexing, (2) approximation, and (3) lineage processing, as we discuss in more detail below. We first introduce two data domains that serve as running examples throughout the thesis.

1.2 Motivating Applications

Here we introduce two applications that we use as running examples throughout this thesis, and in the experimental evaluations of Lahar.

1.2.1 RFID-Based Location Tracking

RFID readers are increasingly deployed in warehouses, office buildings, and hospitals, to track the locations of products, equipment, and even people (e.g. patients carrying RFID tags [100, 117, 83, 101, 11, 50]). This data is used in real-time to locate objects, but can also be queried in a historical context to streamline efficiency (*"How often does the ICU need to borrow a crash cart from neighboring units?"*) or to evaluate details of an organization's procedures (*"How long does each patient spend in pre-op before entering the operating room?"*).

Currently, analysts interested in historical queries on such RFID data store the raw data directly in a relational database, or in a database built specifically for RFID data [45]. Such databases may provide some level of data cleaning (e.g. insertion of missing readings) [46, 56], but they are incapable of reflecting the fundamental uncertainty inherent in sensor systems. Furthermore, these databases all require analysts to reason in terms of tag-detection tuples (e.g. "Bob was sighted by reader 6") instead of room-level locations that are more natural and more useful (e.g. "Bob was in the ICU"). The result is that today's RFID analyses are unintuitive and brittle in the face of noisy data.

In the Markovian stream approach, raw RFID streams are used as input to probabilistic inference (e.g. a particle filter [39]). The output of inference is a Markovian stream that concisely captures the uncertainty about an object's true location (e.g. Bob is either in triage or in the neighboring ICU). This uncertainty is represented as a probability distribution over possible object paths, and takes the form shown in Figure 1.2(b). The first timestep of the stream represents a probability distribution over the possible initial locations of the object. Subsequent timesteps contain temporal correlations that capture the probability of the object transitioning to a new location (or remaining stationary), since an object's location at each instant is strongly related to its location at the previous instant. Lahar allows analysts to query these uncertain location streams without reasoning explicitly about uncertainty. Importantly, RFID-based Markovian streams are expressed at a semantic level that is interesting to analysts (i.e. room-level location).

After the data in a raw RFID stream is converted into a Markovian stream, analysts can run sophisticated queries on the Markovian streams using Lahar. In this framework, analysts can express their queries at the intuitive level of rooms (i.e. *"When did Bob enter the ICU"*), and without regard for any uncertainty or noise present in the input data. This abstracted, robust-to-noise view of location is important in domains beyond hospitals, including supply chains [50, 45] and office environments [124].

1.2.2 Semantic Audio Processing

Text processing is a complex but common task today. Search engines can retrieve text documents based on keywords or keyphrases, and can use techniques such as tf-idf [105] to rank these documents based on their relevance to a particular query. Sophisticated data mining techniques can also compute the similarity between two text documents, for example to detect plagiarism [71].

Search engines and data analysts can, in theory, apply these text analysis techniques to audio and video data by first extracting a text-based transcript of the sentences spoken in these multimedia files. Speech recognition systems have been using probabilistic models for decades to infer phoneme- and word-level transcripts of raw audio signals [87,74]. However, such transcripts invariably contain imprecision (Figure 1.2(a)), because natural language is inherently ambiguous (e.g. “ate” vs. “eight”), because dialects and accents vary across individuals, and because in many situations poor data quality such as background noise limits the amount of information that can be reliably extracted. Standard text analyses are difficult to apply directly to text sequences that contain uncertainty; today’s spoken audio analysis tools are limited to single-keyword searches, or rely on phrase retrieval algorithms that provide approximate rankings and yield many false positives.

By using the Markovian stream approach and processing these imprecise transcripts as Markovian streams in Lahar, analysts can perform more sophisticated analyses on audio data, such as searches on multi-word keyphrases, searches for phrases that include wildcards, and word-counting queries.

1.3 Contributions

To enable applications to easily process sensor, multimedia, or other sequential data, this thesis proposes a novel Markovian stream approach in which raw data sequences are first preprocessed into Markovian streams, which are then queried using a database. Central to this approach is a data management system that can support event queries and event query variants on Markovian streams. While previous work has studied event query processing on *streaming* data [94], in this thesis we focus on processing stream *archives*. An imprecise stream archive processing system is challenging to build for several reasons. First, relative to traditional (deterministic) data, imprecise data is extremely slow to process because each of the exponentially-many possible worlds encoded by the data's imprecise values must be considered. Second, the scale of sensor and multimedia archives can be huge, due to both the high rate at which data is collected over months and years, and because the number of individual input streams is large (e.g. the number of items in a hospital or of podcasts on the web). Finally, some queries return results that scale with the size of the input data, and are thus impossible to enumerate even if they can be efficiently computed. For example, the previously-introduced query, *"What path did the crash cart take from the ICU to a sterilization unit?"* can produce result paths that are arbitrarily long, depending on the amount of time that the cart spent in the hallways before reaching a sterilization unit.

This thesis presents the design, implementation, and evaluation of a series of query processing techniques for Markovian stream *archives*. Together, these techniques support the first Markovian stream database implementation, called Lahar, which makes the Markovian stream approach to stream processing possible. The input to Lahar is a set of imprecise, sequentially-correlated streams called *Markovian streams* [94, 73, 61], which are described in detail in Chapter 2 and shown in Figures 1.2(a) and (b). Lahar supports event queries, aggregates over event query results, and lineage queries. The novel contributions of this thesis are algorithms for processing aggregate-event and lineage queries, and several scalability techniques for improving the speed of Markovian stream processing in general. Concretely, the technical contributions of this thesis are as follow:

1. **Markovian Stream Indexing (Chapter 4):** In relational databases, indexing is commonly used to accelerate query processing over large relations. Traditional indices, however, do not support ordered, imprecise data like that of Markovian streams. To address this challenge, we introduce two variations of traditional B+ tree indices, adapted for indexing Markovian streams, and a third, novel Markovian stream index called the Markov chain index. The first two indices allow Lahar to efficiently identify the Markovian stream intervals that are directly relevant to a particular query, and to retrieve them for processing in order (either chronological

order or order of the highest probability, depending on application requirements). The Markov chain index provides efficient lookup of the correlations between any two Markovian stream instants (equivalently, timesteps), reducing the lookup time from O(N) to O(log N), where N is the distance between the two instants. Experimental results on synthetic and real Markovian streams demonstrate that these indices reduce query times by two orders of magnitude in the best case, and incur negligible overhead even in the worst case.

2. **Approximating Markovian Streams (Chapter 5):** Approximation techniques are often used in data modeling and data management systems to improve performance. Applications that rely on Markovian streams are accustomed to handling imprecision, and are often willing to trade accuracy for performance. To support such a tradeoff, we introduce several Markovian stream approximations that reduce the size and/or complexity of the representation of each Markovian stream instant. We demonstrate on real-world Markovian streams that the complexity, and not the size, of the approximate representation determines performance. We furthermore demonstrate that, by processing approximate representations, Lahar can trade accuracy for efficiency, increasing speed by one to two orders of magnitude with effects on accuracy that vary based on both the type of query and the data domain.

3. **Markovian Stream Lineage (Chapter 6):** Although event query processing is fundamental to Markovian stream management, many applications require additional queries that return detailed information about not only *when*, but *how* an event is matched in an input stream. Supporting these queries is challenging because "how" explanations can grow exponentially in the length of the input stream. To address this challenge, we formally define Markovian stream lineage, which, informally, is a representation of the parts of a Markovian stream that contribute to a particular query result. We introduce an O(N) algorithm for producing the lineage of event queries, where N is the length of a Markovian stream. We further introduce two algorithms for supporting projection on event query lineage, allowing applications to see only subsets of lineage (e.g. start and end locations, or the duration of a query match). We demonstrate on real-world Markovian streams that lineage can be generated and queried scalably, and we further demonstrate optimizations that can reduce the cost of projection by

half.

The rest of this thesis is organized as follows: Chapter 2 provides a brief introduction to Markovian streams and event queries; Chapter 3 discusses the Lahar system, including architecture, prototype implementation, language, and experimental data sets. The primary research contributions of this thesis are presented in Chapters 4, 5, and 6, which discuss indexing, approximation, and lineage, respectively. A discussion of related work follows in Chapter 7. Chapter 8 concludes and outlines areas for future work.

Chapter 2

BACKGROUND

Lahar is the first database for managing Markovian streams. In this chapter, we define these streams formally and give examples of streams from two real-world domains (Section 2.1). We also introduce event queries, a basic form of Markovian stream processing (Section 2.2), and discuss more complex types of Markovian stream processing that are not covered elsewhere in this thesis (Section 2.3).

Before moving on to formal definitions, it is worth noting here that the concept of Markovian streams is a novel one, as is the concept of managing these streams in a database. Although Markovian stream data is generated frequently by various applications (e.g. speech processing [87]), these streams are seldom materialized for later use. Instead, these applications generate Markovian streams as an intermediate processing step, and discard them when processing is complete.

In contrast, a key idea of this thesis is to materialize Markovian streams, store them to disk for later processing, and manage them using a database that is aware of their semantics (i.e. Lahar). Thus, although data meeting the Markovian stream definition has been common for decades, the term "Markovian stream" was coined for this data in the first paper about Lahar, by Ré, Letchner et al. in SIGMOD 2008 [94], where the idea was first put forth to manage these streams as first-class objects in a database.

In this chapter, we first formally define Markovian streams (Section 2.1). We then describe important background for understanding Lahar, focusing on a key algorithm for basic event query processing on these streams which was first introduced by Ré, Letchner et al. [94](Section 2.2). We finish with an overview of additional types of Markovian stream queries (Section 2.3): Some of these are background to this thesis [94], while others are novel but minor contributions of this thesis that do not appear in other chapters.

2.1 Markovian Streams

In the context of this thesis, a *stream* is an unbounded set of pairs $\langle e, i \rangle$ where e is a stream element (equivalent to a tuple in a standard relation) and i is an index indicating the logical ordering of elements e in the stream. For example, an RFID location-tracking system might produce a stream in which each element $e = (a, g)$ is a two-dimensional tuple identifying an RFID antenna ID a and RFID tag ID g, and the stream index i is the timestamp at which the given tag was sighted by the given antenna.

A *Markovian stream* is a stream in which the elements e are 1) imprecise and 2) contain Markovian correlations. A Markovian stream is thus an imprecise, temporally-correlated data sequence. Markovian streams appear in many different domains, when high-level information is *inferred* from low-level data (e.g. location inferred from RFID, or activity inferred from wearable sensors or smart homes). Such inference typically involves some uncertainty. For example, spoken words inferred from an audio signal are often noisy, due to language ambiguity or background noise: a two-word phrase may be “eight launchers” or “ate lunches”, each with some likelihood. Intuitively, Markovian streams capture this uncertainty by identifying: 1) the set of different possible “true” stream values at each instant in time, and 2) the correlations between possible values at adjacent instants. The Markovian stream in Figure 1.2(a), for example, reflects the fact that one of two possible words ({eight, ate}) was spoken at instant s_2, and one of four possible words ({lunches, launchers, lunch, once }) was spoken at instant s_3. Similarly, the arrows between the values at instants s_2 and s_3 represent correlations, which indicate the likelihood of sequences: *if* the true value of the stream at instant s_2 is “eight”, then the true value of the stream at instant s_3 is either “lunches” or “launchers”. If, on the other hand, the true value at s_2 is “ate”, then the next value is either “lunch” or “once”. Each of these possible values and correlations carries a probability value identifying the likelihood with which it is true.

In the remainder of this section, we first give a brief, formal definition of Markovian streams. We follow this definition with two real-world Markovian stream examples which provide a more intuitive description of the Markovian stream data structure.

2.1.1 Formal Definition

Conceptually, a Markovian stream of length N represents a probability distribution over a set of *possible worlds*, which are deterministic sequences of length N. Elements in these sequences are taken from an l-dimensional domain $\mathbf{D} = (D_1 \times \cdots \times D_l)$. All examples and datasets in this thesis use single-dimensional stream domains, so for simplicity of notation we will use a uni-dimensional domain D throughout. Thus, a Markovian stream of length N over domain D is a probability distribution over the sequences (possible worlds) in the set D^N.

Formally, a Markovian stream is a pair $(\mathtt{M}_0, \vec{\mathtt{C}})$. Here, $\mathtt{M}_0$ is a *marginal probability distribution* over D, representing the distribution over the first element in the stream. The vector $\mathtt{M}_0$ consists of entries (d, p) that indicate the probability p that the value of the first stream element is d. $\vec{\mathtt{C}}$ is a sequence of *conditional probability distributions* $\mathtt{C}_i, (0 \leq i < N)$, over $D \times D$. These conditional distributions indicate the probability of specific transitions in the stream (i.e. the probability that one particular domain element follows another at a specific instant i in the stream). For example, the entry (d, d', p) in conditional distribution $\mathtt{C}_i$ indicates that p is the probability that the $(i+1)^{th}$ stream element is d', *conditioned* on the fact that the i^{th} stream element is d. For example, in Figure 1.2(a), the entry (“eight”, “launchers”, 0.3) in C_2 indicates that the probability of the fourth word in the sequence is “launchers” with probability 0.7, *given* that the third word in the sequences is “eight”. The set of all entries in a conditional distribution $\mathtt{C}_i$ together define the transition, or the *correlations*, between the stream state at instants i and $i + 1$.

Markovian streams encode a probability distribution over the set of sequences D^N as follows: a deterministic sequence $x \in D^N$ has probability $p(x) = \mathtt{M}_0(x_0)\mathtt{C}_1(x_0, x_1)\mathtt{C}_2(x_1, x_2)\ldots\mathtt{C}_{n-1}(x_{N-1}, x_N)$. That is, the probability of sequence x is the probability of the first element in x, as given by $\mathtt{M}_0$, times the probability of each transition in x, as given by the conditional distributions $\mathtt{C}_i$. As an example, the probability of the sequence (“Tom has eight launchers ready”) in the Markovian stream in Figure 1.2(a) is (0.4*1.0*1.0*0.3*1.0)=0.12. Because a Markovian stream is a probability distribution, the probabilities of all possible worlds (sequences) in the stream sum to 1.0.

Markovian streams are so-named because they exhibit the Markovian property: the stream state at instant i is independent of the rest of the stream, given the stream state at instants $i - 1$ and $i + 1$. Markovian streams exhibit the Markovian property because the conditional distributions $\mathtt{C}_i$ encode correlations only between adjacent instants (as opposed to encoding relationships between more distant pairs of instants, which would yield a non-Markovian stream).

Markovian streams are produced as the output of probabilistic inference on a temporal graphical model such as a hidden Markov model (HMM) [87], dynamic Bayesian network (DBN) [76], or a junction tree [58]. Many standard techniques exist for performing this inference, including exact (e.g. message passing) and approximate (e.g. MCMC) methods. The generation of Markovian streams is not a contribution or focus of this thesis, since these techniques are standard and Markovian stream generation occurs outside of the Lahar system. However, Section 3.3.1 describes the processes used to generate the Markovian streams used in the evaluation of Lahar.

The ordering of elements in a Markovian stream is defined implicitly by the element indices i. However, Lahar's data model provides support for additional, redundant timestamps to be associated with each stream element. This timestamp field is not part of the formal stream definition, but it is a convenient way for applications to reason in terms of sequences meaningful to them (e.g. timestamps instead of indices assigned by Lahar). For simplicity, Figure 2.1(b) and some other figures in this thesis use the more intuitive, application-set timestamp sequence identifiers instead of the raw indexes used in the formal definition.

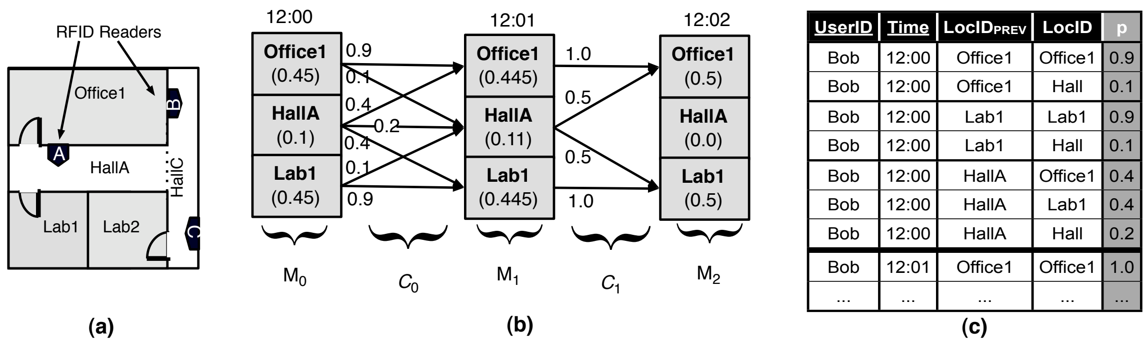
2.1.2 Examples

In this section, we revisit the example Markovian stream domains from the introduction to describe Markovian streams from these domains in more detail and to provide additional intuition about the definition of Markovian streams.

Location From RFID

Recall that an RFID deployment includes a set of mobile RFID tags attached to people or objects, and a set of statically-located RFID readers that detect the presence of tags when they are in the vicinity of the reader. Readers record the presence of nearby tags by recording $\langle tagID, readerID, timestamp \rangle$ tuples in a database.

Figure 2.1(a) shows a small portion of such an RFID deployment, while Figure 2.1(b) shows a sample Markovian stream derived from the RFID readings collected in this setting, which could be



UserID	Time	LocID$_{PREV}$	LocID	p
Bob	12:00	Office1	Office1	0.9
Bob	12:00	Office1	Hall	0.1
Bob	12:00	Lab1	Lab1	0.9
Bob	12:00	Lab1	Hall	0.1
Bob	12:00	HallA	Office1	0.4
Bob	12:00	HallA	Lab1	0.4
Bob	12:00	HallA	Hall	0.2
Bob	12:01	Office1	Office1	1.0
...	...	...	...	...

Figure 2.1: (a) Sample floor plan of a building showing an RFID installation. (b) Markovian stream over an RFID tag's location, inferred from RFID readings recorded in the setting shown in (a). (c) A relational representation of the conditional distributions $\vec{\mathtt{C}}$.

an office building, a home, a hospital, *etc.*. Figure 2.1(c) shows a relational representation of the Markovian stream in Figure 2.1(b). The domain $D = \{\text{Office1}, \text{HallA}, \ldots\}$ of this Markovian stream is the set of locations in the building: each instant of the stream defines a probability distribution over the location of the RFID tag at that instant. In Figure 2.1(b), $\mathtt{M}_0$ indicates that the first element in the sequence is Office1 with probability 0.45, HallA with probability 0.1, *etc.* (rooms not pictured have probability 0.0). The conditional distribution $\mathtt{C}_0$ indicates that, conditioned on Office1 being the "true" room at instant 0, then the "true" room at instant 1 is Office1 with probability 0.9, or HallA with probability 0.1. Each entry in a conditional distribution $\mathtt{C}_i$ is drawn visually as an arrow connecting two domain elements at instants i and $i + 1$ in Figure 2.1(b), but these entries are stored on disk using a relational format (as in Figure 2.1(c)).

Note that, although Figure 2.1(b) explicitly shows the marginal distributions $\mathtt{M}_1$ and $\mathtt{M}_2$, indicating the distribution over the RFID tag location at instants 1 and 2, these marginal distributions $\vec{\mathtt{M}} = [\mathtt{M}_0, \ldots \mathtt{M}_N]$ are not included in the formal Markovian stream definition because they are redundant. Their values can be derived from $(\mathtt{M}_0, \vec{\mathtt{C}})$: The marginal probability that a particular domain element (location) d^k is the true domain element at instant i is the sum of the probabilities of all deterministic sequences $d_0, d_1, d_2, \ldots d_n \in (\mathtt{M}_0, \mathtt{C}_1, \ldots \mathtt{C}_N)$ whose i^{th} element d_i is equal to d^k. However, explicit visualization of these marginal distributions is useful for understanding a Markovian stream, and indeed Lahar stores these marginal distributions explicitly for performance reasons (Section 3.2.4). Thus, in some cases, we refer to a Markovian stream as ($\vec{\mathtt{M}}, \vec{\mathtt{C}}$), to make clear the explicit presence of the marginal distributions $\mathtt{M}$.

Similarly, Lahar does not explicitly store distribution entries (locations, in this example) with zero probability. Instead, any domain element not explicitly listed in a probability distribution (either marginal or conditional) is implicitly assumed to have zero probability (e.g. Lab2, HallC, in Figure 2.1(b).

Location-based Markovian streams are often inferred from RFID readings using sample-based inference methods such as MCMC [39]. Such inference receives as input not only the RFID antenna readings, but also information about the floor plan of the building, the locations of the RFID antennas within the floor plan, statistics about the speed at which objects generally travel, and physical constraints (i.e. that objects cannot move through walls or floors). Sophisticated models may include additional information such as the direction in which people or objects typically turn when passing through a hallway intersection, or information about where people spend the majority of their time. These details serve to increase the fidelity of the Markovian stream with respect to the true movement of the person or object being tracked. We discuss the details of the model used to infer the real-world, RFID-derived Markovian streams in this thesis in Section 3.3.1.

Text From Speech

Recall from Section 1.2.2 that audio processing techniques return imprecise transcripts of the sentences spoken in a given file. These transcripts are difficult to process using standard text processing algorithms, but they are easy to model as Markovian streams, allowing text analysis to be performed in Lahar.

In this case, the stream's uncertain domain D is the set of all English words. An example of such a stream is shown in Figure 1.2(a). The marginal probability distribution at each instant i of this stream represents the distribution over the i^{th} word spoken, while each conditional distribution, drawn as arrows in the figure, states the probability that one particular word is followed by another at a given instant, based on language models and the sounds in the input audio recording. Figure 1.2(a) is shaded to reflect the importance of correlations in this setting: note that *either* one of the green

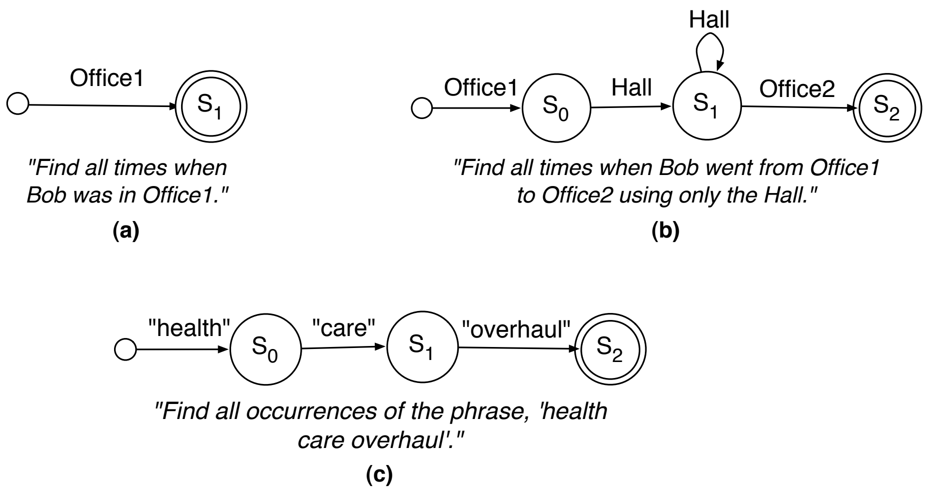

Office1
S1
"Find all times when Bob was in Office1."
(a)
Hall
Office1
S0
Hall
S1
Office2
S2
"Find all times when Bob went from Office1 to Office2 using only the Hall."
(b)
"health"
S0
"care"
S1
"overhaul"
S2
"Find all occurrences of the phrase, 'health care overhaul'."
(c)

Figure 2.2: (a-b) Sample event queries over a location domain. (c) Sample event query over a speech domain.

("Tom...") sentences was spoken, *or* one of the red ("Thomas...") sentences was spoken, but there is zero probability of any sentence that mixes together words from these two sets.

For a discussion of the speech processing used to generate Markovian streams from recorded speech, see Section 3.3.2.

2.2 Event Queries

Event queries search for particular sequences of states within an ordered data sequence. They are a common foundation for deterministic stream processing [35,5], and are thus a natural fit for querying Markovian streams. Example event queries include, *"When did the crash cart move from the ER to the ICU?"* (for location streams), or, *"Find all instances of the phrase 'health care overhaul'."* (for speech streams).

2.2.1 Defining Event Queries

Event queries can be represented as *nondeterministic finite automata* (NFAs), or equivalently, regular expressions. Event query NFAs differ from standard NFAs in that they are expressed in terms of Boolean *predicates* on the Markovian stream domain, instead of in terms of static alphabet sym-

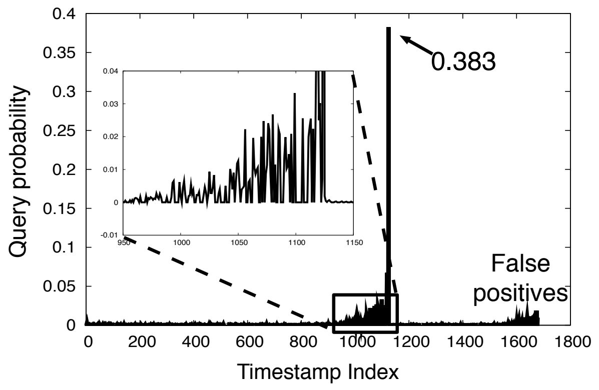


Figure 2.3: Query signal for a query similar to the one in Figure 2.2(b), on a Markovian stream inferred from real-world RFID data.

bols. For example, the NFA query in Figure 2.2(c) is written for the speech domain and searches for three consecutive stream timesteps satisfying equality predicates on the words “health”, “care”, and “overhaul”, respectively. In all figures in this thesis, NFA edge labels such as “Hall” or “health” are shorthand for function-like Boolean predicates that operate on the uncertain elements of the stream (i.e. Hall(location) on an RFID-derived stream or health(word) on an audio-derived stream). The most simple predicates are equality tests (e.g. health(word)), but predicates can be arbitrarily-sophisticated tests on a domain element. Sophisticated predicates may join domain elements with dimension tables (e.g. a “Hall(location)” predicate that returns true on *any* hallway location by joining with a *Hall* table listing the IDs of all hallway locations). Composite predicates can also be created using conjunction, disjunction, and negation.

Figure 2.2(a-c) shows several event queries. The first two queries are written for a location domain. The first, (a), identifies all timesteps in which the RFID tag (attached to Bob, in this case) is in a location labeled “Office1”. The second (b), identifies all instants in which Bob has entered location “Office2”, having previously been in location “Office1” and then having been only in locations satisfying the “Hall” predicate described previously. Figure 2.2(c) shows a query written for a speech domain and searching for all instances of the phrase, “health care overhaul”.

Event query processing on a deterministic stream proceeds by reading a single stream element (i.e. a single instant of data), and updating the state of the NFA according to the transitions triggered by the element. For example, if the state machine in Figure 2.2(b) is in state s_0, an input satisfying the Hall predicate will cause the machine to transition out of s_0 and into s_1. An input of Office2, however, triggers no transitions out of s_0 because it does not satisfy the Hall predicate; in this case the state machine reverts back to its start state (drawn as small, unmarked circles in Figures 2.2(a-c)). An event query is *satisfied* at any instant in which these transitions cause the NFA to enter its final, or accepting, state, identified in Figure 2.2(a-c) using double circles. Section 2.2.2 discusses event query processing on Markovian streams.

In deterministic stream processing, the output of event query processing is a sequence of N zeros and ones, where a 1 at position i means that the query was satisfied at the i^{th} stream instant, and a 0 means the query was not satisfied at that instant. In Markovian stream processing, the output is a sequence of N probabilities, where the probability p_i at position i in the output indicates the probability with which the event query was satisfied at the i^{th} instant in the stream. These output probabilities reflect the fact that Markovian streams are imprecise. The output sequence of $\langle i, p_i \rangle$ tuples is called a *query signal*. Figure 6.1 shows an example of a query signal computed on a real RFID-based Markovian stream, for a query similar to the one shown in Figure 2.2(b). The signal spikes when the RFID tag enters the office ($t \approx 1100$), and shows a much lower series of peaks around a false positive ($t \approx 1600$), when the RFID tag simply moved past the office's doorway without entering. Applications can use simple thresholding to detect the instants when an event actually occurred (e.g. the tag has entered the office at all times when $p > 0.3$).

2.2.2 Processing Event Queries

As described in Section 2.2.1, the input to event query processing is 1) a Markovian stream and 2) an event query. The output is a list of $\langle i, p_i \rangle$ tuples describing the probability p_i with which the event query is satisfied at each instant i in the input stream. Query processing can be performed using one of two methods: a naïve method that requires time exponential in the length of the input stream, or a linear-time algorithm originally published by Ré, Letchner et al.. in SIGMOD 2008 [94]. The event queries that we discuss here correspond to "regular queries" presented in the SIGMOD 2008 paper.

The naïve method for processing event queries performs a straightforward enumeration of all $|D|^N$ deterministic sequences encoded in the input Markovian stream. It uses standard NFA processing techniques to evaluate the query separately on each stream—that is, to identify the timesteps in each deterministic stream that satisfy the query. The final probability that the query is satisfied at instant i is the sum of the probabilities of all deterministic streams in which i is a satisfying timestep. Clearly, this approach is not feasible in practice because it requires enumeration of an exponential number of sequences. Another processing alternative is to process only a sampling of possible worlds, but to guarantee a high-quality result using this approach requires a number of samples that grows exponentially with the length of the input stream.

In place of naïve enumeration, Lahar processes event queries using a matrix-based algorithm [94] that adapts NFA processing machinery to handle Markovian input streams. The algorithm is a single-pass algorithm that runs in time linear in the length N of the input stream. The algorithm processes each instant in the stream exactly once, reading each instant in order from the beginning of the stream to the end.

The event query processing algorithm maintains a 2D state matrix Q with a row for each possible *set* of NFA states (sets instead of individual states because the NFA is non-deterministic and can be in multiple states at once) and a column for each element d in the Markovian stream's uncertain domain. Entry $Q(i, j)$ represents the joint probability that the NFA is in the i^{th} state set *and* that the "true" value of the last input was the j^{th} element of the uncertain domain (e.g. Office1, Lab1, *etc.*). The entries of Q sum to 1.0. Figures 2.4 (a), (b), (c), and (d) show an example event query, input stream, and the Q matrix after processing the query on instants 0 and 1 of the input stream, respectively.

The algorithm computes its updated state using each stream instant into a temporary matrix Q‘, which is then copied back into Q before the next update. Q' is computed as follows: For each entry $Q(i, j)$ (with value p_{ij}) of the state matrix, and for each possible uncertain value j' of the current uncertain input, the algorithm computes two values. First, it determines the state set i' that results from a transition out of the NFA states in set i on input value j'. This value depends on the NFA structure and predicates. Second, the algorithm determines the probability $p(j'|j)$: the probability of input j' conditioned on the previous input being j. This probability can be read directly from the temporal correlations in the input instant of the Markovian stream. The algorithm operator then adds probability p to the value of $Q'(i', j')$, where $p = Q(i, j) * p(j'|j)$.

Consider the event query and input stream shown in Figure 2.4(a) and (b), respectively. After

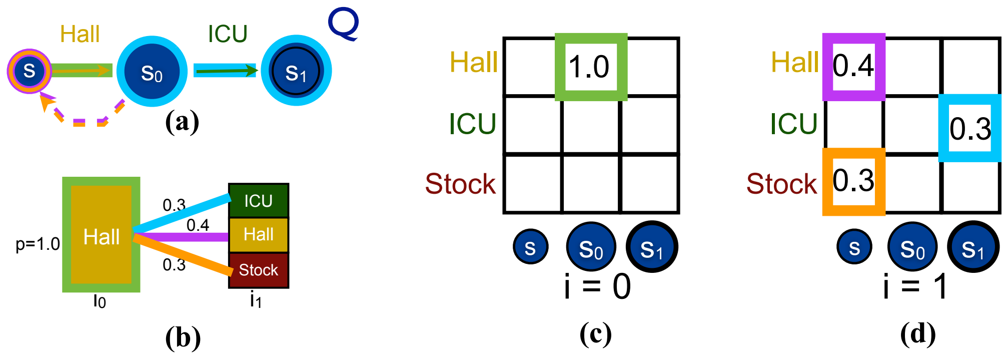


Figure 2.4: (a) Sample event query. (b) Sample Markovian stream snippet. (c) State of the matrix Q after processing the marginal distribution of i_0 in (b). (d) State of Q after processing the correlations linking instants 0 an 1 in (b). Color highlights show the correspondence between the NFA (query) edges activated by each stream element, and the resulting transitions in Q.

initializing Q with the marginal distribution at instant 0 (where $p(Hall) = 1.0$), the state of Q is as shown in Figure 2.4(c). Using the conditional distributions linking instants 0 and 1, Q is updated to the state shown in Figure 2.4(d).

After a given stream instant i is used to update Q, the probability that the query is satisfied at instant i is the sum of all entries in any row of Q corresponding to an NFA state set that includes at least one final (accepting) NFA state.

On each input, computation of Q' requires $O(2^M \times D \times D)$ multiplications, where M is the number of NFA states and D is the size of the Markovian stream's uncertain domain. This cost is quadratic in the size of the uncertain domain (a direct result of Markovian correlations) and exponential in the number M of states in the query NFA. In practice, the D terms are closer to $D' << D$, where D' is the size of the *active* domain, which is the number of domain elements at each timestep that have non-zero probability. Furthermore, the exponential 2^M term reflects the size of the query, which is generally not considered under standard data complexity analyses because it does not change with the size of the data.

2.3 Beyond Event Queries

Markovian stream analysis requires support for many types of query beyond basic event queries. Some of these queries are covered in this thesis, while others are beyond the scope of this work. This section gives a brief description of various types of Markovian stream queries that are not described elsewhere. These include aggregate queries (Section 2.3.1, which are a novel but minimal contribution of this thesis, since they can be computed using a simple variant of the basic event query processing machinery. Section 2.3.2 describes several query classes, first outlined in Ré et al.'s 2008 SIGMOD paper, which are increasingly sophisticated extensions of event queries and are beyond the scope of this thesis and not supported by Lahar. In addition to event queries, Lahar supports *lineage* queries, which are a novel contribution of this thesis and described in detail in Chapter 6.

2.3.1 Aggregate Queries

Markovian stream aggregates come in two flavors: intra-stream aggregates (Section 2.3.1, also called multi-stream aggregates), which execute on a single Markovian stream, and cross-stream aggregates (Section 2.3.1) which combine the results of a single query run over multiple Markovian streams.

Intra-Stream Aggregates

Lahar supports two types of intra-stream aggregation. The first, `EXISTS`, determines whether an event query is satisfied at *any* instant in a Markovian stream. `EXISTS` queries thus produce a single probability value after processing an entire stream. The second type of cross-stream aggregation is `COUNT`. `COUNT` queries return the *number* of instants in a stream that satisfy an event query. Of course, since each instant satisfies a query with some probability, the output of a `COUNT` query is a distribution over the possible count values.

Intuition suggests that these two aggregations might be computed as a post-processing step on the output of event query processing. Such an approach would be incorrect, however, since event query result probabilities produced for different stream instants are correlated (this is a consequence of the temporal correlations in the Markovian input stream).

Fortunately, intra-stream aggregations can easily be computed using a simple variant of the

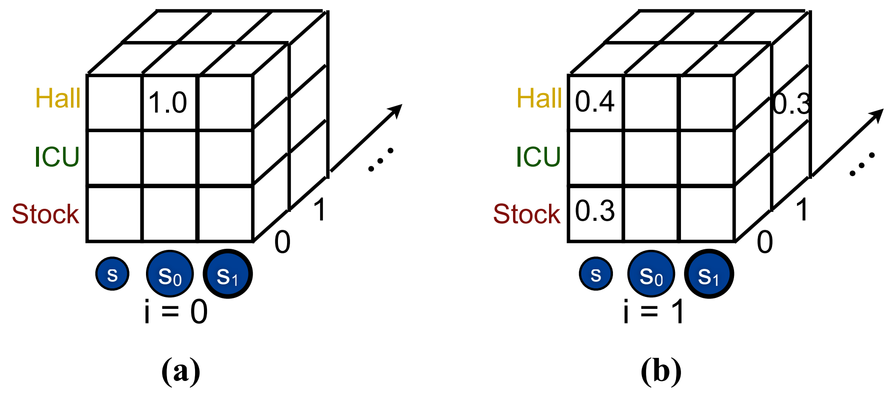


Figure 2.5: Q matrices augmented for intra-stream aggregation processing. (a) and (b) are augmented versions of Figures 2.4(c) and (d), respectively. Note that in (b), when probability mass enters the column of the query's final state s_1, the mass moves "backward" one step in the third dimension, incrementing the aggregate count value by one.

event query processing described in Section 2.2.2. In this variant, the matrix Q is extended from two dimensions $Q(i, j)$ to three: (i, j, k). The third dimension has an entry for each possible value of the aggregation. Figure 2.5(a) and (b) show the extended versions of the matrices shown in Figure2.4(c) and (d), respectively. Using the extended matrix Q, processing continues as described in Section 2.2.2, with one addition: when any probability is added into a matrix entry whose column label corresponds to a final NFA state (e.g. s_1 in Figure 2.5, its k value is increased by one (informally, the mass is "pushed back" one level in the matrix).

One important characteristic of this approach to processing intra-stream aggregates is that performance can degrade with the length of the input stream. The size of the third, k dimension grows with the size of the COUNT domain, which in turn can grow linearly (worst-case) with the length of the input stream. This scaling problem has been noted in prior art and can be avoided in cases where only summary statistics (e.g. the expected value) about the final count are required [61]. However, Lahar's current implementation always computes an exact count distribution.

Cross-Stream Aggregates

Lahar supports two types of cross-stream aggregation: `STREAMEXISTS`, which computes the probability that *any* input stream satisfies the event query, and `STREAMSUM`, which counts the number of streams that satisfy a query. Despite their similarity to intra-stream aggregates, cross-stream aggregates are much simpler to process because separate Markovian streams are assumed to be independent. For this reason, both cross-stream aggregates can be computed by first evaluating the basic event query separately on each input stream, and then aggregating the results as a post-processing step. For example, the post-processing step for a `STREAMEXISTS` aggregation over n input streams is: $p(agg) = 1.0 - \prod_{i=1}^{n} (1 - p_i)$.

2.3.2 Additional Query Types

Among the Markovian stream queries beyond the scope of this thesis are *safe* queries and *unsafe* queries [94]. Informally, safe queries join together the results of related but non-identical event queries processed on two or more different input streams. An example safe query is, *"Find all times when Sue and Bob entered room 355 together, and then Bob left the room without Sue."*. Safe queries are tractable in archived settings, but processing them requires an extended algebra beyond the scope of Lahar's regular-expression-based machinery. In contrast, unsafe queries are intractable. Intuitively, unsafe queries are characterized by sequence queries with "global" predicates that cannot be evaluated on a single stream instant. An example unsafe query is, *"Find all times when the crash cart stayed in the same room for 5 minutes."*. In this case, the "same room" predicate requires the location of the crash cart at time i to be compared to its location at time $i - 1$, *etc.*. which is a non-local (i.e. global) evaluation. Note that the same query reduces to a standard event query and is thus tractable if it is rewritten to specify a particular room (e.g. *"Find all times when the crash cart stayed in room 355 for 5 minutes."*). The original SIGMOD paper on event query processing includes a formal definition of unsafe and safe Markovian stream queries [94].

2.4 Summary

In this chapter, we first reviewed the Markovian stream model of imprecise sequences. The Markovian stream model represents a probability distribution over a set of sequences, and is characterized as a sequence of timesteps: Each expresses a marginal probability distribution over domain elements (e.g. locations), as well as a conditional probability distribution correlating domain elements at the given timestep and its immediate successor in the stream. We reviewed event queries, which are sequence-finding queries represented as NFAs whose edges are labeled with predicates over domain elements. Finally, we introduced aggregate variants of event queries, including cross-stream and intra-stream aggregates, and we briefly discussed broader classes of Markovian stream query, including safe and unsafe queries.

Chapter 3
THE LAHAR SYSTEM

In this chapter, we outline system architecture and implementation details relevant to anyone wishing to use or modify Lahar or the real-world data used in experiments in this thesis. Readers interested only in understanding the research contributions of Lahar may wish to skip this chapter.

3.1 LaharQL: Specifying Queries

Lahar currently supports a declarative query language that allows applications to create/drop tables (stream schemas), to load Markovian streams into Lahar from external locations, and to ask limited versions of event queries and lineage/aggregate versions of event queries. This language is not ideal, in that it cannot express the full set of queries that Lahar can answer (Lahar's programmatic interface allows for expression of the full range of supported queries). It is also not as intuitive or compact as it might be. Development of a more suitable language is left to future work; in this section, we outline Lahar's current language and highlight areas for potential improvement.

Lahar partitions streams based on schemas, which are analogous to SQL tables or relations: each schema is identified by a name (unique key), and contains data (here, streams) conforming to the schema. Lahar can store multiple streams adhering to the same schema as long as each stream is uniquely identified with a stream key (Lahar stores each stream adhering to a schema in a separate file). Two example schemas used in this thesis are an Audio schema, which requires that each stream instant contains a distribution over words (string objects), and an RFID schema, which requires that each stream instant contains a distribution over locations (represented as integer identifiers).

3.1.1 Creating and Deleting Schemas

We begin with an example Lahar statement creating a schema named 'RFIDLocation' with two columns named 'location' and 'velocity', respectively:

```
CREATE TABLE RFIDLocation (
location STRING,
velocity DOUBLE);
```

This statement creates a schema over a two-dimensional domain (location, velocity). Internally to Lahar, this schema is simply a directory containing metadata about the schema. When RFIDLocation streams are loaded, Lahar will check their schemas against the schema metadata, and will store each stream as its own file within the RFIDLocation schema directory. The schema conformity check ensures that each instant of an RFIDLocation stream defines a joint probability distribution over (location, velocity) as well as conditional distributions correlating these two values at the given instant and its successor instant. Data types supported by Lahar include INT, DOUBLE, STRING, LONG, and CHAR. Finally, Lahar does not support any further options for schema creation: specifically, it does not allow for default value assignments or constraint specifications such as null or consistency checks.

Lahar's drop table (schema) statement is analogous to that of SQL. An example is:

```
DROP TABLE RFIDLocation;
```

In Lahar, if a DROP TABLE statement is issued on a non-empty table, the schema and all of its contents are deleted.

3.1.2 Loading Streams

Recall from chapter 2 that Markovian streams are generated via a probabilistic inference process that is executed externally to Lahar. For this reason, Lahar's sole method of loading data is to copy an existing stream into the system using a statement like the following:

```
LOAD STREAM Bob
INTO RFIDLocation
FROMFILE <path-to-stream>;
```

This statement loads the existing stream, expected to be in location ⟨path-to-stream⟩ in Berkeley Database (BDB, a persistent key/value store) [82] format, into the RFIDLocation schema under the stream key 'Bob'. As mentioned previously, this loading operation comprises a consistency check between the RFIDLocation schema and the instants of the 'Bob' stream, and the creation of

```
1. SELECT [INSTANTS | EXISTS | COUNT]
2. FROM SchemaName
3.   [WITHKEY = 'stream-key1' [AND stream-key2'] [...]]
4. EVENT e1 [[NEXT | BEFORE e2] [NEXT | BEFORE e3] [...]]
5.    WHERE event-predicates
6. GROUP BY stream-key-predicates
7.    USING [STREAMCOUNT | STREAMEXISTS]
```

Figure 3.1: LaharQL syntax. Line 1 specifies temporal aggregations (if any). Line 2 specifies the schema of the target stream, and line 3 specifies the stream key(s). Lines 4-5 specify the event query NFA. Lines 6-7 specify cross-stream aggregations (line 7) and grouping predicates (line 6).

a new file 'Bob' within the RFIDLocation schema directory, where the new stream is stored. If the referenced stream file does not exist, or if the table being loaded does not exist or already contains a stream with the key being loaded, the statement aborts without loading the stream. Data not in BDB format, such as text files, must be converted into BDB format prior to loading.

Lahar's statement for deleting streams is similar:

```
DELETE STREAM Bob
FROM RFIDLocation;
```

Lahar does not currently support append operations, although such functionality would clearly be useful. This is not a fundamental limitation, but is simply a limitation of the current implementation (both in the functionality and in the language).

3.1.3 Querying Streams

We begin with an example of a basic event query (the full LaharQL syntax is shown in Figure 3.1). The query below asks for the probability that each instant in the RFIDLocation stream with key 'Bob' has value 'Office'; it returns this probability for each stream instant:

```
SELECT INSTANTS
FROM RFIDLocation
WITHKEY = Bob
EVENT e1
WHERE e1.location = Office;
```

In the first line of this example, the keyword `INSTANTS` indicates that this is an event query (as opposed to an aggregate or lineage query, discussed shortly). The second line specifies the schema of the stream begin queried, and the third line identifies its key. Lines 4-5 express the event pattern, which in this case is a simple one-state automaton (event query automata are discussed in Section 2.2) that is satisfied by any timestep in which the (imprecisely-known) stream element 'location' has value 'Office'.

Event query specifications can be more sophisticated, as in the following example:

```
SELECT INSTANTS
FROM RFIDLocation
WITHKEY = Bob
EVENT e1 NEXT e2 BEFORE e3
WHERE e1.location = Office1
  AND e2.location = Hall
  AND e3.location = Office2;
```

The last four lines of this query specify the NFA shown in Figure 2.2(b). The `EVENT` line of the query specifies by its use of three event labels e1, e2, and e3 that the query pattern is a 3-state NFA. The `NEXT` keyword indicates an automaton state with only an incoming edge; the `BEFORE` keyword, by contrast, specifies an NFA state that has an incoming edge *and* a self-loop edge, both of which share the same predicate. The ordering of the events in the `EVENT` clause specifies the ordering of the NFA states. The `WHERE` clause, which spans the last three lines of the above example, specifies the predicates used to label the edges of the query NFA. Lahar currently supports conjuncts and disjuncts (`AND`s and `OR`s) of equality predicates, each of which must be specified on a single event element (e.g. `e1` or `e2`; predicates comparing the values of different event elements such as `e1.location = e2.location` are not allowed).

One consequence of Lahar's use of the `NEXT`/`BEFORE` construct is that this allows only the specification of *linear* NFAs, which do not have branches (excepting self-loops). This limitation is a language limitation only; Lahar can process arbitrary NFA queries. Although we have found that in practice, most real-world queries are linear, Lahar's programmatic interface accepts NFA objects and can thus be used to process arbitrary NFAs. Overcoming the limitation of linear-only NFA expression in Lahar's language while maintaining an intuitive syntax is an area for future consideration.

Aggregate Queries

Lahar supports two classes of aggregate query—intra-stream aggregates and multi-stream aggregates—each described in Section 2.3.1. Intra-stream aggregates aggregate the number or existence of pattern match instances in a single stream, and are specified in Lahar's syntax using the `COUNT` and `EXISTS` keywords, respectively. The following example shows an intra-stream aggregate query which outputs the number (more precisely, a *distribution* over the number) of times that the sequence 'Hall'-'Office' occurs during Bob's day:

```
SELECT COUNT
FROM RFIDLocation
WITHKEY = Bob
EVENT e1 NEXT e2
WHERE e1.location = Hall
  AND e2.location = Office;
```

Lahar's multi-stream aggregates compute the number of streams in which an event query is satisfied, for each instant in the set of streams. The set of streams included in the aggregation is specified using predicates on stream keys. Currently, Lahar's syntax is limited to conjuncts of equality predicates on stream keys. The type of multi-stream aggregation is specified using the keywords `STREAMEXISTS` or `STREAMCOUNT`, which signal to return Boolean values indicating whether a query was satisfied in any stream, or a counts of the number of streams in which a query was satisfied, respectively. An example query asking for the number of streams in which the sequence 'Hall'-'Office' is satisfied at each instant in Bob's and Jane's day follows:

```
SELECT INSTANTS
FROM RFIDLocation
WITHKEY = Bob
  AND WITHKEY = Jane
EVENT e1 NEXT e2
WHERE e1.location = Hall
  AND e2.location = Office
USING STREAMCOUNT;
```

Finally, we note that intra-stream and multi-stream aggregations can be performed within a single query by combining one of the `SELECT COUNT/EXISTS` keywords with one of the `USING STREAMEXISTS/STREAMCOUNT` keywords. Figure 3.2 shows a schematic of the formats of

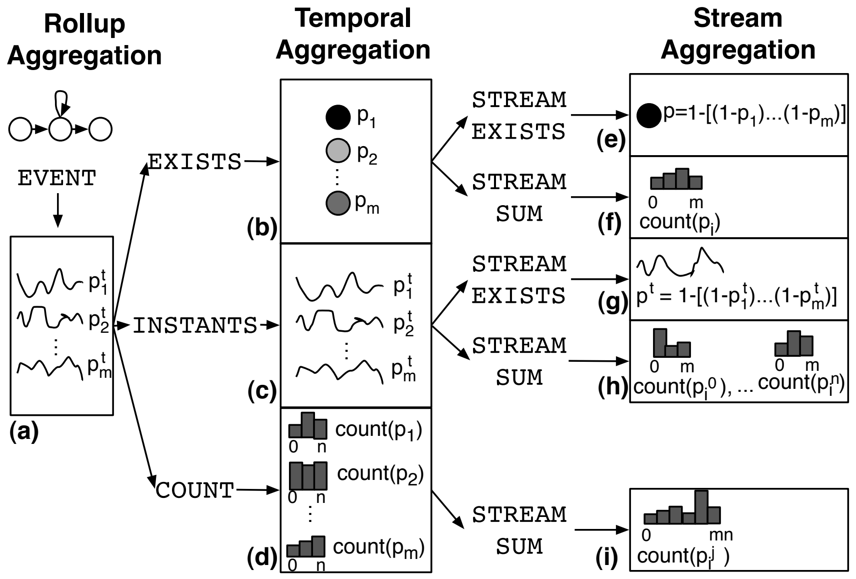


Figure 3.2: Lahar's aggregate query semantics for m streams of length n. (a) Standard event query output (query signal). (b-d) Temporal (intra-stream) aggregation semantics: `EXISTS`, `INSTANTS` and `COUNT` queries produce, respectively, single outputs per stream (b), a query signal per stream (c), and a histogram of query matches (counts) per stream (d). (e-i) Cross-stream aggregation semantics. `STREAMEXISTS` queries produce a single-valued result for each set of inputs, while `STREAMSUM` queries produce a histogram for each set of inputs. The combination of `COUNT` and `STREAMEXISTS` is not allowed.

query results under various combinations of these keywords.

Lineage Queries

Although we do not discuss lineage queries until Chapter 6, we note here that Lahar's syntax includes a simple `SELECT LINEAGE` construct for specifying lineage queries:

```
SELECT LINEAGE
FROM RFIDLocation
WITHKEY = Bob
EVENT e1 NEXT e2 BEFORE e3
WHERE e1.location = Office
  AND e2.location = Hall
  AND e3.location = Lab2;
```

This query returns the lineage (described in Chapter 6) of the event query shown in Figure 6.2(a). In Chapter 6 we discuss projection operations on lineage results, but these operations can currently be specified only through Lahar's programmatic interface.

3.2 System Architecture

Lahar's architecture is fairly conventional and reflects the organization of a standard DBMS. Figure 3.3(a) shows Lahar's architecture, which is implemented in a prototype of approximately 25,000 lines of Java code. An application interface receives input (expressed either in Lahar's language or programmatically) and passes it to the parser, which passes data update commands to the data loader, and translates queries into basic building blocks which are passed to the optimizer. The optimizer then creates a query plan, which it passes to the execution engine to process. Output from the execution engine is passed back to the application interface, which relays it up to applications. Lahar's storage manager coordinates all disk accesses. This section describes each component in detail.

Lahar is currently a single-threaded, single-process, single-machine system. The performance of many Markovian stream queries would benefit from a multi-threaded or distributed implementation, and the research questions involved in the development of such an implementation are discussed in Chapter 8.

3.2.1 Lahar API

Most human users interact with Lahar using its query language and an interactive terminal. Under this model, users type queries into the terminal and Lahar prints query results back to the screen. Applications wishing to post-process or visually display query results can alternatively use Lahar's programmatic API to provide more control over queries and the formatting of results.

The Java API comprises a basic set of capabilities, including functions to test for the presence of

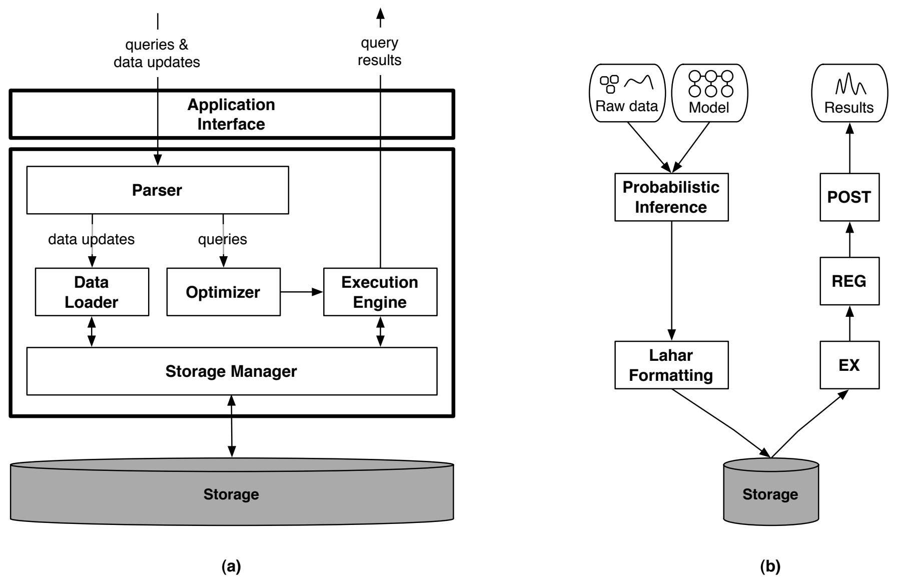


Figure 3.3: (a) Architecture of the Lahar system. (b) Data flow in Lahar. The three operators `Ex`, `Reg`, and `Post` are chained together into a query plan that is executed in the Execution Engine.

a particular table (schema) or stream key; functions to list existing schemas or keys of the streams adhering to a given schema; functions for creating or deleting schemas or individual streams; and several functions for executing a query. The most simple function for executing a query accepts a textual query string, while the most complex accepts many different arguments, including a query NFA object and a set of flags indicating whether projection, aggregation, *etc.*are to be applied. This latter function is particularly useful for applications wishing to process event query patterns that are not expressible in Lahar's current language, such as non-linear event queries. All of the API functions for query execution return a set of query results in memory, allowing applications to sort, print, visualize or otherwise post-process them as necessary.

3.2.2 Parser

Lahar's parser takes as input a text string expressing a LaharQL statement (create/drop table, load/delete stream, or query command). It produces as output a Java object called ParsedQuery, which contains the basic information needed to construct a query plan—this construction occurs in the Optimizer. Applications that interact with Lahar through the programmatic API bypass the Parser. Each API call constructs an appropriate ParsedQuery object in memory, which is passed to the Optimizer, at which point LaharQL commands and API commands follow the same pipeline. A ParsedQuery object includes information such as the name/key of the stream(s) being queried, an NFA representing the query pattern and its predicates, flags indicating whether any aggregation or lineage processing is required, *etc.*.

The LaharQL syntax is specified internally as an ANTLR [84] grammar, allowing Lahar to easily parse inputs using standard ANTLR tools. ANTLR provides basic support for parsing context-free grammars. Using this support, Lahar's parser checks that each input is syntactically correct, then converts text input into a parse tree. Custom Java code within the parser then converts each branch of the parse tree (i.e. the WHERE clause, the SELECT clause, *etc.*) into information required by the ParsedQuery object. Lahar's language is discussed in Section 3.1.

3.2.3 Data Loader

Lahar's data loader is responsible for loading a Markovian stream to disk. Recall that the stream being loaded must already be in the appropriate BDB format. Lahar does not currently support data updates, so the loader's primary functions are checking for overwrites, and checking each new stream for consistency (e.g. a stream's first instant must have seqID zero, all subsequent seqIDs must be consecutive, and the entries of each probability distributions must sum to 1.0).

The current no-update policy is a limitation of implementation—it is not a fundamental restriction imposed by any of the query processing techniques used in Lahar. Because Markovian streams must be created offline via post-processing of historical data, they are most commonly created and inserted into Lahar in batches. Applications desiring arbitrary inserts/deletes are exceedingly rare. However, an append-only model would be a useful abstraction for future versions of Lahar, to allow streams to grow naturally over time as new data is collected. In the current implementation, such appends can be achieved by deleting a stream and re-inserting a new copy that includes any desired appends.

3.2.4 Storage Manager

Lahar's storage manager mediates interactions between Lahar and data stored on disk. The storage manager keeps track of where each stream (identified by a stream key) resides on disk, and provides an interface to allow Lahar to retrieve one or more instants from a particular stream. The storage manager also creates and manages several indices for each archived Markovian stream, to optimize retrieval of stream instants based on various criteria (Section 4), and a set of approximate versions of each Markovian stream, for use in computing fast but approximate results (Chapter 5). Instead of a traditional system catalog, Lahar's storage manager maintains a directory structure that includes unique folders for each stream key, and unique locations within that folder for each approximated stream copy, index, lineage graph, *etc.*. This directory structure is hard-coded into the system and eliminates need for a traditional system catalog.

As the interface between Lahar and disk, the storage manager is responsible for determining the organization of Markovian streams with respect to each other, and also the organization of instants within each Markovian stream. The current implementation of Lahar creates a separate file for each Markovian stream, and within each file, it stores the appropriate Markovian stream and all indices into the stream. These files are actually BDB databases.

Within a particular stream, instants are stored in order of their sequence IDs, starting at zero, to optimize for scan-based accesses. There exist two reasonable alternatives for storing a single Markovian stream: the "coclustered" option materializes both the marginal and conditional distribution for each instant, and places them adjacent to each other on disk. The "separated" option again materializes both types of distribution, but stores all of the marginal distributions, in sequence order, in one location, and all conditional distributions, in sequence order, in a separate location. The "separated" option optimizes for workloads in which many timesteps satisfy queries, while the "coclustered" option optimizes for workloads in which queries are satisfied infrequently. For simplicity, Lahar's current implementation supports only the "coclustered" option.

3.2.5 Optimizer & Execution Engine

Lahar's optimizer creates executable query plans from the information extracted by the parser from raw input. The query plans are then passed to the execution engine, which executes the plan. Due to the structure of Lahar's query plans, both the optimizer and the execution engine are extremely lightweight. In order to explain this, this section begins with a description of Lahar's operators, continues on to describe the arrangement of these operators into a query plan, and finally discuss the optimizer and execution engine in the context of the query plan structures.

Operators

Lahar's query plans comprise three operators: `Ex`, `Reg`, and `Post`, each described next. Each operator implements the "getNext" interface, in which operator outputs can be retrieved from each operator one at a time by calling the getNext() function.

The `Ex` operator: The `Ex` ("Extract") operator is responsible for retrieving stream instants from disk. Different implementations of this operator use different access methods (stream scan, B+tree index, *etc.*), exposed through the storage manager interface, to load stream instants into memory. The output of the `Ex` operator's getNext() function is a single Markovian stream instant (including both a marginal distribution and the conditional distribution describing the correlations between the returned instant and the next).

The `Reg` operator: The `Reg` ("Regular expression") operator is responsible for evaluating a regular expression (equivalently, an NFA) on its input. These inputs are Markovian stream instants obtained from an `Ex` operator. The `Reg` operator generally produces one output for each input instant: this output is a single probability value indicating the probability with which the query's NFA is satisfied at the input instant. In the case of intra-stream aggregate queries (`EXISTS` and `COUNT`, described in Section 2.3.1), the `Reg` operator is also responsible for performing the intra-stream aggregation. In this case, the `Reg` operator produces a single output (as opposed to an output for each instant), and the value of this output is the value of the aggregation. The different behaviors of the `Reg` operator for different circumstances are written internally as different implementations of a single `Reg` operator interface. At runtime, the query optimizer is responsible for instantiating the correct `Reg` operator according to the query specification.

The `Post` operator: The `Post` ("Postprocessing") operator performs any necessary post-processing on the output of the `Reg` operator. In some cases, this post-processing involves computing aggregates across multiple Markovian streams. In other cases, the post-processing is devoted to computing or pruning lineage. In simple cases, when no post-processing is necessary, the `Post` operator simply pipes its inputs straight through to output.

Query Plans

Every Lahar query plan contains at least one instance of all three operators, arranged into a pipeline as shown in Figure 3.3(b). Queries that aggregate over multiple streams yield tree-shaped plans rooted at a single instance of the `Post` operator, which receives input from a separate instance of the `Reg` operator for each stream. In either case, data flows from the `Ex` operator to the `Reg` operator to the `Post` operator. The output of the `Post` operator is returned to applications.

Optimizer & Execution Engine

Lahar's optimizer is trivial, in part because of the rigid structure of Lahar query plans. In a relational DBMS, optimizers have the freedom to reorder operators in a query plan (i.e. to push selections down so they are applied before joins). In Lahar, the operator order is fixed. Thus, in Lahar, the only opportunity for optimization is in the choice of which implementation of each operator to instantiate. Different implementations of the `Ex` operator, in particular, use different access methods (indices), or pull data from streams approximated in different ways. In Lahar's current implementation, the optimizer chooses approximations based upon explicit instructions given programmatically by applications, with automatic optimization of this choice being an area for future work (Section 8).

Lahar's execution is simplistic for the simple reason that each operator implements the getNext() interface. To execute a query plan, the execution engine simply calls getNext() on the plans' root operator (always a `Post` operator) until it returns an empty value. Each operator in the plan has a handle to its child operator(s), and calls getNext() on these children to source it own input data.

3.3 Data Sets

The real-world experiments in this thesis are based on Markovian streams derived from two real-world data sets. This section describes both the raw data and the process by which Markovian streams were inferred from this data.

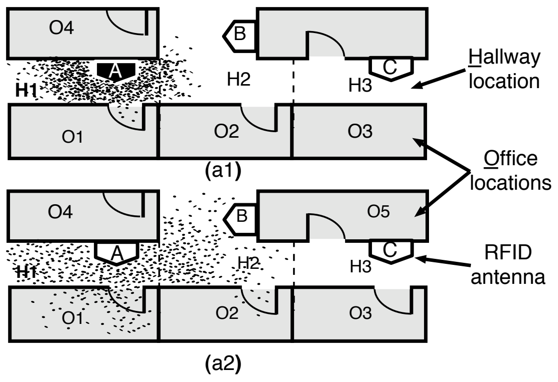
3.3.1 Office-Based RFID

The primary motivation for most of the work done on Lahar was the desire to draw additional utility from RFID data collected by the RFID Ecosystem at the University of Washington [116]. The RFID Ecosystem is a deployment of 160 RFID antennas (40 readers) installed in the hallways of the six-story Allen Center, which houses the Computer Science & Engineering department. Since the data used in this thesis was collected, antennas have been added into some stairwells and a small number of labs as well. Many faculty, students, and staff affix small RFID tags to their books, laptops, keychains, *etc.*, whose locations are then detected and logged by the readers. Although these users are primarily motivated by access to real-time object- or friend-finding applications [125, 80], the data generated by this system use is a rich source of historical data that can be mined using Markovian streams.

Data collected by the RFID Ecosystem is logged in a standard relational database. RFID readings are stored in a TRE (T̲ag R̲ead E̲vent) table with schema $\langle Timestamp, AntennaID, TagID\rangle$. Separate dimension tables store information linking antenna IDs to antenna location information, and tag IDs to (privacy-preserving) information about the type of object the tag is attached to, or who it belongs to. The set of all readings in the TRE table that share a tag ID comprise the "evidence" from which a single Markovian stream is inferred.

Inference of the RFID-derived Markovian streams used in this thesis was performed using a *particle filter* [39] that we implemented in Java. Particle filtering is a sampling-based technique for performing probabilistic inference. In this approach, samples (called particles) are used to represent the distribution over a tag's location. At timestep 7 (Figure 3.4(a1)), Bob (or more precisely, Bob's RFID tag) is sighted by antenna A and so his location distribution is tightly concentrated. To update Bob's location for timestep 8, the particle filter first predicts the location of each particle at the next timestep based on its current location (but independently from the locations of other particles). It then resamples the particles, choosing with a higher probability those particles whose locations are more consistent with the sensor readings (e.g. those particles that are within the read range of an antenna that sighted the tag). Because Bob is not sighted by any antennas at timestep 8, his location distribution is correspondingly diffuse (Figure 3.4(a2)).

Both the particle motion and the notion of sensor-location consistency are defined by a hid-



tag	t	l	$p(l)$
Bob	7	H1	0.80
Bob	7	H2	0.15
Bob	7	O1	0.05

(b1)

tag	t	l	$p(l)$
Bob	8	H1	0.50
Bob	8	H2	0.30
Bob	8	O1	0.15
Bob	8	O2	0.05

(b2)

tag	t	t'	l	l'	$p(l'\|l)$
Bob	7	8	H1	H1	0.625
Bob	7	8	H1	H2	0.250
Bob	7	8	H1	O1	0.125
Bob	7	8	H1	O2	0.000
Bob	7	8	H2	H1	0.000
Bob	7	8	H2	H2	0.666
Bob	7	8	H2	O1	0.000
Bob	7	8	H2	O2	0.333
Bob	7	8	O1	H1	0.000
Bob	7	8	O1	H2	0.000
Bob	7	8	O1	O1	1.000
Bob	7	8	O1	O2	0.000

(c)

Figure 3.4: (a1) Particle filter state in an instant *i* in which antenna A detects the tag. (a2) Particle filter state at instant *i* + 1.

den Markov model (HMM) [87]. The HMM's motion model expresses distributions over normal movement speeds, and also constraints on motion paths (e.g. tags cannot move through walls). The HMM's sensor model encodes characteristics of RFID reader ranges and noise characteristics.

A marginal probability distribution over a tag's location at a given timestep is computed by dividing the number of particles in each discrete location by the total number of particles. Relational tables representing Bob's location distribution at timesteps 7 and 8 are shown in Figures 3.4(b1) and (b2), respectively (note however, that Lahar does not use this relational format for internal storage).

The above filtering process can be applied in real time to create a Markovian stream from raw sensor readings; however, for applications interested in historical queries, an additional post-processing step called smoothing [87] can be applied to produce a more refined Markovian stream. Suppose that Bob is sighted by antenna C at timestep 9 (not pictured) of Figure 3.4. Given this sighting, in retrospect it is highly unlikely that Bob was in O1 at timestep 8. The process of smoothing revises past distributions (e.g. Bob's location at timestep 8) to make them consistent with future sensor readings (e.g. the sighting of Bob at C at time 9): in this case, by decreasing the probability that Bob was in O1 at timestep 8. For simplicity, in Figure 3.4 the filtered and smoothed marginals are identical, though in general smoothing changes marginal values as described.

The smoothing process also extracts a set of *correlations* between distributions at adjacent timesteps, capturing the likelihood of transitioning from one state to another. These correlations are captured using probabilistic constraints encoded in the HMM. For example, the correlations shown in Figure 3.4(c) reflect the fact that Bob cannot walk through walls: the bottom entry in the table states that the probability of Bob entering $O2$ at timestep 8 *given* that he was in O1 at timestep 7 is zero. Similarly, *given* that Bob is in H2 at time 7, he is most likely to remain there at time 8 since he is unlikely to abruptly switch direction and head back to O1. Thus the overall result of smoothing is a Markovian stream that contains non-trivial correlations and reflects reality more accurately than a filtered stream alone.

Many experiments in this thesis are based upon two sets of Markovian streams inferred from real-world RFID data: we call these data sets the *ambiguous* and *unambiguous* data sets. Each set comprises five traces of approximately 10 minutes (2MB) of data each. All traces reflect a subject walking through the halls of an office building, visiting several different rooms for around 30 seconds each. In the ambiguous traces, the rooms visited by the subject are indistinguishable from at least one other room, from the perspective of the RFID system. An example of indistinguishable rooms is the pair (Office1, Lab1) in Figure 2.1(a). Another example of indistinguishable rooms is the pair (O3, O5) in Figure 3.4, since an RFID reading at antenna C followed by no readings can be equally-well explained by the subject entering room O3 *or* room O5. By contrast, the rooms entered by the subject in the traces in the unambiguous set are all unambiguous–that is, it is largely straightforward to determine when the subject has entered these rooms. Room O1 in Figure 3.4 is an example of an unambiguous room, since an RFID reading at antenna A followed by no other readings has only one explanation: the subject has entered room O1.

Noise in the system prevents probabilistic inference from ever determining a subject's location precisely, even when the subject enters unambiguous rooms; however, the level of certainty about a subject's location is generally higher in the unambiguous traces. By contrast, when the subject enters rooms in the ambiguous traces, the Markovian stream generally reflects uncertainty about the room that was entered, specifying 2-3 different rooms with approximately equal probability. When the subject is walking through the hallways, the level of uncertainty is equal in both the ambiguous and unambiguous traces.

One hour of active readings from a single tag in the RFID Ecosystem, in uncompressed, text format fills up roughly 100-200KB of disk space—here, active means that the tag is detected at least once per minute. Thus one gigabyte of space can hold around 50-100 hours of raw RFID readings. On the other hand, one hour of Markovian stream data derived from these RFID readings requires around 12MB of space: the Markovian streams are 128 times *larger* than the raw data! The exact size of a Markovian stream varies depending on the amount of uncertainty in the data, but in the RFID case the relatively larger size of the Markovian stream is largely due to the highly compact nature of raw RFID readings. In the next section we demonstrate that, in a different domain, Markovian streams are more than an order of magnitude *smaller* than the raw data they represent.

3.3.2 Speech

Another domain in which Markovian streams are a useful model is speech processing, because a Markovian stream can compactly model uncertainty about which words were actually spoken in a given sentence or snippet which may be noisy. Lahar's utility in this context is demonstrated on a set of real-world Markovian streams inferred from spoken audio (five-minute NPR news updates). The uncertain domain of these streams is the set of words in the English language; an example of a stream from this domain appears in Figure 1.2(a).

Inferring a Markovian stream from an audio snippet is no trivial task. It requires detailed knowledge of acoustics as well as models of language: speech processing is a long- and well-studied problem. For this reason, Lahar's audio streams are inferred using Sphinx, an off-the-shelf speech processing tool [118]. Using generic English-language acoustic and language models (also off-the-shelf), Sphinx transforms an audio snippet into a *lattice*. A lattice is a graph structure, standard in speech processing, in which each node represents an utterance (a word along with its probability), and a directed edge connects two utterances that occurred consecutively in the audio input. The lattice is easily converted into a Markovian stream in which each path through the lattice becomes a unique path (possible world) in the Markovian stream. This is done by aligning the lattice nodes according to their sequence ordering, and normalizing the probabilities of edges exiting each node.

In the NPR newscasts used as audio input for Lahar's test data, a Markovian stream representing five minutes' worth of speech is roughly 1.3 MB in size (equivalently, one gigabyte an hold around 64 hours' worth of speech-derived Markovian streams), and has a length of around 800 instants. Most instants in the NPR audio streams contain only one or two words with non-zero probability; the most uncertain instant represents non-zero probabilities on seven different words: 'thin', 'things', 'theme' 'third', 'tougher', 'tough' and 'think'. Interestingly, the true word at this instant is 'the', which is not even captured as a possibility by the speech recognition software.

In contrast, the original audio data from which the 1.3 MB Markovian stream is derived is roughly 24MB: the Markovian stream is *nineteen times smaller* than the raw input data. Interestingly, in the speech domain, Markovian streams are compact relative to raw data; in the RFID domain, the reverse is true.

Chapter 4

INDEXING MARKOVIAN STREAMS

Recall that Lahar's basic event query processing algorithm uses a full, start-to-finish scan of the Markovian stream(s). Such scan-based approaches are natural for *streaming* evaluation, but they are highly inefficient for querying disk-based archives, which is the focus of this thesis. This chapter discusses several techniques for leveraging the disk-based nature of Lahar's input data via novel access methods to improve performance. The work in this chapter was first published in ICDE 2008 [73].

In this chapter, we observe that, for most Markovian stream queries, the fraction of stream instants that satisfy one or more query predicates is low. This situation is analogous to a relational database in which only a small fraction of tuples satisfy one or more query predicates. In such situations, indexes are commonly used to identify and retrieve the small number of tuples that satisfy a predicate, so that a full scan of the table can be avoided. In this chapter, we apply the same idea to Markovian streams. We first adapt relational indexes to Markovian streams, to allow efficient, targeted retrieval of only the set of stream instants that satisfies one or more query predicates. Although these indexes can accelerate processing of some queries (specifically, *fixed-length* queries), they cannot help Lahar avoid a full stream scan to process *variable-length* queries, in which large intervals of the input stream are relevant because they express correlation information relating two non-consecutive, possibly distant instants of an input stream. To accelerate these queries, we develop the novel Markov chain index that allows efficient lookup of precomputed correlation relationships between any pair of two instants in a Markovian stream. Concretely, the contributions of this chapter are as follow:

1. Access methods optimized for *fixed-length* queries, based upon a novel adaptation of standard indexing techniques.
2. A top-k optimization for *fixed-length* queries. This second access method exploits insights about the structure of Markovian event probabilities to adapt standard top-k pruning tech-

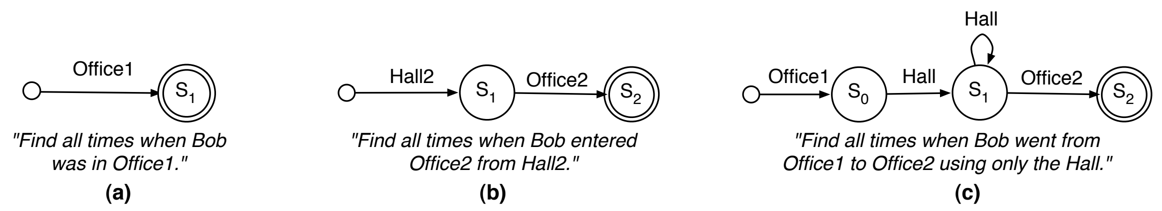

Office1
S1
"Find all times when Bob was in Office1."
(a)
Hall2
S1
Office2
S2
"Find all times when Bob entered Office2 from Hall2."
(b)
Hall
Office1
S0
Hall
S1
Office2
S2
"Find all times when Bob went from Office1 to Office2 using only the Hall."
(c)

Figure 4.1: Three linear event queries written for a location domain. (a) and (b) show fixed-length queries, while (c) shows a variable-length query.

niques to Markovian stream data, using standard B+ trees.

3. A novel index structure (the Markov chain index) and associated access method for *variable-length* queries, for which standard indexing is insufficient.
4. A discussion of practical issues relating to our three access methods, including predicate evaluation and a comparison of physical disk layouts.

In this chapter, we consider only *linear* event queries. Linear event queries yield NFAs that are linear (all query examples in this thesis are linear). Concretely, a linear event query is a concatenation of *links*, where each link is one of either 1) a single predicate (e.g. `Hallway`), or 2) a pair of predicates in which the first contains a Kleene star (e.g. ($\neg$`CoffeeRoom`*, `CoffeeRoom`). In the latter case, we refer to the second predicate as the link's primary predicate or simply 'predicate', since this is the predicate that transitions to the next query link. A list of links uniquely determines a linear query. We restrict our discussion to linear queries because they guarantee that the states of an NFA are visited in a fixed order, which simplifies the construction of indexing algorithms; furthermore, Lahar's current implementation supports indexing only for linear queries.

For the purposes of this chapter, (linear) event queries are divided into two classes: (1) *fixed-length queries*, whose NFA representations are loop-free, and (2) *variable-length queries*, whose NFAs contain loops. Figures 4.1(a) and (b) show fixed-length queries, while Figure 4.1(c) shows a variable-length query. The class of fixed-length queries can be satisfied only by fixed-length stream intervals (i.e. an m-link query can be satisfied only by stream intervals of length m). In contrast, intervals satisfying variable-length queries may span an arbitrary number of timesteps. As we will see in the following sections, this distinction allows for very different optimization of the queries in each class.

The work most closely related to Lahar's indexing techniques is the INDSEP index by Kanagal & Deshpande [62], which was developed shortly after the publication of material in this chapter. The INDSEP index builds upon and generalizes the MC index presented here (Section 4.2). Singh et al.'s index for categorical, imprecise data [112] predates the work in this chapter and is similar to the B+ tree indices we present here (Section 4.1). Singh et al.'s work assumes relational rather than sequential data and is thus designed to support select-project-join queries rather than event-style queries; however, the basic index structure is similar in both cases. Remaining work on indexing imprecise sequences focuses primarily on indices for supporting region-based queries in moving-object databases [18,27,114], which is a very different problem from the one tackled in this chapter. We discuss additional related work in Chapter 7.

The access methods in this chapter appear in Lahar as different physical implementations of the `Ex` operator (Section 3.2.5). They are compatible with the aggregation queries discussed in Section 2.3.1 and with the approximation techniques discussed in Chapter 5, but Lahar's current implementation does not support these access methods for processing lineage queries (Chapter 6).

4.1 Indexing Fixed-Length Queries

Fixed-length event queries can be processed using simple adaptations of standard indexing techniques. In particular, Lahar constructs a standard B+ tree secondary index on a Markovian stream. The B+ tree uses search keys of the form: `(attribute, instant-index)`. Here `attribute` is the uncertain stream attribute (or set of attributes) being indexed and `instant-index` is the stream sequence identifier (seqID). For example, a query plan for the Entered-Office query in Figure 4.1(b) might leverage an index on `(location, instant-index)`. We call this index BT_C (C for "chronological", since this is the ordering of keys sharing the same attribute value). An example is shown in Figure 4.2.

The purpose of the BT_C index is to allow Lahar to efficiently identify the set of instants satisfying a given predicate. For this reason, the BT_C indexes the domain elements present in each instant's *marginal* probability distribution, which defines the set of domain elements that are possibly true

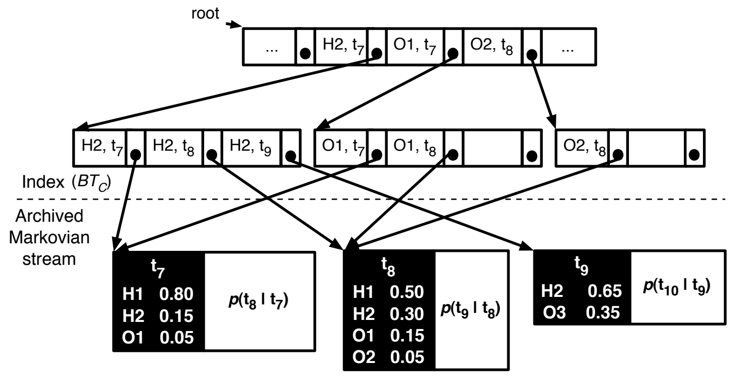


Figure 4.2: Bottom: Three timesteps' worth of the Markovian stream representing Bob's location, also shown in Figure 3.4(b-c). Top: A portion of the BT_C index on this stream. Note that the conditional distributions (correlations) are represented only symbolically here, because the BT_C index only indexes the marginal distributions of each instant the Markovian stream.

at a given instant, rather than an instant's conditional distribution, which defines correlations between domain elements at the given instant and the next. In Lahar's data representation, both the marginal and conditional distributions for each instant are stored together; thus, once an instant is identified by the BT_C as relevant, both the marginal and conditional distributions of the instant can be simultaneously retrieved from disk.

Because each timestep represents a distribution over states, each timestep can appear multiple times in this index. For example, timestep 7 in Figure 4.2 appears in three index keys, with locations H2, O1, and H1 (not shown). Importantly, the index includes only value/time pairs with a non-zero marginal probability, since zero-valued marginal entries are not explicitly stored in the stream.

Recall that a fixed-length query Q_F comprising m states can be satisfied only by stream intervals of length m. The goal of indexing in this case is to efficiently identify these intervals. For each predicate r in Q_F, an index such as BT_C can be used to retrieve the set of stream timesteps satisfying r. Here, *satisfying timesteps* are defined as the set of timesteps in which predicate r is satisfied with non-zero probability.

A standard equijoin between the sets of timesteps satisfying various query link predicates is useless for identifying relevant intervals. Instead, Lahar requires a temporally-aware join that identifies contiguous *sequences* of timesteps. This join is implemented using a separate index cursor on each predicate r. These cursors are advanced forward in parallel while maintaining the relative offsets required by the ordering of predicates in Q_F. The cursors *intersect* when they together reference a sequence of m consecutive timesteps, in the appropriate query-specified ordering. Thus every *intersection* identifies a length-m stream interval that is a potential query match. This process is similar to the standard merge join, and shares the corresponding linear-time data complexity.

This access method, which we call the B+Tree approach, is shown in Algorithm 1. Lines 1-2 initialize cursors on the index entries satisfying the predicates of Q_F. In line 3 these cursors are advanced until they *intersect* on an interval `I`. Lines 4-6 process `I` through the `Reg` operator.

An example of the pruning done by the B+Tree approach can be seen on the stream segment pictured in Figure 4.2. On the two-link, fixed-length query shown in Figure 4.1(b), with predicates $(H2, O2)$, the cursors on $H2$ and $O2$ first intersect on the interval (t_7, t_8), which is passed to `Reg`. In contrast, the index entry $(H2, t_8)$ has no intersecting entry $(O2, t_9)$, so this interval is not retrieved from disk or passed through the `Reg` operator.

One convenient feature of Lahar's implementation of temporally-aware index joins is that it retrieves relevant intervals in chronological order. Overlapping intervals can thus be combined before invoking `Reg`. To see this, consider the fixed-length query $(O1, H2)$ on the data in Figure 4.2. The B+Tree algorithm will identify both (t_7, t_8) and (t_8, t_9) as requiring further processing. By instead passing the single, longer interval (t_7, t_9), through `Reg`, it can avoid double-processing of timestep t_8. Thus on the densest data sets, the B+Tree approach gracefully degenerates into a naïve stream scan, with additional overhead to handle the BT_C index cursors.

Top-K Optimization

The B+Tree algorithm efficiently computes the probability of every query match in a stream; however, recall from Figure 6.1 that many of these matches are of low quality and are thus uninteresting to applications. The challenge in this case is that of *high-quality event retrieval*, in which only the top k query matches, or only those matches with probabilities above a given fixed threshold, are returned.

The key observation for efficient optimization of these queries is the following: within a length-

Algorithm 1 Fixed-Length Query Access Method (General)

Note: For simplicity, merging of overlapping intervals is not shown. For additional simplicity, we assume here that the `Ex` operator can make calls to the `Reg` operator, although in Lahar's implementation the getNext() interface of these operators prohibits such direct communication.

Input: Fixed-length query Q_F comprising m states, Markovian stream `M`, B+ Tree index BT_C on the single attribute of `M`.

Output: Probability that Q_F is satisfied at each timestep $t \in$ `M`

```
1: for each predicate $r_j$ in $Q_F$ do
2:     Initialize cursor $C_j$ on predicate $r_j$ in $BT_C$
3: for each interval I in the intersection of ($C_0$, ..., $C_m$) do
4:     Reg.startSequence(I[0].marginal)
5:     for each timestep $i$ in I do
6:         $\langle$I[$i$].$t$, $p\rangle$ = Reg.update(I[$i$].cpt)
```

m interval, the marginal probability that the i^{th} link predicate is satisfied at the i^{th} instant in the interval is an upper bound on the probability that the interval satisfies the query (i.e. that the query has a non-zero probability of being true at the last timestep in the interval). As an example, consider the fixed-length query $(H1, H2)$ (Figure 4.1(b)) on the interval (t_7, t_8) of the location stream shown in Figure 2.1(b). Both $p(H1$ at $t_7) = 0.8$ and $p(H2$ at $t_8) = 0.3$ are upper bounds on the probability that the interval (t_7, t_8) matches the query, which in this case is $0.8 * 0.25 = 0.2$. Intuitively, these marginal probabilities are upper bounds because a sequence of events cannot be *more* likely than any of its individual components.

This observation implies that Lahar can adapt the well-known Threshold Algorithm (TA) and its variants [40] to its problem. The basic idea is to process fixed-length stream intervals in decreasing order of marginal probability. For each interval, Lahar uses the maximum marginal probability among all query predicates to determine an upper bound on the probability that the interval will match the query; it then processes each interval in order of decreasing upper bound.

More specifically, Lahar supports an additional secondary B+ tree index on the Markovian stream, called the BT_P (p for "probability") index. It uses search keys of the form `(attribute,prob)`, where `attribute` is the uncertain stream attribute being indexed and `prob` is the marginal probability that the attribute value is true at the indexed timestep. Within the

BT_P index, keys sharing an attribute value are ordered in *decreasing* order of marginal probability. As with the BT_C index, the BT_P index includes only keys with non-zero probability.

Algorithm 2 outlines an access method (the top-k B+Tree approach) that leverages the BT_P index and the TA technique to efficiently process top-k queries. After initialization (lines 1-3), the algorithm scans in parallel all leaves of the B+ tree that match the different query predicates, returning the entry M[t] with the highest remaining marginal probability (line 4). The algorithm terminates when the maximum marginal probability of all remaining index entries (e.g. the marginal probability of predicate r_j in timestep M[t]) is below the probability of all k of the current best query matches (lines 5-6). If this condition is not met, the marginal probability of *each* predicate in the interval I is examined (lines 7-8). If none of these is low enough to prune the interval, the interval is processed through the Reg operator (lines 9-12) and the resulting probability is incorporated into the top k matches if appropriate.

As mentioned above, the marginal probability of each predicate in a length-m interval is only an upper bound on the probability that the interval matches the query; the actual match probability may be much lower or even zero. In data where this is common, the top-k B+Tree algorithm has little opportunity for pruning, and the B+Tree implementation of Ex based on the BT_C index, followed by a sort on the output tuples, will often outperform the top-k B+Tree approach because of its ability to combine the processing of overlapping intervals. The top-k approach, by contrast, significantly outperforms the standard B+Tree approach on queries with clear peaks, such as that shown in Figure 6.1. This tradeoff is further discussed in the evaluation (Section 4.3).

4.2 Indexing Variable-Length Queries

The fixed-length access methods in the previous section are inapplicable to variable-length queries, because variable-length queries can be satisfied by stream intervals of arbitrary length. A full stream scan can be avoided in this case using the following insight: while query match intervals may be arbitrarily long, generally only a small number of timesteps in each interval contain data *relevant* to the query (i.e., satisfy at least one query predicate with non-zero probability). Furthermore, the query NFA changes state only on these relevant inputs. In fact, the “irrelevant” intermediate timesteps require processing only because together they contain the correlation information relating

Algorithm 2 Fixed-Length Query Access Method (Top-K)

Note: For simplicity, we assume here that the `Ex` operator can make calls to the `Reg` operator, although in Lahar's implementation the getNext() interface of these operators prohibits such direct communication.

Input: Fixed-length query Q_F comprising m states, Markovian stream `M`, B+ Tree index BT_P on the single attribute of `M`, and k

Output: Top k timesteps at which Q_F is satisfied in `M`.

```
1:  currTopK.initializeEmpty
2:  for each predicate $r_j$ in $Q_F$ do
3:      Initialize cursor $C_j$ on predicate $r_j$ in $BT_P$
4:  for each timestep M[$t$] w/ max prob. referenced by any $C_j$ do
5:      if M[$t$].marginal.prob($r_j$)<= currTopK.min then
6:          Terminate
7:      I = {M[$t-j$], ..., M[$t-j+m$]}
8:      if I[$l$].marginal.prob($r_l$)> currTopK.min, $\forall\ 0 \leq l < n$ then
9:          Reg.startSequence(I[0].marginal)
10:         for each $i$ in I do
11:             $\langle$I[$i$].$t$, $p\rangle$ = Reg.update(I[$i$].cpt)
12:             currTopK.evaluate(p)
```

each relevant timestep to the next. Thus, processing might be optimized if we can develop an efficient (in both space and time) method for retrieving the correlations between distant stream timesteps.

This section introduces the novel Markov chain index (*MC*) that achieves this goal, along with an `Ex` implementation that leverages it.

Markov Chain Index

The Markov chain (*MC*) index is a hierarchical index that provides efficient lookup and/or computation of the correlations (also called CPTs, or Conditional Probability Tables) relating any two Markovian stream timesteps. The index stores a small set of precomputed CPTs organized in the hierarchy structure shown in Figure 4.3. The lowest level of the hierarchy ($i = 0$) is simply the

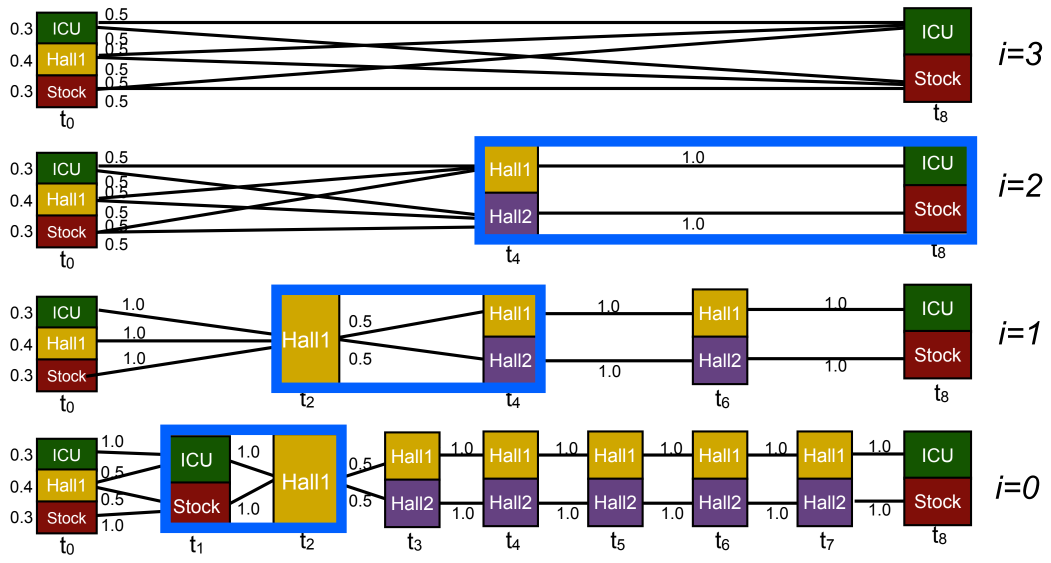


Figure 4.3: Markov chain index for $\alpha = 2$. The bottom row ($i = 0$) is the raw Markovian stream, in this case over an uncertain location domain (in a hospital). The three upper rows are the index. The index entries required to compute the CPT between timesteps t_1 and t_8 are highlighted; without the index, a scan of the Markovian stream would be required.

length-M Markovian stream. The density of additional levels of the tree is controlled by an integer parameter α. Each additional index level i contains a set of $M/(\alpha^i)$ entries, each of which relates a pair of timesteps separated by a distance of α^i. The total number of levels in the index is $\log_\alpha(M)$, and each index entry is the product of α entries of the index level below it. The example in Figure 4.3 is drawn for $\alpha = 2$; larger values of α decrease the storage requirements of the index.

CPTs not stored directly in the *MC* index can be computed as the product of existing entries, using the chain rule of probabilities: $p(t_j|t_i) = p(t_j|t_k)p(t_k|t_i)$. The upper bound on the number of CPTs that must be multiplied to compute a CPT spanning n timesteps is $2\log_\alpha(n)$, since at most two entries from each applicable index level must participate. In comparison, without the index this number would be simply n. In Figure 4.3, the two shaded index entries represent those whose product is the CPT relating t_0 and t_5.

Variable-Length Algorithm

The Markov chain index naturally yields a simple algorithm (Algorithm 3, which we call the MC index approach) for processing variable-length queries. In lines 2-3, a separate index cursor on BT_C is initialized on each query predicate. Line 4 advances these cursors forward in parallel, entering into the loop once for each timestep M[t] satisfying *any* of the query predicates. Upon the first entrance into this loop, execution jumps to line 6. Upon subsequent iterations, line 8 uses the *MC* index to lookup/compute the CPT between the previous relevant timestep (t_{prev}) and the current one (t). In line 9 this CPT is used to update the Reg operator. In this way the algorithm performs a single conceptual pass over the entire stream, but leverages the *MC* index to avoid retrieving large spans of irrelevant data from disk.

When a variable-length query contains loop predicates that are *not* negations of non-loop predicates (e.g. the Hall loop predicate of the query in Figure 4.1(c)), the MC index and algorithm as presented above are insufficient. In this case, the large stream intervals requiring summarization are not those containing *irrelevant* data, but instead are those that continuously satisfy the query loop predicate (*Hall* in the example of the previous sentence). These conditionals are not present in the MC index as described above; however, they can be captured for a given predicate (e.g. *Hall*) in a separate instance of the MC index whose entries are conditioned on satisfaction of the predicate. The details of the index construction and associated access method are a straightforward extension of those presented here for the general case.

In order to use the MC index approach to variable-length query evaluation, a BT_C index must be present for *every* predicate in the query, to guarantee that all relevant timesteps are identified. This is in contrast to the B+Tree algorithm for fixed-length queries, which computes correct results (though perhaps less efficiently) even when only a subset of query predicates is indexed.

4.3 Experimental Evaluation

In this section, we evaluate the B+ tree indexes and MC index through experiments with the prototype Lahar implementation described in Section 3. All experiments were conducted on a 2.0GHz Linux machine with 16GB of RAM.

They demonstrate on both synthetic and real data that standard B+ tree indexing techniques (the

Algorithm 3 Variable-Length Query Access Method (MC Index)

Note: For simplicity, we assume here that the `Ex` operator can make calls to the `Reg` operator, although in Lahar's implementation the getNext() interface of these operators prohibits such direct communication.

Input: Variable-length query Q_V comprising m states, Markovian stream `M`, B+ Tree index BT_C on the single attribute of `M`, Markov chain index MC

Output: Probability that Q_v is satisfied at each timestep $t \in$ `M`

```
 1: $t_{prev} = \emptyset$;
 2: for each predicate $r_j$ in $Q_V$ do
 3:     Initialize cursor $C_j$ on predicate $r_j$ in $BT_C$
 4: for each timestep M[$t$] referenced by any $C_j$ do
 5:     if $t_{prev} == \emptyset$ then
 6:         Reg.startSequence(M[$t$].marginal)
 7:     else
 8:         $cpt$ = MC.computeCPT($t_{prev}$, $t$)
 9:         $\langle$M[$t$].$t$, $p\rangle$ = Reg.update($cpt$)
10:     $t_{prev} = t$
```

B+Tree [Algorithm 1] and top-k B+Tree [Algorithm 2] methods) provide orders of magnitude improvements in performance over a naïve stream scan on fixed-length queries, even while preserving the probabilistic, correlated relationships within Markovian streams. They further demonstrate that the novel Markov chain index provides the same magnitudes of speedup on variable-length queries.

4.3.1 Setup

We evaluated Lahar's indexing methods using both synthetic and real Markovian streams. Each synthetic Markovian stream comprises 8 hours of data (30,000 timesteps). To maintain realistic properties, we constructed these streams by concatenating together various 30-second stream "snippets" generated from an RFID simulator reflecting the physical layout of the real-world RFID Ecosystem. In each snippet, a simulated person carrying an RFID tag walks down a short corridor, into a room where he stays for 15 seconds, and then back down the corridor. By altering the room labels in these snippets, we controlled the relevant properties of each stream with respect to each test query.

The real RFID data set was collected using the RFID Ecosystem. Eight volunteers carried 58 tags as they went through one-hour versions of typical daily routines (working in their offices, having meetings, taking coffee breaks, etc.). These routines visited locations across two floors, spanning an area of roughly 10,000 square feet discretized into 352 locations. The 17 antennas on each of the two floors were placed only in the corridors (offices, labs, etc. had no antennas inside them). Markovian streams were inferred from these RFID traces using the techniques described in Section 3.3.1. Eight of the traces were selected for use in the below experiments, based on their realistic reflection of location uncertainty (i.e., some tags were never or almost never detected and we did not use the resulting traces).

We found that, on the real RFID data set, the fraction of *relevant* timesteps (timesteps satisfying at least one query predicate with non-zero probability) exhibits bimodal behavior: either almost all or almost none of the timesteps in a stream are relevant to a specific query. We call this fraction of relevant timesteps *data density* and note that it is defined on a Markovian stream with respect to a specific query. Data density tends to be bimodal simply because the amount of time that a person spends in a given place tends to be bimodal. For example, the data density of queries involving a person's office tend to be high (0.75 and up) since the majority of a person's time is spent in his office. However, for queries about coffee rooms, other people's offices, *etc.*, the fraction of relevant timesteps is very small because the percentage of time a person spends near these places is low. Data density is an important parameter because it determines the relative performance of different access methods.

4.3.2 Performance: Fixed-Length Queries

In this Section, we first evaluate the detailed performance of fixed-length access methods on a two-link query, because it is a highly common type of query. We then present results generalized to fixed-length queries of different lengths.

Performance of Two-Link Queries

In order to establish the viability of the B+Tree approach, we first evaluate its performance on a two-link Entered-Room query (Figure 4.1(b)) on synthetic data, and compare the results to the naïve full

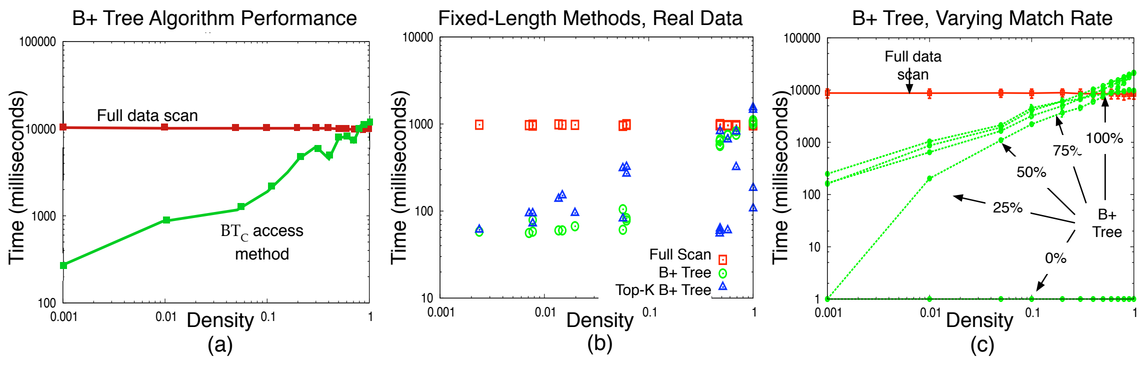


Figure 4.4: Evaluation of access methods optimized for fixed-length queries. (a) Worst-case performance of the B+Tree algorithm on synthetic data. (b) Performance of fixed-length approaches on real-world data. (c) Performance of the B+Tree algorithm on increasingly-favorable data sets.

stream scan. Figure 4.4(a) shows logscale performance numbers for both algorithms. As expected, when the data density is low, the B+Tree algorithm significantly outperforms a full stream scan—in this case, by 1-2 orders of magnitude. Conversely when most stream instants are relevant to the query, the B+Tree approach degenerates into a full data scan with additional overhead for the B+ tree lookups. Note that the B+Tree algorithm performance here is worst-case, because in this data set, *every* relevant instant (instants that satisfy at least one query predicate) participates in a valid query match. This property reduces the amount of pruning and increases the amount of disk I/O that the algorithm needs to do.

Figure 4.4(b) compares the performance of the naïve stream scan and the B+Tree and top-k B+Tree algorithms on a set of 22 different Entered-Room queries in one real, 28-minute RFID stream. Each of the 22 queries corresponds to a different room, and is responsible for three plotted points (one for each algorithm). Not surprisingly, the plot confirms the results shown in Fig. 4.4(a): the speedup of the B+Tree approach over a naïve scan increases as density decreases, providing improvements of at least an order of magnitude when this density is low. The lack of data points in the x range [0.1, 0.5] in Figure 4.4(b) reflects the bimodality of data density.

In the set of queries with low data density, the top-k B+Tree approach (Algorithm 2, plotted here for $k = 1$) performs poorly relative to the B+Tree algorithm. The sparse nature of the relevant data here provides the top-k B+Tree approach with little opportunity for additional pruning. Furthermore, the ability of the B+Tree algorithm to process overlapping intervals in a single pass often allows it to outperform the top-k B+Tree algorithm.

When the data density is high, however, the performance of the top-k B+Tree algorithm is often– though not always—much better (here, by an order of magnitude) than that of either alternative. The key feature allowing efficient pruning and therefore fast performance is the existence of a small number of sharp peaks in the query signal. The query signal shown in Figure 6.1 exhibits such behavior, and indeed this query is responsible for the top-k B+Tree point highlighted in Fig. 4.4(b) at $x = 1.0$. While the existence of such a peak cannot be precisely determined ahead of query processing (indeed, to do so would be to answer the top-k query), a reasonable heuristic is to try and use the top-k B+Tree algorithm for queries expected to have a high data density.

Figure 4.4(c) shows a more detailed, synthetic-data evaluation of the behavior of the B+Tree approach on an Entered-Room query. Each curve plots the performance of the algorithm when a different, fixed fraction of relevant timesteps participate in query matches. The curve for a query match rate of 100% is precisely the curve from Figure 4.4(a).

For any single curve in Figure 4.4(c), decreasing the data density (x-axis) decreases the number of query match intervals that the B+Tree algorithm must fetch from disk and send through the `Reg` operator, which increases performance. For a given data density on the x-axis, a decrease in the number of query matches causes a proportional increase in performance because some partial query matches can be identified early as dead-ends, allowing Lahar to avoid reading some instants from disk. When the fraction of relevant timesteps is low (e.g. 0.01), the difference between a 100% and 25% query match rate results in an order of magnitude difference in processing time.

Performance on Longer Queries

We demonstrate the scalability of the B+Tree algorithm on queries comprising more than two links using real RFID data. Figure 4.5 shows the results. Each of the three major columns of this table contains performance data for a real-world stream on an Entered-Room query written using 2, 3, or 4 links (real-world queries seldom require more than 4 links). The queries with 3 and 4 links require a tag's presence at multiple specific hallway locations outside a room before the room is entered.

		Stream: James Q: Entered-Office # States in query:			Stream: Sally Q: Entered-Office # States in query:			Stream: Pat Q: Coffee-Room # States in query:		
		2	**3**	**4**	**2**	**3**	**4**	**2**	**3**	**4**
	Stream length (minutes)	7.7	7.7	7.7	7.6	7.6	7.6	28	28	28
	Stream length (timesteps)	462	462	462	458	458	458	1683	1683	1683
	# relevant timesteps	149	194	211	5	5	6	33	53	253
	Time: Full Scan (msec)	548	625	1206	554	613	1201	977	1124	3034
FIXED	**# query matches**	26	26	11	2	1	1	3	1	1
	Time: B+ Tree (msec)	174	232	266	59	67	132	67	71	135
	Time: Top-K B+ Tree (msec)	56	75	214	67	98	165	95	366	1970

Figure 4.5: Statistics describing several streams from our real data set, and the speed of fixed-length queries with varying lengths on these streams.

The `Reg` operator slows exponentially with each additional query link, as can be seen in the fourth row of Fig. 4.5. Because the B+Tree approach is able to avoid many `Reg` updates, the performance of the B+Tree approach relative to a full stream scan (row 6 vs. row 4) improves dramatically on longer queries. The performance of the top-k B+Tree algorithm shows similar trends, although the relative performance gain is slightly less pronounced than for the B+Tree approach.

The results in the figure show the performance of the B+Tree and top-k B+Tree algorithms when indices serve to identify matching timesteps for all query predicates. In the case where some query predicates are not indexed, performance scales predominantly with the selectivity of the *intersection* of the available indices, independent of the number of predicates indexed. A second-order effect reflects improved performance when the number of predicate indices decreases, simply because the index overhead is reduced.

4.3.3 Performance: Variable-Length Queries

The performance of the MC index (α=2) approach versus a naïve stream scan, on synthetic data and the now-familiar Entered-Room query (a variable-length version, like that of Figure 4.1(c)), are shown in Figure 4.6(a). Data density is again the dominant factor in the performance of both algorithms, which exhibit the same trends as the B+Tree approach. Figures 4.6(a) and 4.4(a) are directly comparable.

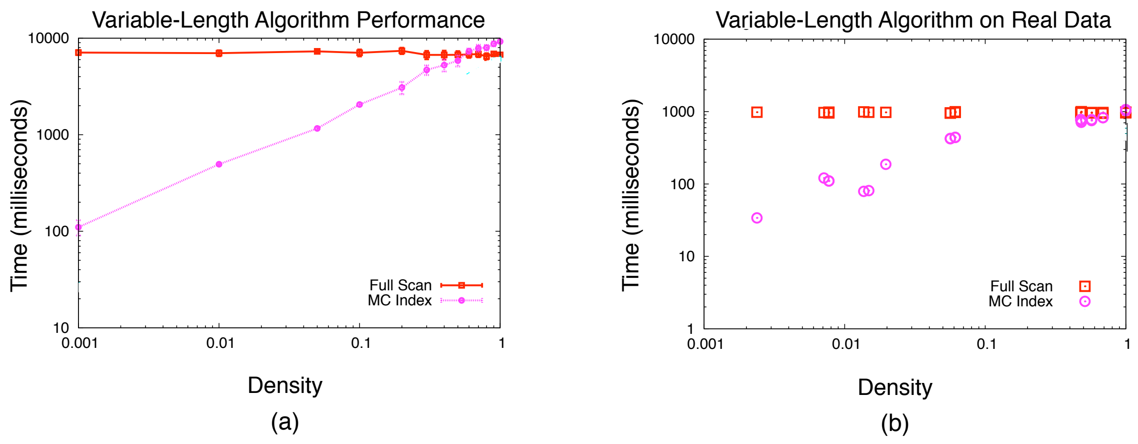


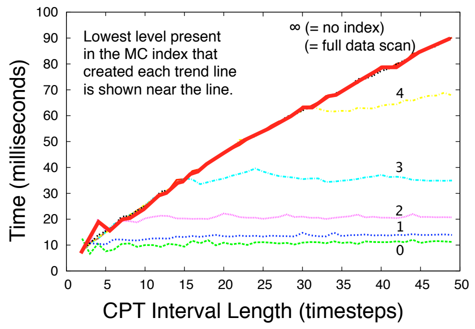
Figure 4.6: Evaluation of access methods optimized for variable-length queries. (a) Performance of the three algorithms as the number of relevant timesteps varies. (b) Performance on a real-world stream.

The performance of the MC index algorithm on real-world data is shown in Figure 4.6(b). As on the synthetic data in Figure 4.6(a), the MC index access method performance scales inversely with data density and outperforms the full data scan by more than an order of magnitude when density is low. The queries shown in Figure 4.6(b) are precisely the queries plotted in Figure 4.4(b), with Kleene closures added to make these queries variable-length. The data in these two plots is therefore directly comparable (note the fact that the naïve data scan reflects the same constant time in both figures).

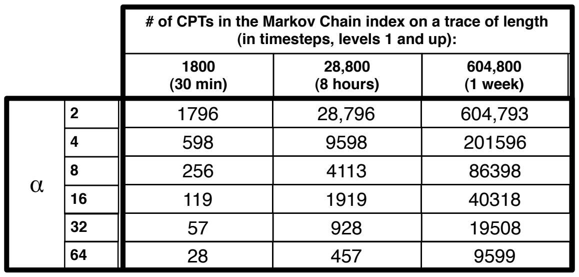
Markov Chain Index

The purpose of the Markov chain index is to provide efficient access to (precomputed) correlations between distant Markovian stream timesteps. Figure 4.7 provides details of the index's tradeoff between storage space and lookup speed.

Figure 4.7(a) shows the average time required to compute correlations between two timesteps separated by intervals of varying size. Because the placement of these intervals relative to stored index entries is important, reported results are the averages over all placements. Each higher curve in the graph represents performance when an additional index level is removed (this is a proxy for



(a)



		# of CPTs in the Markov Chain index on a trace of length (in timesteps, levels 1 and up):		
		1800 (30 min)	**28,800 (8 hours)**	**604,800 (1 week)**
α	**2**	1796	28,796	604,793
	4	598	9598	201596
	8	256	4113	86398
	16	119	1919	40318
	32	57	928	19508
	64	28	457	9599

(b)

Figure 4.7: (a) Time required to compute correlations across intervals of varying lengths using the Markov chain index. Each successive line plots performance when omitting an increasing number of lower index levels. (b) Storage requirements for the Markov chain index under varying α values.

increasing α) Clearly, correlations across intervals *smaller* than the span of the lowest available level of the Markov chain index can be computed only with a full scan, which accounts for the upper-bound behavior of the leftmost, naïve scan curve. The spacing between the flat portions of each i curve demonstrates that each additional index level reduces CPT lookup time by half.

In terms of speed, then, MC indices parameterized with lower values of α (roughly equivalent to the addition of increasingly low levels i in the above discussion) provide better performance by storing a greater number of precomputed conditionals. While this performance is gained at the cost of disk space, note that the most efficient parameterization, using $\alpha = 2$, merely doubles the stream storage requirements. Certainly many applications in the archived, warehouse-like setting targeted by Lahar will find this an acceptable tradeoff; however, disk-starved applications can leverage the MC index within their resource limitations by increasing α. Storage requirements for various α on streams of varying lengths are shown in 4.7(b).

4.4 Conclusions

In this chapter we presented access methods that enable Lahar to execute event queries efficiently over *archived* Markovian streams. Using novel indices, Lahar is able to selectively process only relevant parts of an input stream, thus achieving significant performance gains. At the same time, Lahar preserves result accuracy by retaining the Markovian properties of the stream while skipping data.

To achieve high performance, Lahar distinguishes different types of queries (fixed-length and variable-length). It then uses novel and efficient access methods specialized for each query type. For fixed-length queries, Lahar uses novel adaptations of standard B+ tree indices. For variable-length queries, it leverages a new type of index, the MC index, to effectively summarize unimportant parts of a stream. Additionally, Lahar supports top-k queries that effectively filter out noise in query results and can also improve performance in the case of fixed-length queries. Using synthetic and real data, we demonstrated that Lahar offers performance that can be orders magnitude better than that of a full stream scan.

Overall, effective techniques for managing noisy sensor data are important for many applications today and we view this work as an important step in this direction.

Chapter 5

APPROXIMATING MARKOVIAN STREAMS

Processing Markovian stream archives presents performance challenges not only because these archives can be large, but because these streams are imprecise and thus more complex to process than traditional, deterministic streams. Algorithms that rely on scanning an entire to stream to answer a query [94] are untenable in this setting. In the previous chapter, we presented indexing techniques that can help improve query performance. In this chapter, we study the complementary question of stream compression.

In this chapter we focus on compression techniques to improve performance, as is common in data warehouses [3, 89, 49]. The work in this chapter was first published in ICDE 2009 [73]. The performance benefits of compression are complementary to those of Markovian stream indexing, as both techniques can be used in conjunction with one another.

While traditional databases use *lossless* compression to improve performance, *lossy* techniques (equivalently, approximations) are a more promising avenue for Markovian stream warehouses. In addition to reducing the number of bytes required to represent a Markovian stream, judiciously-chosen lossy compressions can reduce the cost of event query processing by shrinking the size of the uncertain domain, or by reducing the complexity of the data representation. Applications that query Markovian streams are already accustomed to handling uncertainty, so they can often tolerate a small amount of additional imprecision incurred by the use of lossy techniques, which are the focus of this chapter.

Lahar accommodates the needs of different applications by materializing several different approximate copies of each Markovian stream in addition to the original. This allows applications or the Lahar optimizer to choose an appropriate approximate (or precise) stream at query time.

Existing work on approximation/compression for imprecise sequences is sparse. Some of the approximations studied in this chapter, including independence [103] and MAP [87] are well-known tools in the artificial intelligence community; however, their effect on the performance/accuracy of Markovian stream queries has not been studied. In this chapter we develop compressed data representations that can be queried directly without decompression, an idea which has appeared in other database systems (C-store [3], Blink [89, 90], Cypress [98]) but which has not yet been explored for imprecise sequences. We discuss additional related work in Chapter 7.

5.1 Approximations

The approximations used in Lahar are used to improve query processing performance. Importantly, reducing the size of a Markovian stream's disk footprint is important only insofar as it improves performance. Although a smaller stream footprint improves I/O time, we demonstrate shortly that the primary bottleneck in Lahar is actually processing power. Lahar is largely CPU-bound due to the quadratic complexity of handling correlations (Section 2.2.2). For this reason, the approximations introduced here focus primarily on producing approximate structures that are *faster* to process than standard Markovian stream instants, rather than focusing on compression techniques that make the instants *smaller*. Each approximate representation uses a different, simplified structure to represent the instants in a stream; each simplified structure is faster to process than a standard Markovian stream instant. Because the reduction of processor load is the key to improving performance in this context, none of Lahar's compression techniques require decompression—that is, each approximate Markovian stream instant can be processed directly in Lahar's query plans.

These approximations techniques fall into two categories: *numeric* and *semantic*, described in Sections 5.1.1 and 5.1.2.

5.1.1 Numeric Approximations

Numeric approximations do not consider the semantics of the stream's uncertain domain. These compressions include:

Independence compression simply removes all temporal correlations from a Markovian stream. The effect of this compression is dramatic (more precisely, quadratic), both in terms of disk storage and in terms of the effect on `Reg` performance. Independence compression is illustrated in Figure 5.1(b).

Consider a Markovian stream on an uncertain domain D: the correlations between two adjacent

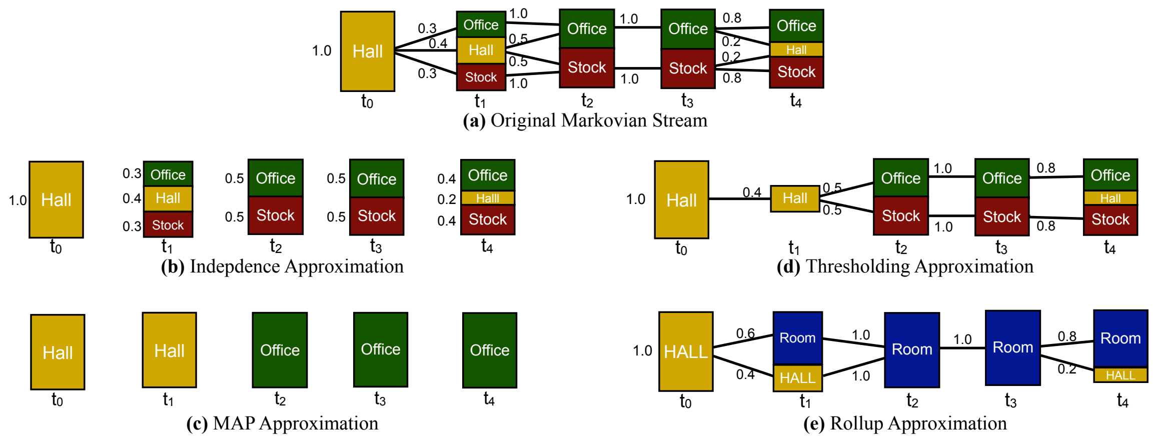

(a) Original Markovian Stream
(b) Indepdence Approximation
(c) MAP Approximation
(d) Thresholding Approximation
(e) Rollup Approximation
Hall
Office
Stock
Room
HALL
t0
t1
t2
t3
t4

Figure 5.1: A Markovian stream and four approximate representations.

timesteps of this stream form a matrix of size $|D| \times |D|$. After dropping these correlations, the independence-compressed stream view must store only the marginal distribution over the uncertain domain at each timestep. This marginal distribution has size $|D|$, yielding quadratic space savings.

Independence compression yields quadratic performance improvements in the `Reg` operator as well. Recall the state matrix Q (Section 2.2.2) that is the core of the event query processing algorithms: this matrix has a separate column for each of the $|D|$ elements in the Markovian stream's uncertain domain. Conceptually, each row of Q represents a different set of possible worlds; the i^{th} row of Q represents the possible states of the query NFA *conditioned* on the fact that the i^{th} domain element was the previous input (Figure 2.4(c)). The `Reg` operator must keep track of the previous input in order to correctly compute the temporal correlations between this previous input and the next input to arrive. If the input stream is independent, then the $|D|$ columns of Q are identical at all times, on all inputs. Clearly in this case there is no need for `Reg` to maintain a full matrix, and Q can be reduced to a single vector V. Thus, instead of the $M \times D \times D$ operations required to update Q, the `Reg` operator update on an independence-compressed input requires only $M \times D$ operations. This is a quadratic performance improvement.

MAP compression identifies the single most likely deterministic sequence (called the **M**aximum **a** **P**osteriori sequence) in a Markovian stream. It saves this single, deterministic stream and discards everything else. MAP approximation thus eliminates not only correlations (like independence), but it eliminates all uncertainty, as is illustrated in Figure 5.1(c). A MAP-compressed timestep requires constant space to store, since it contains only a single element from the stream's uncertain domain. A MAP-compressed timestep also requires constant time to process, since it is deterministic. Instead of a matrix or a vector, the internal state of the `Reg` operator on MAP input is simply a number that identifies the single *set* of states that the NFA is currently in. MAP approximation has a special significance since the MAP sequence is the most common way to deterministically represent an uncertain stream.

Thresholding compression uses a parameter T to prune/discard any temporal correlations (conditional probabilities) v such that $v < T$. This approximation is shown in Figure 5.1(d). The remaining values are normalized in order to maintain a consistent Markovian stream. Recall that Lahar implements null suppression—meaning that it does not explicitly represent elements with zero probability—so assigning zero probability to any element is equivalent to pruning the element from the stream. Clearly, higher values of T produce more aggressively-compressed streams. T values of 0.5 or higher produce deterministic streams in which the most likely element at each timestep has a normalized probability of 1.0. We discuss the trade-offs of various choices of T in Section 5.2.

5.1.2 Semantic Approximation: Rollup

In contrast to numeric compression, semantic compression is applied with an awareness of the semantics of Markovian streams.

Rollup compression uses a concept hierarchy to produce materialized Markovian stream rollups. These rollups represent the uncertain domain elements at a coarser level of granularity. For example, consider the uncertain location domain shown in Figure 2.1(a): {Office1, Lab1, Lab2, HallA, HallC}. One possible rollup compression on this domain is defined by a "Floor Plan" concept hierarchy that groups all room locations into a single concept and all hallway locations into another. The resulting rollup compression has the domain: {Room, Hallway}, as is seen in Figure 5.1(e).

Rollup views retain both the uncertainty and the temporal correlations from the underlying Markovian stream, so they require quadratic space to represent and quadratic time to process. However, these costs are quadratic in the size of a smaller domain. Rollups thus achieve performance gains according to the degree to which they collapse together the elements of the Markovian stream's uncertain domain.

A rollup view is equivalent to a Markovian stream produced by inference on a model (e.g. an HMM) defined at the granularity of the rollup. Thus rollup compression is lossless with respect to the coarse-grained underlying model. A rollup view is *not* lossless, however, with respect to the Markovian stream inferred using the original, fine-grained model.

To see this, consider the fine-grained office model in Figure 2.1(a), and suppose that the the dotted door between HallA and HallC is locked (Bob has no key). Suppose further that Bob's location in this model is uncertain: we do not know for sure which side of the door Bob was on. Any Markovian stream inferred using this model will assign zero probability to trajectories in which Bob visits both Lab1 and Lab2, since Bob cannot walk through walls or through locked doors. Now consider a rollup-compressed view in which all hallway locations (in this case, HallA and HallC) are collapsed into a single conceptual entity called Hallway. The trajectory Lab1-HallA-HallC-Lab2, which had zero probability in the uncompressed Markovian stream, is mapped to Lab1-Hall-Hall-Lab2 in the rollup-compressed stream. This latter trajectory may not have zero probability in the compressed stream, because the rollup compression discards correlations between entities that are collapsed together (e.g. it discards knowledge of the locked door between HallA and HallC that makes these locations negatively correlated in a single trajectory).

Rollup-compressed views are thus lossy with respect to fine-grained models, because some correlation information is discarded by the compression. However, rollup-compressed marginal values are not affected, and thus rollup compression can be used in conjunction with independence compression without incurring additional loss beyond that incurred by independence alone. The combination of independence and rollup compression is particularly advantageous because it can significantly reduce the size of the uncertain domain.

5.2 Experimental Evaluation

This section examines the impact of lossy compression techniques on both performance and the precision of query results, demonstrating that these techniques can yield large performance gains while altering result probabilities only minimally.

5.2.1 Setup

[Data] Real-world Markovian streams are critical to this evaluation of Lahar's approximations, which focuses on empirical rather than worst-case accuracy. The test streams are inferred from readings collected in the RFID Ecosystem, as described in Section 3.3.1. From over 6.6 million tag sighting events we curated two data sets which labeled *unambiguous* and *ambiguous*. These data sets are described further in Section 3.3. Each set contains five distinct RFID traces manually annotated with detailed ground-truth location information, for a total of 2.2 hours of Markovian streams (sampled at 1Hz).

The traces in both sets reflect a person walking around an office environment like the one shown in Figure 2.1(a), entering and exiting various rooms for brief (1 minute) intervals. The Markovian streams inferred from the unambiguous dataset contain significant uncertainty, but identify a single most likely room during each in-room interval; in contrast, streams inferred from the ambiguous dataset generally identify 2-3 most likely rooms with roughly equal probability. Temporal correlations are thus stronger in the ambiguous data. More concretely the conditional distributions in this data are less similar to independent distributions; the latter would include possible paths in which the person "hops" between different rooms at different instants.

[Queries] In this chapter, we evaluate Lahar on a set of queries designed to highlight the strengths and weaknesses of various compression techniques. We also evaluate the sensitivity of these queries to Markovian stream ambiguity/correlations. These queries search for the room-entry events present in the trace sets (specified using varying numbers of query links, according to the purpose of each experiment). For simplicity we restrict the evaluation to fixed-length event queries—these queries are more common and also more sensitive to correlations.

[Defining Error] Throughout this evaluation, we define error as the absolute difference between the exact probability of a query and the probability as computed on an approximated stream. An equally valid and interesting evaluation might compare query results on different compressed views with respect to their fidelity to the underlying ground truth sequence. Such an evaluation confuses two sources of imprecision, however: imprecision from the inference process and imprecision from the compression process. For clarity, we us the inferred Markovian streams as the exact baseline. As a point of reference, the error of randomly-chosen query probabilities in this framework is 0.5

Milliseconds per stream timestep (`INSTANTS` query)

Query Length / Rollup	1 Link	2 Links	4 Links	8 Links
Original (D=966)	**57.37** ± 43.45	**96.80** ± 68.70	**391.4** ± 298.3	**9924** ± 7356
Rollup View (C=4)	**4.36** ± 1.80	**6.37** ± 3.10	**19.12** ± 10.39	**256.3** ± 203.1
Rollup View (C=1)	**3.60** ± 0.91	**4.86** ± 1.20	**12.64** ± 2.25	**212.7** ± 10.9

Figure 5.2: Performance of Lahar using compressed stream views to compute exact query results for non-aggregated queries of varying lengths. For rollup views, the parameter C indicates the size of the rollup domain; thus, a rollup view with C=1 is not useful, but represents an upper bound on the performance benefits that can be achieved using rollup compression.

(the expected difference between two randomly-chosen values between 0.0 and 1.0).

5.2.2 *Efficiency of Lossless Compression*

Lahar can leverage compressed stream views to compute precise query results in two cases. First, Lahar can compute precise results (with respect to a coarse-grained data model, as explained in Section 5.1.2) using rollup-compressed views. Figure 5.2 shows Lahar's performance on a rollup view that reduces the stream's uncertain domain from 966 discrete locations (individual rooms, halls, *etc.*.) down to 4 coarse-grained location types (offices, halls, stairs, and "other"). The 241-fold reduction in domain size makes query processing 15-39 times faster (Figure 5.2 rows 1-2)—the apparent mismatch between the magnitudes of these reductions reflects the fact that in this experiment, the `Reg` operator was optimized to scale quadratically in the size of the *active* uncertain domain (instead of the full uncertain domain). The active domain in the experiments in Figure 5.2 was roughly 20, which is consistent with the 15-39x speedups. The case of C=1 reflects a stream with a domain size of one; while such a stream is not particularly useful, it provides an upper bound on the magnitude of acceleration that an application can expect to achieve using rollup compression.

Second, single-link queries, such as the one in Figure 2.2(a), evaluated *without* temporal aggregation (i.e. as `INSTANTS` queries but not `EXISTS` or `COUNT`) can be processed precisely on an independence-compressed view. Assuming independence doesn't affect these queries because they examine timesteps in isolation; however, if temporal aggregation is applied then these correlations are again required to compute precise results. The independence approximation accelerates query processing by approximately an order of magnitude, as can be seen in Figure 5.3.

5.2.3 Efficiency & Accuracy of Approximations

In this section, we examine in more detail the accuracy/performance tradeoffs incurred by Lahar's lossy approximations. We look first at the performance benefits of compression (Section 5.2.3), then at the effects on result quality (Section 5.2.3).

Performance

The mean latency of Lahar's `Reg` and `Ex` operators on various compressed stream views is shown in Figure 5.3, computed over a set of five 4-link `INSTANTS` queries. Queries with different lengths and temporal aggregation semantics show similar trends. Note that the y-axis is *not* in logscale, as small timing differences appear more clearly on linear axes.

Figure 5.3 clearly demonstrates that Lahar can dramatically reduce processing time by leveraging compressed stream views. Thresholded views accelerate performance by a factor of 2-7, depending on the threshold. Independence- and MAP-compressed views achieve performance boosts of one and two orders of magnitude, respectively.

The performance benefits of the independence- and MAP-compressed views stem directly from their reductions to the dimensionality of the `Reg` operator's state matrix. To see this, consider in Figure 5.3 `Reg` performance on the thresholded stream with T=0.5; this stream is deterministic. However, Lahar is able to process the independence view—which maintains full uncertainty within each timestep—2.3 times faster than it processes the "deterministic" thresholded stream, despite the fact that independence view is 1.45 times larger than the thresholded view! Because the determinism of the thresholded view is a side-effect, but not a guaranteed property, of the thresholded view, Lahar must still process it using a 2D state matrix Q that is an order of magnitude slower to update than the 1D state vector used to process the independence view. The smaller physical size of both independent and aggressively-thresholded streams is only a secondary factor contributing to the lower latency of processing these views.

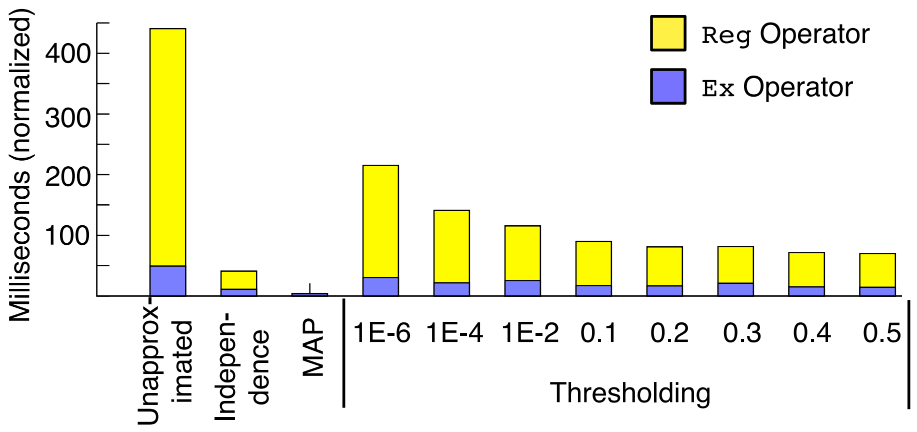


Figure 5.3: Unaggregated `Reg` and `Ex` operator performance on various compressed stream views. Performance on `EXISTS` and `COUNT` semantics is similar.

Result Quality

Lahar's performance-improving compression techniques are worthwhile only if they produce query results that can be used by applications. As we discuss in Section 5.2.3, compressed Markovian stream views can differ significantly from the original stream; however, applications care not about these stream-level differences but about their effects on query results. In this section we demonstrate that, empirically, Lahar can compute high-quality query results on lossily-compressed Markovian streams. Real-world RFID data is critical to this empirical evaluation.

The quality of Lahar's query results is measured in terms of the absolute difference between result probabilities computed on approximate and original Markovian streams.

Figure 5.4 shows the query-level probability differences on a single 4-link event query aggregated using `EXISTS` semantics (left column), `COUNT` semantics (right column), and using no temporal aggregation (middle column). The top and bottom row of the figure show query error on the unambiguous and ambiguous trace sets, respectively. For `EXISTS` and `INSTANTS` queries, the y-axis (error) is the average difference in query probability. For `COUNT` queries, which produce distributions over a count value, the error is computed as the Earth Mover's Distance (EMD) between these distributions [102]. Intuitively, an EMD value of n means that the count estimates of the two distributions differ by n. Finally, the quartiles of the `INSTANTS` plots (Figure 5.4 middle column)

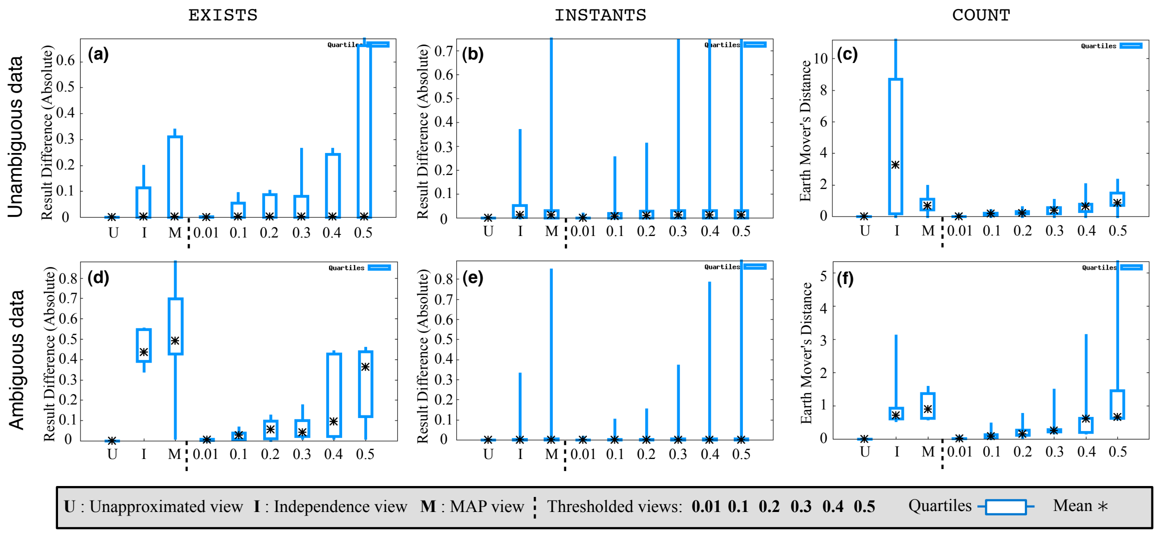


Figure 5.4: Differences in query result probabilities computed on exact and compressed Markovian streams. Differences are shown for each compression technique on a set of five 4-link queries with varying temporal aggregation semantics (Figure columns) and on two trace sets (Figure rows).

include only errors incurred on instants in which query probabilities are non-zero, since the vast majority of timesteps do not match the query and thus generate trivially-produced errors of zero magnitude. If plotted, obscure the magnitudes of the errors on the small number of timesteps when the query is actually satisfied with some probability.

Figure 5.4 shows that overall, differences in query results on compressed views are low. As thresholds increase, these differences naturally increase (on all types of data and temporal aggregation). The differences incurred by independence- and MAP-compressed views are less predictable, however, and since these are the views that yield the best performance, we look at these errors in more detail below.

Independence views ignore the temporal correlations in a Markovian stream. In these data sets, this generally results in overestimation of event query probabilities. Although the magnitudes of these individual over-estimates are small (Figure 5.4(b), (e)), temporal aggregation of many of these over-estimated values can result in larger differences (Figure 5.4(a), (c), (d), and (f)).

In addition to computing potentially over-estimated event query values, independence also causes `COUNT` queries to over-count. we verified that the EMD error shown in Figure 5.4(c) and (f) is indeed caused by over-counting. The over-counting error is higher on the unambiguous trace set (Figure 5.4(c)) than on the ambiguous trace set (Figure 5.4(f)) because the event query is satisfied in the unambiguous trace set with higher probability—causing Lahar to have a higher confidence in its over-counted values.

MAP views replace a Markovian stream distribution with the single most likely (deterministic) sequence in the distribution. The quality of query results on these views is thus dependent on the level of uncertainty in the data; highly uncertain datasets are poorly represented by a single estimate. In practice we found that even our unambiguous trace set contained enough uncertainty that MAP error varied widely across a wide range of queries; thus we abstain from drawing further conclusions about MAP, except to note that the unpredictability of its error is a potential liability. A Lahar administrator must carefully identify the datasets on which MAP views can be accurately used.

Stream-Level Error

As mentioned in the previous section, approximated Markovian streams can differ significantly from the original streams. We quantify these differences at the distribution (stream) level by computing the EMD distance between the pairwise joint distributions of each stream. These differences, computed over our entire RFID archive, are shown in Figure 5.5. Interestingly, while these stream-level differences and the query-level differences shown in Figure 5.4 follow the same trends, the *magnitudes* of the stream-level differences are much greater (`COUNT` magnitudes are not comparable). We surmise that the reason these differences do not propagate up to the query/application level is that our techniques are often trimming noise from the stream (e.g. thresholding may drop errant tuples). This means that, perhaps counterintuitively, the approximations studied here may in some cases improve Markovian stream quality (with respect to ground truth), although we do not test this hypothesis in our experiments.

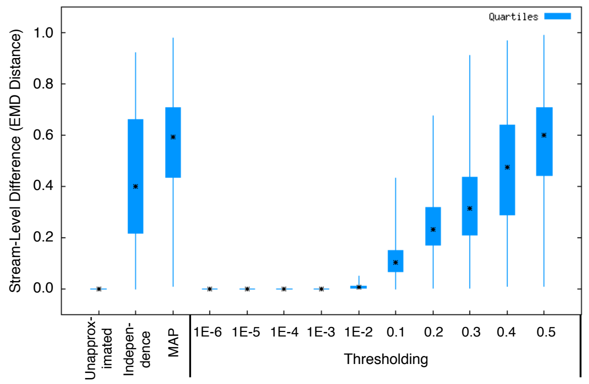


Figure 5.5: Distribution-level differences between Markovian streams and lossily-compressed views, computed over our entire RFID-based trace archive. Stream difference is computed using the Earth Mover's Distance on the joint distributions over each pair of consecutive stream timesteps.

5.3 Conclusions

In this chapter, we studied the performance/accuracy trade-offs of several standard approximations of Markovian streams, in an RFID-based location tracking domain. This study is the first to perform a cost/benefit analysis on imprecise stream models. We found that the trade-off space is rich, affording many opportunities for query acceleration with minimal impact on query error. We characterized query error on event queries and aggregated event queries, including count- and existence-based temporal and cross-stream aggregations. A detailed analysis of both performance and accuracy revealed that high-performance applications are well-served by a combination of MAP and independence models, while the needs of high-accuracy applications are best met by threshold- or rollup-based model.

Chapter 6

MARKOVIAN STREAM LINEAGE

While the previous two chapters have focused on making Lahar more efficient, we turn our focus in this chapter to making Lahar more expressive. In particular, we motivate the need for *lineage queries* on Markovian streams, and describe a set of techniques used by Lahar to support these queries.

Thus far we have focused primarily on *event queries*. Event queries are specified as NFAs, and include questions such as, "When did the crash cart move from a sterilization unit into the ICU?", or "Find all occurrences of the phrase 'overhaul health care'." Recall that the output of an event query is a list of Boolean values, one per instant of the input stream, specifying whether the query pattern is satisfied at that instant. In practice, because Lahar evaluates event queries on imprecise streams, the output is also imprecise. Thus Lahar answers event queries by producing a list of pairs (i, p_i), where i identifies an instant in the input stream and p_i identifies the probability that the event query is satisfied at instant i. An example of this output is shown in Figure 6.1. This output format is the same for all event queries, independent of the complexity of the pattern being matched.

Although event queries are powerful, they identify only the instants at which a query pattern is matched. They do not provide any information about *how* the input stream matched the pattern, *when* the pattern match may have begun, or *which* domain elements within the input stream caused the pattern to be matched. Many applications require this information, to determine for example which sterilization unit a crash cart came from, or how long it took to reach the ICU; or, in an audio domain, to retrieve the value of X in an occurrence of the phrase "X dollar project" or to retrieve spoken quotations between the words "quote" and "unquote".

Answering non-Boolean queries about *how* and *when* an event pattern was matched requires examination of the *lineage* of an event query. Informally, event query lineage is a record of how an event query is matched in a particular input stream, including information about when each query match begins; what subsequence of the input stream corresponds to the match; and what stream

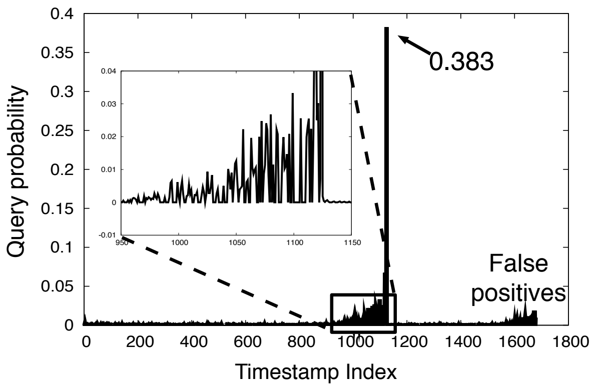


Figure 6.1: Output of an event query. The x-axis shows the instant index and the y-axis shows the probability with which the event query is matched at each instant i.

(domain) elements activate which query NFA transitions, and at what times. Figures 6.2(a) and (b) show an example event query and Markovian stream, respectively, while Figure 6.2(c) shows the lineage that describes the match of the query in (a) at instant i_4 of the input stream in (b). The lineage sequences in Figure 6.2(c) show that the query match began at either instant i_0 or i_1. The lineage sequences further identify the input stream sequences that cause the query match (one example is the sequence (Office1, HallA, HallA, HallB, Lab2) with probability 0.308), and the state that the query NFA is in after each transition (for the input sequence just listed, the NFA state sequence is (s_1, s_2, s_2, s_2, s_3). We defer a formal definition of lineage and description of Figure 6.2(d) to Section 6.2. Note that lineage is sufficient to reconstruct query execution: that is, given an event query's lineage, the original input stream contains no additional information relevant to the query.

The primary challenge to computing Markovian stream lineage is its size, which can grow exponentially in the length of the input stream. Recall that a Markovian stream of length N contains an exponential number D^N of unique sequences. Each of these sequences may generate a unique lineage sequence, like those shown in Figure 6.2(c), for *each* instant $0 \leq i \leq N$ in the stream. Enumerating the D^N lineage sequences that describe a single query match is intractable, and doing so for each query match in an input stream is even more unrealistic. Lahar addresses the exponential size of lineage by computing and storing lineage using a compact representation that leverages the Markovian nature of both the input stream and the lineage sequences. Intuitively, Lahar represents lineage using a Markovian structure similar to a Markovian stream, in which each element is augmented to represent not only a domain element from the input Markovian stream, but to also represent the state that the query NFA is in after processing the domain element. Figure 6.2(d) shows an example of this structure, which we call a *lineage graph*. Note that the lineage graph is not simply a subset of the input Markovian stream; lineage graph elements are augmented with NFA state information which is specific to a particular NFA and is thus not found in the input itself. We define the lineage graph structure formally in Section 6.2, but note here that its Markovian structure allows it to compactly represent an exponential number of lineage sequences for each instant in the input Markovian stream.

Although Lahar can efficiently compute and compactly represent an exponential number of lineage sequences, enumerating all of these results for applications is neither feasible nor desirable. Instead, Lahar takes a top-k approach that has become standard in systems that handle uncertain data [29, 113, 93]. In the context of Markovian stream lineage, the top-k approach allows applications to specify an integer value k, and obliges Lahar to enumerate as output only the k most likely lineage sequences associated with each query match. In most cases, a small value (less than 10) of k is sufficient for applications to obtain meaningful results, as we demonstrate in Section 6.5. Enumerating only the k most likely lineage sequences from a lineage graph requires that Lahar support ranked enumeration, which it does using a known variant of the Viterbi algorithm [41], described in Section 6.3.2.

In many cases, applications are interested in only a subset of the elements in each lineage sequence. An application asking the query in Figure 6.2(a), for example, may be interested only in knowing *which* office a person visited before entering Lab2, without caring about the sequence of hallways the person visited between the office and lab. In this example, the desired result is a set of lineage sequences in which elements corresponding to the Hall and Lab2 locations have been projected away, leaving only the elements associated with the Office predicate. The first lineage sequence in Figure 6.2(c) in this case projects down to a single-element result sequence (i_1:Office2(s_1)), while the second and third sequences project onto the same final result, (i_0:Office1(s_1)). Similarly, an application interested only in *when* a query match begins may project away all elements of lineage except for the first timestamp of the match.

Importantly, projection of lineage sequences must occur *before* top-k path enumeration, which requires that Lahar support projection internally: Projection cannot be left to applications. Consider the top-1 lineage sequence in Figure 6.2(c), in which Hall and Lab locations are projected away. Projection performed on the top-1 result (the first sequence in Figure 6.2(c), with probability 0.315) will yield (i_1:Office2(s_1)). When projection is computed *before* top-k sequence enumeration, the second and third lineage sequences in Figure 6.2(c) project down to the same final result, (i_0:Office1(s_1)), which has probability 0.308+0.0385=0.3465. This result is the correct top-1 projected result; note that it is different from the result computed by performing projection after top-k enumeration.

Lahar supports projection on lineage graphs using several different algorithms, which we describe in Section 6.4. Intuitively, these algorithms remove edges of the lineage graph corresponding to elements that are to be projected away, while maintaining just enough information to recreate appropriate connections between the remaining sequence elements. We introduce a basic algorithm that executes many scans of the lineage graph, followed by two optimized versions that each require only a single pass over the lineage graph. All algorithms produce as output a modified version of the lineage graph, allowing top-k enumeration to be executed after projection with no modifications to the enumeration algorithm.

We evaluate the performance cost of constructing lineage, projecting it (in 2 different ways), and of computing the top k lineage paths; we also evaluate the quality of top-k answer enumeration for small k, all on 2 real-world data sets. We demonstrate that constructing a lineage graph and enumerating the top k lineage sequences adds a query processing overhead of 3x or less using our single-pass algorithms, compared to standard Boolean query processing without lineage. We further demonstrate, through examples, the importance of supporting projection on a lineage graph, and we demonstrate that the overhead incurred by processing such projection is manageable and can be minimized by selecting an appropriate projection algorithm.

The Markovian stream lineage introduced in this chapter differs fundamentally from the lineage used in imprecise data management systems [24] such as Trio [6, 127] or prDB [63] because it is lineage for sequence (event) queries; event query lineage cannot be captured by the Boolean formulas used in existing systems. Similarly, lineage definitions and algorithms used in sequential data processing systems such as SASE [5] or the work of Shen et al. [111] are inadequate. SASE tracks the lineage of event queries on deterministic sequences, but defines lineage as subsequences of the input (as opposed to this thesis which defines lineage as subsequences augmented with NFA state information), and it does not handle imprecision. The work of Shen et al. takes an approach similar to SASE and handles imprecise streams but does not handle any correlations between timesteps, including Markovian correlations. The only work to study event query lineage on Markovian streams is recent work by Kimelfeld and Ré [68], which outlines a set of primarily negative results. Their work, however, identifies a set of Markovian stream transducers whose output is tractable. These transducers can capture all of the real-world lineage queries motivating our work on Lahar, so in this chapter we focus on such transducers. We develop and evaluate the first algorithms for computing tractable lineage (the work of Ré and Kimelfeld does not provide any algorithms for computing lineage) and enumerating top-k lineage paths. Additionally, in contrast to previous work, we demonstrate the importance of supporting projections and provide the first algorithm for doing so.

In the rest of this chapter, we first provide an intuitive overview of lineage and a set of Markovian stream lineage queries that we use as running examples through the chapter, and also in our evaluation (Section 6.1). We then formally define Markovian stream lineage and its representation in Lahar in Section 6.2. Section 6.3 introduces algorithms for constructing lineage and computing the top k paths for a query match, while Section 6.4 introduces algorithms for performing projection on lineage. We evaluate the performance of these algorithms in Section 6.5 and conclude in Section 6.6.

6.1 Motivating Scenarios, Overview, & Summary of Contributions

In this section, we outline a set of example queries, provide an intuitive overview of Markovian stream lineage, and finally summarize the contributions of this chapter.

6.1.1 Motivating Lineage Queries

Figure 6.3 shows a set of queries used here as running examples, and which we also execute on real-world data as part of our experimental evaluation. These queries are written to detect various movements of a fictional character Bob as he moves throughout an office building. These queries are all satisfied several times in our real-world RFID dataset, described further in Section 6.5. They are representative of queries that might be used in a hospital or home to detect movements of people, equipment, or objects.

Our four representative event queries are labeled Q_A, Q_B, Q_C, and Q_D, and are shown in Figures 6.3(a), (b), (c), and (d), respectively. Each frame of the figure shows the DFA that describes the event query, an English-language description of the sequence detected by the query, and a schematic diagram of what the query's lineage might look like on a real-world input stream (not pictured). These queries are useful for applications trying to learn general usage patterns for a building, or trying to learn about the daily movement patterns of individuals. We selected these queries in particular for two reasons: First, they each yield lineage graphs with different characteristics, allowing us to demonstrate the effects of various lineage properties on query performance. Second, elements of the lineage graphs of each of these queries can be projected in various ways to support novel analyses, allowing us to demonstrate both the utility and the performance of our lineage projection algorithms.

Two projection-based variations of each basic query Q_A, Q_B, Q_C, and Q_D can be seen in Figure 6.3(e), (f), (g), and (h), respectively. Projection-based variations of each basic event query Q_X are labeled Q_{X1} and Q_{X2}. Each frame of Figures 6.3(e-h) shows English-language versions of the two projection-based variants of the corresponding event query, along with a diagram of what the query's lineage looks like after the appropriate projection has been applied. Query DFAs are not redrawn because they are the same as the DFAs for the basic event queries.

Consider as an example queries Q_{A1} and Q_{A2}, both of which are projected versions of Q_A (*"Find all times when Bob went from one room to another"*). Query Q_{A1} asks about the identity of the rooms visited by Bob, but projects away all information about the halls Bob used to move between the rooms. This projection is reflected in the lack of lineage elements in the projected graph corresponding to Hall elements. Query Q_{A2} represents the inverse projection of Q_{A1}: it asks about the path Bob took between two rooms, but projects away the identity of the rooms themselves. As a final example, consider Q_{C1} which projects away all elements of the lineage graph, leaving identifiers only for the start and final instants of the query match. Such aggressive projection is useful for applications wishing to determine the length of a query match, which in the case of Q_{C1} represents the length of time that Bob spent in room A. We define more precisely in Sections 6.2 and 6.4 the types of lineage queries and projections supported by Lahar.

6.1.2 Intuitive Overview of Lineage

Intuitively, we define the *lineage* of a regular 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 (starting with the DFA start state and ending at a final state). An example DFA query Q, input Markovian stream M, and lineage of Q on M (for instant i_4) are shown in Figures 6.2(a), (b), and (c), respectively. Each lineage sequence comprises a domain element from the input stream and a DFA state. The set of lineage sequences for 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.

The lineage definition we propose here is a natural definition. It is analogous to "how" lineage in a relational setting [24] because it identifies not only the set of input stream subsequences that match a query, but also the operations (DFA edge transitions) used by each subsequence to create a query result. This definition of lineage is equivalent to the output of an "indexed s-projector" as defined in recent work by Kimelfeld and Ré [68], who identify this lineage format as the only tractable type of the many alternatives they explore.

6.1.3 Summary of Contributions

The lineage work in this thesis extends the results of Kimelfeld and Ré in two ways: first, by identifying and supporting a broader class of projections than the indexed s-projectors defined by Kimelfeld and Ré, and second, by providing and evaluating algorithms for producing lineage. Specifically, Section 6.4 demonstrates support for *arbitrary* projections on Markovian stream lineage, while the work of Kimelfeld and Ré is restricted to projections of lineage prefixes and suffixes only.

Concretely, the contributions of this chapter are as follow:

1. A definition of lineage for event queries on Markovian streams, and a compact *lineage graph* representation for this lineage.
2. Algorithms for constructing lineage graphs and enumerating the top-k lineage sequences for a query match, in time linear in the stream length.
3. A definition of lineage projection, and quadratic algorithms (in the stream length) for applying

projection.

4. An experimental evaluation of our lineage processing algorithms on real-world streams inferred from RFID data.

6.2 Defining Lineage

In this section, we define the set of queries for which Markovian stream lineage and provide a formal definition of lineage. We also introduce a compact data structure, called a lineage graph, used to represent lineage efficiently. Finally, we outline a set of measures used to characterize the event query lineage; these characteristics are used to describe lineage in our experiments (Section 6.5).

6.2.1 Unambiguous DFA Queries

As shown by Kimelfeld and Ré [68], Markovian stream lineage is in general intractable to compute for arbitrary NFAs. However, we identify a class of queries that are common in practice, and for which efficient computation is possible. This is the class of *unambiguous, DFA* event queries, and these are the only queries we consider in the remainder of this chapter.

DFA queries are those whose finite automaton representations are deterministic—that is, they are expressed as DFAs rather than NFAs. Unambiguous queries are queries for which, on a *deterministic* input stream, the longest and shortest substrings that match the query at a given instant are the same. For example, the query pattern (RoomA, (¬RoomB)*, RoomB), which searches for instances of a person first in RoomA and then later in RoomB, does not meet the definition of unambiguous. On the deterministic input stream aaab, for example, the longest matching substring ending at instant 3 is 'aaab', while the shortest is 'ab'. However, this query expression can be rewritten into a similar query that is unambiguous: on the same input stream, the rewritten query (RoomA, (¬(RoomA ∨ RoomB))*, RoomB) yields the unambiguous match 'ab'. By definition, fixed-length queries are unambiguous. DFAs beginning with a repeated pattern, such as RoomA*, are not (other DFA structures can also cause ambiguity).

We introduce these constraints to achieve a definition of Markovian stream lineage that is as expressive as possible but also yields efficient algorithms for construction, projection, and top-k enumeration of lineage sequences. We require the DFA property because, as Kimelfeld and

Ré [68] demonstrate, Markovian stream lineage cannot be tractably computed for arbitrary NFA event queries. The unambiguous-query property ensures that each deterministic sequence encoded in a Markovian input stream produces only one lineage sequence (the DFA property is also required to achieve this effect). We use this property in Sections 6.2.2 and 6.2.3 to develop compact, Markovian structure to represent lineage. We note from experience that the vast majority of real-world event queries are naturally expressed as unambiguous DFAS; furthermore, most queries that do not have both properties can be easily rewritten into DFA queries with similar semantics and no ambiguity.

6.2.2 Formal Lineage Definition

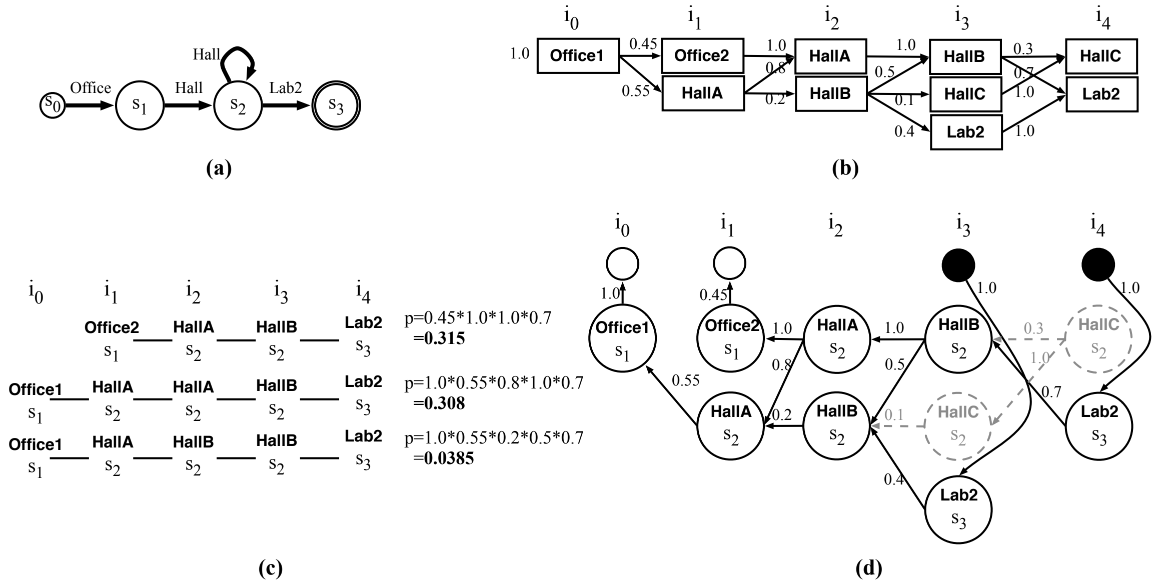
We first define Markovian stream lineage formally in a deterministic setting, and then broaden the definition to cover imprecise (specifically, Markovian) streams. In both settings, recall that we define lineage in terms of a single stream instant i. In a deterministic setting, the input stream is a single sequence of domain elements $(d_0, \dots d_N)$. 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 $l = (i_s, e_{i-n}, \dots, e_i)$ comprising a list of $\langle$ index, domain-element, DFA-state $\rangle$ triples e preceded by a single element i_s indicating the index at which the sequence begins. The index i_s is redundant with the index contained in the element e_{i-n}, but is required for performing projection, which we discuss in Section 6.4. The number n is the length of the lineage sequence; the value of n can be as small as 1 or as large as the length N of the input stream, depending on the input stream and query. Because each sequence element e is a triple containing domain element identifiers *and* DFA states, the lineage sequence l identifies both the segment of the input stream that matches the query at instant i, as well as the DFA state transitions triggered by each element in the segment. Note that both the DFA-query constraint and the unambiguous-query constraint are required for this definition of lineage to hold, even in a deterministic setting. If a query violates either constraint, it is possible for the lineage at instant i to comprise more than one sequence of elements e. We will see shortly that our compact representation of lineage in a non-deterministic setting depends on the property that each *deterministic* input produces at most a single lineage sequence for each stream instant.

This definition of lineage in a deterministic setting generalizes straightforwardly to an imprecise, Markovian setting. In the latter, the lineage $L^M_Q(i)$ 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. The probability $p(l)$ assigned to a lineage sequence l is the sum of the probabilities of all sequences in the input Markovian stream that generate lineage sequence l (by definition, each sequence through the input stream can yield at most one lineage sequence ending at each instant). Each individual lineage sequence l is defined as in the deterministic case. The number of lineage sequences j can range from zero up to the number of unique sequences in the input stream.

The lineage sequences in a set have three important properties: First, they do *not* define a proper probability distribution, because they do not necessarily sum to one. Their sum is the probability that the query is satisfied at instant i in the input stream. Second, they are unique and disjoint. Finally, because the input stream on which lineage is defined is Markovian, the lineage sequences in a set are also Markovian—that is, they can be compactly represented in a Markovian structure, which we discuss shortly. Figure 6.2 shows an example of lineage in an imprecise, Markovian setting. Figures 6.2(a) and (b) show an unambiguous DFA query and a Markovian input stream, respectively, while Figure 6.2(c) shows the set of lineage sequences that represent the lineage of the given query on the given input at instant i_4. For simplicity, Figure 6.2(c) shows the index value of each $\langle$index, domain-element, DFA-state$\rangle$ triple only once per instant, above the set of elements that share the index value. We discuss Figure 6.2(d), a compact representation of this lineage, shortly.

6.2.3 Lineage Graphs

We now introduce a compact representation of lineage, which we call the *lineage graph* or equivalently, *lineage stream*. The lineage graph is a Markovian structure that can efficiently encode the lineage of a given query for *every* instant in a given input stream simultaneously. Given a Markovian stream M and a query Q, the lineage graph of Q on M is a directed acyclic graph (DAG) $\langle V^M_Q, E^M_Q \rangle$, comprising vertices $v \in V^M_Q$ and edges $e \in E^M_Q$. The lineage graph may be disconnected, and is written as simply $\langle V, E \rangle$ when the query and input stream are clear from context. Figure 6.2(d) shows the lineage graph for the query and input stream shown in Figures 6.2(a) and (b), respectively. Dashed vertices and edges in this figure are not a part of the lineage and will be addressed in Section 6.3.



Office
Hall
Hall
Lab2
(a)
Office1
Office2
HallA
HallB
HallC
Lab2
(b)
p=0.45*1.0*1.0*0.7 =0.315
p=1.0*0.55*0.8*1.0*0.7 =0.308
p=1.0*0.55*0.2*0.5*0.7 =0.0385
(c)
(d)

Figure 6.2: (a) Unambiguous DFA query Q. (b) Input Markovian stream M. (c) The set of lineage sequences describing the lineage for instant i_4. (d) The lineage graph describing the lineage of query Q on input stream M, for all instants (only instants i_3 and i_4 have non-empty lineage).

Each path in this graph that starts at a root (black) node and ends at a leaf (white) node represents a unique lineage sequence, as defined previously.

Note that all edges in the lineage graph point backwards in time, in contrast to Markovian stream edges which are directed forward in time. Although the edge direction does not affect the semantics of the lineage graph, backward-pointing edges facilitate backwards-traversal of the graph, which is useful for enumerating all lineage sequences ending at a particular instant.

We define the lineage graph by construction. Recall that each element in a lineage sequence is defined as a triple: $e = \langle i, d, s \rangle$ where i is an instant index, d is a domain element, and s is a DFA state. For each such element appearing anywhere in a set of lineage sequences, a single node $v_d^s(i)$ is added to the lineage graph. Thus the lineage graph contains at most one node per sequence element e (i.e. it is deduplicated). For example, the lineage sequences starting at instant i_0 in Figure 6.2(c) share a first element $e = \langle i_0, \text{Office1}, s_1 \rangle$. In the lineage graph shown in Figure 6.2(d), the node

$v^{s_1}_{Office1}(0)$ represents the first element in both sequences.

An edge is added to the lineage graph connecting node $v^{s'}_{d'}(i+1)$ to node $v^{s}_{d}(i)$ if the two associated elements $e = \langle i, d, s\rangle$ and $e' = \langle i+1, d', s'\rangle$ are consecutive in *any* lineage sequence in the set. The probability assigned to such an edge is the probability of the corresponding edge in the input Markovian stream, determined by the combination of instant index and domain element. For example, the lineage graph edge from node $v^{s_2}_{HallA}(1)$ to $v^{s_1}_{Office1}(0)$ in Figure 6.2(d) has probability 0.55, because 0.55 is the probability of the edge connecting the Office1 element at instant i_0 to the HallA element at instant i_1 in the input stream (Figure 6.2(c)). As with lineage graph nodes, edges are deduplicated: Two lineage graph nodes can be connected by at most a single edge, regardless of the number of times they appear consecutively in the enumerated set lineage sequences. Note that a single Markovian stream edge may be associated with multiple edges in the lineage graph, connecting nodes associated with different DFA states (this does not occur in the scenario of Figure 6.2).

In order to facilitate enumeration of all lineage sequences ending at a given instant i, the lineage graph contains an additional "final" node $v_{final}(i)$ for each instant i at which *any* lineage sequence ends. Edges with probability 1.0 connect this final node to any nodes associated with instant *i and* associated with an accepting (final) DFA state. In Figure 6.2(d), final states are drawn in black and are connected to nodes associated with s_3, the accepting state of the DFA in Figure 6.2(a).

Similarly, the lineage graph contains a "start" node $v_{start}(i)$ for each instant i at which any lineage sequence begins. These start nodes are connected to other 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 6.2(a)). In Figure 6.2(d), start nodes are drawn as small white circles. The probability assigned to an edge out of a node $v^{s}_{d}(i)$ into start node $v_{start}(i)$ is the probability that domain element d is true at instant i (according to the input Markovian stream). Note that this probability is a *marginal* probability and not a conditional one: it does not depend on the value of the previous element in the stream. Start nodes represent the i_s prefix of each lineage sequence as defined in Section 6.2.2, and they are necessary in order to mark the instant at which a query match begins once projection begins (discussed in the next section).

The Markovian nature of the input stream, combined with the DFA and unambiguous-query restrictions, guarantee that any set of lineage sequences can be compactly represented in this manner. By contrast, an arbitrary set of lineage sequences not adhering to the definition in Section 6.2.2 might be represented only by a graph that also encodes additional sequences not present in the original set, rendering the graph useless as an encoding of the original set.

As noted previously, the lineage graph is a compact encoding of the lineage for *all* instants of an input stream, for a given query. Recall that the lineage $L_Q^M(i)$ for instant i is a set of lineage sequences along with a probability for each: the sequences in this set are precisely the sequences that are rooted at $v_{final}(i)$ of the lineage graph, and the probability of each sequence is the product of the probabilities of all lineage graph edges participating in the sequence. Because $\langle V_Q^M, E_Q^M \rangle$ is a DAG, each set of paths is finite and enumerable; however, although each set of lineage sequences is compactly represented in the lineage graph, the number of paths in any set may be exponential in the length of the input stream. Note that, while the lineage graph is defined for an entire query/stream pair, lineage is defined in terms of a particular instant: lineage $L(i)$ is a set of sequences that describes all possible histories resulting in a query match at instant i.

As an example, the lineage graph in Figure 6.2(d) 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 that begin at the final (black) node. 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) = .032. The lineage $L_Q^M(4)$ for instant five includes three paths, two of which end at instant 0 and one of which ends at instant 1. Note that, although the example lineage graph in Figure 6.2(d) contains at most one lineage node per Markovian stream entry, in general the lineage graph may contain q nodes associated with a single domain element at a given instant, where q is the number of states in the query DFA.

We describe algorithms for constructing lineage graphs in Section 6.3.

6.2.4 Characterizing Lineage

A lineage graph is a complex structure. Through our work with real-world lineage, we have identified several characteristics of lineage that have a significant impact on lineage query performance and quality. We outline these characteristics here to provide a vocabulary for discussing lineage. The effects on queries of varying these characteristics are demonstrated in Section 6.5.

Size: We define the size of a lineage graph as the number of nodes. Larger lineage graphs are produced by queries that are more frequently satisfied, and/or are matched by longer subsequences of the input stream. Figure 6.3 shows examples of queries with differently-sized lineage graphs. For the remainder of this chapter, we call query Q_A in Figure 6.3(a) a large-lineage query (7422 nodes); queries Q_B and Q_C in Figure 6.3(b) and (c) medium-lineage queries (1371 and 1885 nodes, respectively); and Q_D in Figure 6.3(d) a small-lineage query (only 175 nodes). We describe the real-world input streams on which these lineages were constructed in Section 6.5. We use the labels 'large', 'medium', and 'small' only to facilitate discussion. As we demonstrate in Section 6.5, the size of a lineage graph affects query performance, with large lineage graphs generally incurring higher total I/O and CPU costs during generation of the lineage graph.

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. Queries Q_A and Q_B in Figures 6.3(a) and (b) yield highly-connected lineage graphs due to the presence of the unselective Hall predicate, while queries Q_C and Q_D in Figure 6.3(c) and (d) yield poorly-connected lineage graphs. Connectivity is important for understanding the cost of projection, with the cost of projection generally increasing with connectivity.

Skew: In contrast to size and connectivity, 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 very low. Consider an event query Q that is satisfied at instant i of Markovian stream M with probability $p(Q[M]@i)$. The sum of the probabilities of all lineage sequences ending at instant i equals $p(Q[M]@i)$. The skew of this particular query match is the sum of the probabilities of the top k lineage sequences, divided by $p(Q[M]@i)$. Thus, skew values fall in the range $(0.0, 1.0]$. A value 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 lies in the top-k set. High skew values reflect a poor diversity of lineage sequences, usually because one or two sequences have significantly higher probabilities than the rest. Viewed from a different perspective, skew is a measure of the utility of a particular value of k for a given stream and query: high skew values indicate that applications can indeed get meaningful information by examining only the k most likely likely lineage paths, possibly saving them the cost of enumerating a larger number of paths.

We define skew formally over an entire lineage graph: $skew(\langle V_Q^M, E_Q^M \rangle, k) = \frac{\sum_{i=0}^{N}(\sum_{j=1}^{k}(K(i,j)))}{\sum_{i=0}^{N}(p(Q[M]@i))}$ where $K(i,j)$ refers to the probability value of the j^{th} most likely lineage sequence resulting in a query match at stream instant i, and N is the length of the input Markovian stream. As an example, consider the three lineage sequences in Figure 6.2(c), which together describe the lineage for the query in Figure 6.2(a) at instant i_4 of the input stream in Figure 6.2(b). For k=1, the skew at instant i_4 is $\frac{0.315}{0.315 + 0.308 + 0.0385} = 0.476$. For k=2, the skew increases to $\frac{0.315 + 0.308}{0.315 + 0.308 + 0.0385} = 0.942$.

In the context of lineage, skew is a positive feature. High skew values indicate to applications that the value of k is sufficient, and can possibly even be reduced to improve performance; low (poor) skew values, on the other hand, indicate to applications that the value of k should be increased. As we will demonstrate in Section 6.5, skew is a useful tool for measuring of the effect of projection on lineage graphs.

6.3 Constructing Lineage

In this section, we first briefly describe a straightforward but intractable algorithm for constructing Markovian stream event query lineage. We then introduce a novel, tractable algorithm with running time linear in the length of the input stream.

6.3.1 Naïve Construction Algorithms

Intuition suggests that a natural way to construct lineage for DFA queries on Markovian streams is to simply adapt the standard Markovian stream DFA processing algorithm for this purpose. The standard (Boolean) algorithm is described in Section 2.2.2. At a high level, it is a single-pass algorithm that iteratively updates a two-dimensional probability distribution Q, using the correlations linking each consecutive stream instant. Each row of Q corresponds to a *set* of DFA states, and each column corresponds to a domain element $d \in D$; entry $Q(i,j)$ is a number between 0 and 1, indicating the probability that the query DFA is in the i^{th} set of states, and that the "true" value of the last stream instant is the j^{th} domain element (Figure 2.4). For reasons of space we omit details of the algorithm used to update Q; these details can be found in earlier papers on Lahar [94,73], and also in Chapter 2.

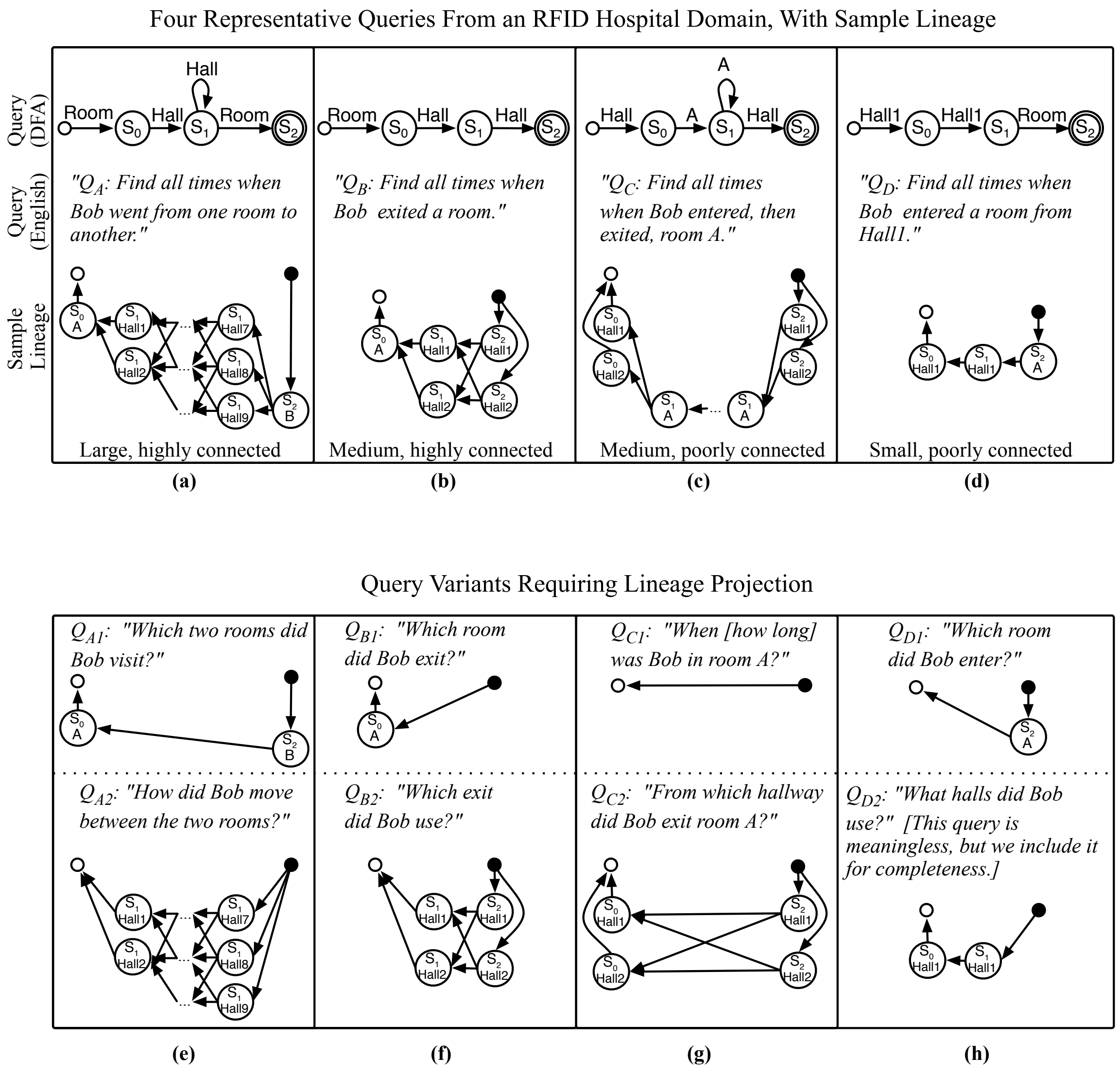

Four Representative Queries From an RFID Hospital Domain, With Sample Lineage
Query (DFA)
Query (English)
Sample Lineage
Room
Hall
Hall
Room
"Q_A: Find all times when Bob went from one room to another."
"Q_B: Find all times when Bob exited a room."
"Q_C: Find all times when Bob entered, then exited, room A."
"Q_D: Find all times when Bob entered a room from Hall1."
Large, highly connected
Medium, highly connected
Medium, poorly connected
Small, poorly connected
(a)
(b)
(c)
(d)
Query Variants Requiring Lineage Projection
Q_{A1}: "Which two rooms did Bob visit?"
Q_{B1}: "Which room did Bob exit?"
Q_{C1}: "When [how long] was Bob in room A?"
Q_{D1}: "Which room did Bob enter?"
Q_{A2}: "How did Bob move between the two rooms?"
Q_{B2}: "Which exit did Bob use?"
Q_{C2}: "From which hallway did Bob exit room A?"
Q_{D2}: "What halls did Bob use?" [This query is meaningless, but we include it for completeness.]
(e)
(f)
(g)
(h)

Figure 6.3: (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 that require 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.

To compute lineage, the natural extension of this technique is to augment the distribution Q with a third dimension, which contains a value for every possible lineage sequence. This allows the processing algorithm to track in memory not only query match probabilities as before, but also to construct and track the lineage sequences associated with each query match, including projections. This technique of extending the distribution Q with a third dimension to track additional state has previously been used to compute aggregations over regular query matches [73], as described in Section 2.3.1. The details of the extension are similar in this case.

The clear problem with this approach is that, for many queries, the number of possible lineage sequences grows exponentially with the length N of the input stream, quickly rendering Q too large to process. Indeed, the standard Boolean event query processing algorithm is efficient precisely because, by exploiting the Markov property of its input streams, it is able to ignore exponentially-sized history; it is thus not surprising that the algorithm becomes intractable once it must track all possible histories (lineage sequences). However, for queries whose lineage sequences are both small and bounded, this approach can work very well. Such queries include short, fixed-length queries (i.e. queries whose DFAs include no loops and only a small number of states).

A second naïve approach to lineage construction is to enumerate each deterministic sequence represented in the input Markovian stream, construct lineage separately for each, and then take the union of the resulting lineage elements. In this case, the probability of a particular lineage sequence is the sum of the probabilities of all deterministic paths that generate the sequence. This approach is clearly intractable for any query, because the number of deterministic sequences that must be enumerated is $O(D^N)$, which is exponential in the length N of the stream.

6.3.2 Linear-Time Lineage Construction

The lineage graph defined in Section 6.2.3 exhibits the Markov property. By exploiting the Markov properties of both the input stream and lineage graph, Lahar is able to generate the lineage graph $\langle V_Q^M, E_Q^M \rangle$ for input Markovian stream M and query Q using a linear-time algorithm that performs two passes over the input stream. This algorithm is described in this section. The number of paths through this lineage graph that contribute to the lineage $L_Q^M(i)$ of a query match at a particular instant i may be exponential in the length of the input stream: Even though the lineage graph compactly represents these paths, enumerating them can be costly. For this reason, Lahar supports enumeration only of the top k paths associated with each $L_Q^M(i)$. The algorithm for converting a lineage graph $\langle V_Q^M, E_Q^M \rangle$ into a top-k version from which the top k paths can be easily enumerated is described later in this section.

Lineage Graph Construction

Lahar's 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. 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. These additional edges are identified and pruned in the second phase of lineage graph construction, which begins at the last instant in the graph and proceeds backwards until the first graph instant is reached.

In the remainder of this section we describe in more detail the two phases of lineage graph construction.

Lineage Construction: Forward Pass. At a high level, the forward pass of the lineage construction algorithm processes the input stream from beginning to end. As it processes each instant i, it generates nodes and edges in the lineage graph. For each instant, Lahar first creates nodes associated with lineage sequences that begin at instant i, and then it creates nodes and edges that extend existing lineage sequences (including those beginning at instant i). We review these two construction processes in turn; they are also outlined in Algorithm 4.

Upon processing a new instant i of the input stream, Lahar first creates nodes (start nodes and others) associated with lineage sequences beginning at instant i (Lines 3-9 of Algorithm 4). First, Lahar adds a start node $v_{start}(i)$ to the lineage graph. It then creates a lineage graph node $v_d^s(i)$ for each pair (d, s), where d is a domain element with marginal probability $p_m > 0$ at instant i, and s is a DFA state reachable from the start state via a transition satisfied by d. "Start" edges $v_{start}(i) \leftarrow v_d^s(i)$, with probabilities p_m, are added connecting each new node with the start node of the same instant. As an example, in Figure 6.2(d), the nodes created in this step are start nodes $v_{start}(0)$ and $v_{start}(1)$; and $v^{s_1}_{Office1}(0)$ and $v^{s_1}_{Office2}(1)$, which are connected to their appropriate start nodes via edges having probability 1.0 and 0.45, respectively.

As the second step of processing a new instant i, Lahar adds edges and nodes that continue lineage sequences beginning at instant i or earlier (Lines 10-20 of Algorithm 4). New nodes are added at instant $i + 1$ of the lineage graph, and new edges are added connecting nodes at instants i and $i + 1$, as follows: For each lineage graph node v^s_d at instant i; and for each edge $d \rightarrow d'$ (with probability p_e) from instant i to $i + 1$ in the input Markovian stream; and for each DFA transition $s \rightarrow s'$ satisfied by domain element d', Lahar adds a "middle" edge $v^s_d(i) \leftarrow v^{s'}_{d'}(i + 1)$ with probability p_e to the lineage graph. If the node $v^{s'}_{d'}(i + 1)$ does not already exist, it is added to the graph to facilitate addition of the new edge. If a Markovian stream edge $d \rightarrow d'$ does not satisfy any transitions out of DFA state s, then Lahar does not add an edge t the lineage graph, and simply continues examining other input stream edges. After all appropriate lineage graph edges from instant i have been added in this manner, Lahar creates a final node $v_{final}(i + 1)$ and "final" edges $v^{s'}_{d'}(i+1) \leftarrow v_{final}(i+1)$ connecting the final node $v_{final}(i+1)$ to each node $v^{s'}_{d'}(i+1)$ in which s' is a final (accepting) state of the DFA. This final step is represented in Lines 18-20 of Algorithm 4.

As an example of the second processing step, in which existing lineage sequences are extended, consider the node $v^{s_1}_{Office1}(0)$, where $d = Office1$ and $s = s_1$, in Figure 6.2(d). Two corresponding edges $d \rightarrow d'$ between instants 0 and 1 exist in the input stream in Figure 6.2(b): the edges $Office1 \rightarrow Office2$ ($p_e = 0.45$) and $Office1 \rightarrow HallA$ ($p_e = 0.55$). Consider first the edge $Office1 \rightarrow Office2$ (e.g. assign $d' = Office2$). The DFA in Figure 6.2(a) contains no transition $s_1 \rightarrow s'$ whose predicate is satisfied by domain element Office2, so in this case Lahar adds nothing to the lineage graph. Now consider the edge $Office1 \rightarrow HallA$ (e.g. assign $d' = HallA$). The DFA transition $s_1 \rightarrow s_2$ with predicate 'Hall' is satisfied by the $HallA$ element, so Lahar adds the edge $v^{s_1}_{Office1}(0) \leftarrow v^{s_2}_{HallA}(1)$ to the lineage graph, with probability $p_e = 0.55$.

These two steps are repeated for each instant i in turn, until the end of the input stream is reached, signaling the end of the forward pass of the lineage construction algorithm.

Lineage Construction: Backward Pass. The second phase of lineage graph construction is a backward pass over the pre-lineage graph constructed in the first phase. During this backward pass, "dead-end" branches of the lineage graph are identified and removed. These dead-end branches are the result of partial query matches, and are illustrated in Figure 6.2(d) as dashed elements.

Pruning of unnecessary pre-lineage graph elements is straightforward. Beginning at instant $N - 1$ and moving toward instant 0, Lahar performs the following steps for the pre-lineage graph nodes associated with each instant i: 1) The final node $v_{final}(i)$ is marked as valid; 2) Any node $v(i)$ associated with instant i and reachable via a single edge *from a valid node* is itself marked as valid; and finally, 3) Any node associated with instant $i + 1$ that is not marked as valid is removed. By performing these steps for all instants, the pre-lineage graph is transformed into the lineage graph, with no unnecessary elements.

The backward, pruning phase of lineage construction requires only the pre-lineage graph as input; it does not read or write the input Markovian stream. Furthermore, this backward pass is memory-efficient because it must keep only two instants' worth of the pre-lineage graph in memory at any given time.

Top-K Lineage

Once the lineage graph is constructed, Lahar can provide lineage paths to applications by enumerating the paths rooted at the final node of the instant for which lineage is requested. This enumeration can be performed as a simple depth-first traversal of the lineage graph, but there is no way to enumerate lineage paths in ranked order using this approach. A ranked enumeration algorithm is required to allow Lahar to return meaningful results to applications without enumerating all lineage paths, which as we have seen can number exponentially in the length of the input stream.

Lahar supports ranked lineage path enumeration using a top-k approach analogous to the top-k queries supported in other database systems [29, 113, 93]. In this case, applications specify a value for k, and, for each instant in which lineage is requested, Lahar returns only the k lineage paths with the highest probabilities, instead of enumerating all relevant paths.

In order to support top-k lineage queries, Lahar converts the lineage graph into a top-k lineage graph, from which the top k paths for each query match can be easily enumerated. This conversion process requires a single pass in the forward direction over the lineage graph, and is separate from any additional accesses required to later enumerate the top k paths.

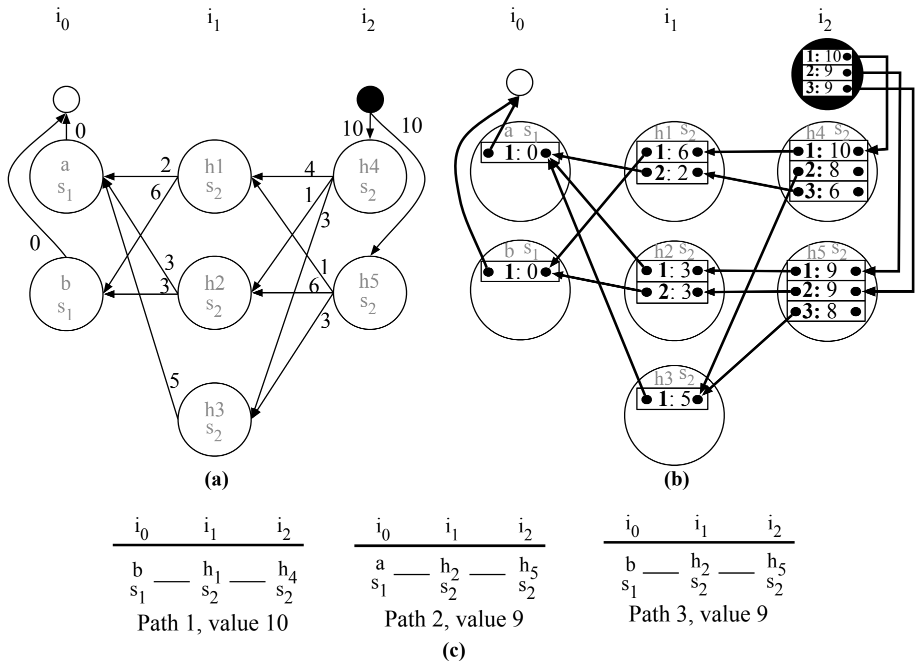


Figure 6.4: (a) Sample lineage graph snippet over locations $\{a, b, h1, \ldots h4\}$ and DFA states $\{s_0, s_1, s_2\}$. For clarity, the log-probability values of graph edges have been replaced with symbolic integer values; in either representation, higher values are ranked before lower ones. (b) Graph in (a) after top-k conversion, performed for k=3. (c) Output returned to applications, enumerating the top-3 lineage sequences for instant i_2.

Lahar's top-k conversion algorithm is a 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) [41]. This algorithm begins at the first (earliest/lowest) index in the lineage graph and moves to the last, converting each node at the given index into a top-k lineage graph node in turn. The conversion replaces each node's outgoing edges with a set of top-k edges.

A top-k edge is a tuple $\langle rank, dNode, dRank, p\rangle$, where *rank* is the rank of the edge in its set (i.e. a top-k edge with rank 1 marks the most likely path out of a node); *dNode* is the identity of the previous node in the path; *dRank* is the rank of the edge to be followed *out* of *dNode*, and *p* is the probability of the path. Note that a single top-k edge *e* out of a node *n* uniquely specifies a path beginning at *n* and ending at (potentially) the beginning of the stream, and that the probability *p* associated with e is the total probability of this entire path. Because path probabilities are computed as the product of the probabilities of each edge along the path, Lahar uses logarithmic probabilities in its top-k algorithm, which allows multiplications to be replaced with additions and avoids numeric underflow [103].

Figures 6.4(a) and (b) show a lineage graph snippet, and the top-k converted snippet for k=3, respectively. In Figure 6.4(b), the *rank* and p values of each edge are shown explicitly in boxes, as $rank : p$, while the *dNode* and *dRank* of each edge are indicated by arrows. For example, the most likely path out of node h4 is indicated by edge $\langle rank = 1, dNode = (h1, s2), dRank = 1, p = 10\rangle$ while the third-most likely path out of the same node is indicated by the edge $\langle rank = 3, dNode = (h1, s2), dRank = 2, p = 6\rangle$. Note that both of these edges share a destination node value of $h1$, making the *dRank* value necessary.

Once a lineage graph is converted into a top-k lineage graph, Lahar can enumerate the top k lineage paths matching a particular instant i simply by following the top-k edges out of the $v_{final}(i)$ node of the converted graph. The top-k conversion and path enumeration process can be applied to any lineage graph, whether the graph has undergone a projection transformation (next section) or not.

6.4 Lineage Projection

Although some queries can be answered by enumerating ranked lineage paths directly from a lineage graph, many queries require that parts of the graph be *projected* before such enumeration is useful. Figures 6.3(e-h) list examples of such queries, along with lineage graphs projected as necessary for each query (the lineage graphs prior to projection are shown in Figures 6.3(a-d), respectively). Consider query Q_{A1} (Figure 6.3(e)), in which an application is interested not only in when Bob moved between rooms, but also in knowing the identities of the rooms themselves. The application is *not* interested in knowing what path Bob took through the hallways, so lineage graph elements corresponding to hallways can be projected away. A schematic diagram of the resulting lineage graph after this projection is shown in Figure 6.3(e), while the original lineage graph, without projection, appears in Figure 6.3(a). In Figure 6.3, queries Q_{A1}, Q_{B1}, Q_{D1}, Q_{B2} and Q_{C2} are *which*-style projection queries whose answers identify individual domain elements (here, locations); query Q_{C1} is a *when*-style query whose answer is a time interval; and queries Q_{A2} and Q_{D2} are *how*-style queries whose answers are paths or subsets of paths through the lineage graph.

We ran query Q_{A1} on a real-world Markovian stream derived from RFID data, and compared the quality of results obtained using projection, and using no projection (the latter is equivalent to allowing an application to perform projection as a post-processing step on the k most likely paths returned by Lahar). When Lahar performed projection of hallway elements *before* enumerating the top ten lineage paths, these ten paths covered 79% of the lineage (i.e. skew was 0.79 for k=10). In contrast, the top ten paths of the unprojected lineage covered only 1.17E-6% of the lineage (i.e. the unprojected lineage graph has very low skew), which is not enough coverage to produce meaningful query results even if the 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 (although recall that this answer only covers 1.17E-8% of the lineage). The top ten paths computed on the projected graph, which together cover 79% of the lineage, indicate that Bob started in room A with probability of only 0.43; with probability 0.57, he began in an adjacent room, room B. Thus an application interested in query Q_{A1} cannot produce accurate results without projecting the lineage graph, regardless of the sophistication of its post-processing.

In this section, we outline the semantics of lineage projection, and our algorithms for performing this projection in practice. In this dissertation, we support projection at the level of individual lineage graph edges. In the work of this dissertation, applications specify one or more Boolean projection tests that Lahar applies locally to each edge in the lineage graph; those edges passing the test are retained, while the rest are projected away. This is an extension to the work of Kimelfeld and Ré, whose Markovian stream lineage analysis applies only to projection of prefixes and suffixes of an event query. Of the queries in Figure 6.3(e-h), some (Q_{B1}, Q_{C1}, Q_{D1}, and Q_{A2}) are handled in the framework of Kimelfeld and Ré, while the remaining projection queries Q_{A1}, Q_{B2}, Q_{C2}, and Q_{D2} can be answered only using the techniques in this dissertation.

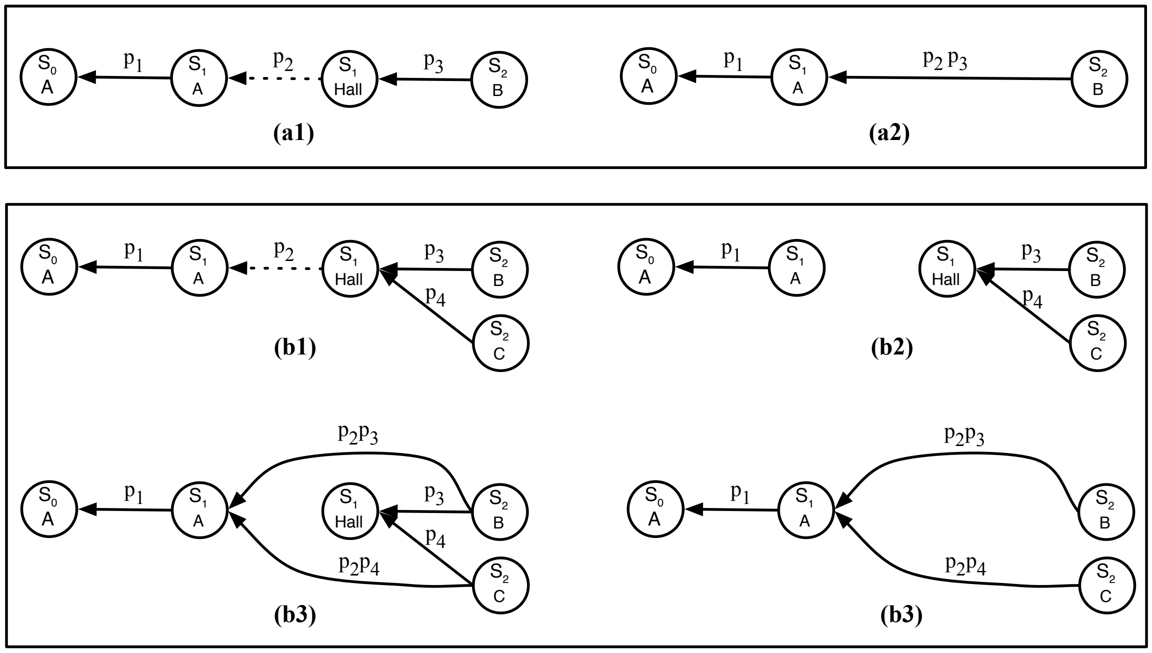
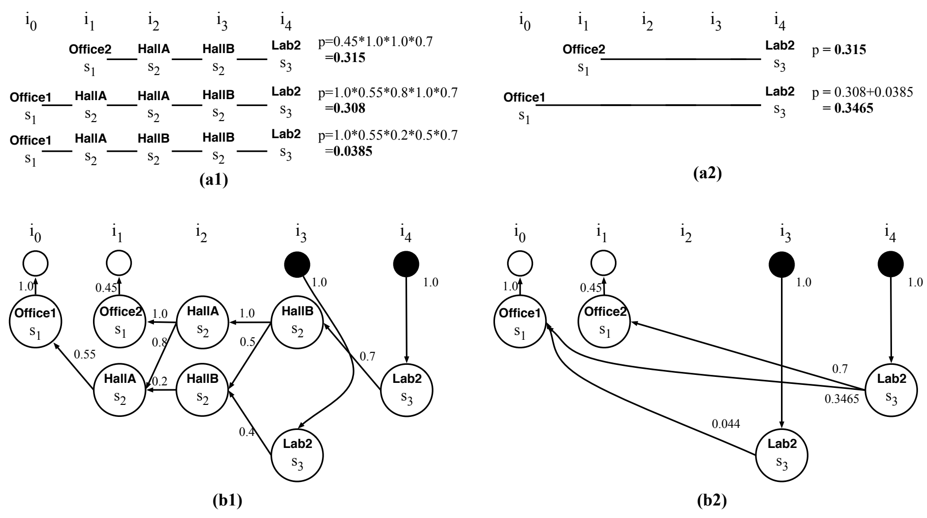


Figure 6.5: (a1) A lineage sequence before projection of the element (i_2, s_1,Hall), marked by the dotted edge. (a2) The sequence in (a1) after projection. (b1-b4): Transformation of a lineage graph to project out the dotted edge. The three stages of projection are described in Section 6.4.2.

6.4.1 Formal Definition

We define projection on lineage $L_Q^M(i)$ as a transformation that removes a given element e' 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 result in two or more identical sequences, the sum of the probabilities of these identical sequences is assigned to a single, deduplicated result. Figures 6.5(a1) and (a2) show a lineage sequence before and after projection on the pair ($[i_1, s_1$, A$]$, $[i_2, s_1$, Hall$]$), respectively. We return to Figure 6.5(b) in Section 6.4.2.



i0 i1 i2 i3 i4
Office2 s1 HallA s2 HallB s2 Lab2 s3 p=0.45*1.0*1.0*0.7 =0.315
Office1 s1 HallA s2 HallA s2 HallB s2 Lab2 s3 p=1.0*0.55*0.8*1.0*0.7 =0.308
Office1 s1 HallA s2 HallB s2 HallB s2 Lab2 s3 p=1.0*0.55*0.2*0.5*0.7 =0.0385
(a1)
Office2 s1 Lab2 s3 p = 0.315
Office1 s1 Lab2 s3 p = 0.308+0.0385 = 0.3465
(a2)
1.0 0.45 1.0 1.0 1.0 0.8 0.5 0.55 0.2 0.7 0.4
(b1)
0.7 0.3465 0.044
(b2)

Figure 6.6: (a1) Sample lineage (Lahar output) for instant i_4 from the example scenario in Figure 6.2. (a2) Lineage from (a1) in which hallway elements have been projected away. (b1), (b2): Lineage graphs encoding the lineage shown in (a1) and (a2), respectively.

The elements e' and e used in the definition of projection include instant indexes, which makes projection transformations specific to given instants in the lineage. After such a projection, the element e' may still be present in some lineage sequences (specifically, those sequences in which e' is preceded by an element other than e). We define projection in this way in order to guarantee that projection preserves the Markovian property of the set of lineage sequences: that is, even after arbitrarily-chosen projections of this type, the resulting set of lineage sequences can be compactly represented using a lineage graph. Intuitively, the projection of a pair (e, e') can be applied directly to the lineage graph representation of a lineage sequence set, simply by removing the lineage graph edge linking elements e and e' (Figure 6.6).

This definition of projection requires that applications make projection decisions about pairs of adjacent elements (e, e'). This definition allows for projection of lineage elements associated with a particular DFA edge (identified by the DFA states of e and e). This definition of projection also allows for projection of lineage elements associated with individual domain elements or DFA states. Such projection decisions can be made simply by evaluating the domain element or DFA state associated with element e' without regard to the values of e. For example, the projected lineage graph for query Q_{A1} in Figure 6.3(e) is obtained from the lineage graph in Figure 6.3(a) by performing projection on all pairs (e, e') in which e' is associated with DFA state s_1, or equivalently, on all pairs in which e' is associated with a domain element satisfying the Hall predicate.

The characteristics of a lineage graph derive from the graph's structure, and may therefore change when projections are applied that alter the graph. Projection can potentially reduce, but may never increase, the number of nodes (i.e. the size) of a lineage graph. Projection can increase the skew (as when multiple paths in the original lineage graph are projected onto the same result); however, projection may either increase or decrease the connectivity of a graph, depending on the graph and the projections applied. We discuss the effects of projection on lineage graphs further in Section 6.4.

As mentioned previously, the start element i_s cannot be projected out of any lineage sequence. Without these start elements, two lineage sequences starting at different instants can potentially project onto one another. Kimelfeld & Ré proved that projection is intractable in this case [68]; their solution, the "indexed s-projector", is able to produce lineage with start indexes equivalent to the start indexes provided by the i_s element of the lineage definition in this thesis (indexed s-projectors, however, handle projection only of prefixes and suffixes of lineage, while this chapter addresses projection of arbitrary lineage graph edges).

6.4.2 Applying Projection

In the previous section, we defined projection formally as a transformation on a set of lineage sequences. In this section we note that this transformation can conveniently be applied directly on the lineage graph representation of this set, eliminating the need to enumerate lineage sequences during projection. Specifically, 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 of the graph; however, care must be taken to maintain connectivity of the graph during this process.

Concretely, applying the projection transform (e, e') to a set of lineage sequences can be achieved by: 1) removing the lineage graph edge $(e \leftarrow e')$; 2) adding a new edge $e \leftarrow e''$ to the lineage graph for every existing edge $e' \leftarrow e''$, and 3) removing from the graph all edges $e' \leftarrow e''$ *if* e' has no outgoing edges. This process is depicted in Figure 6.5(b), where (b1) shows a lineage graph, and (b2)-(b4) show the graph after each step of projection (the dotted edge is the one being projected away). Together, steps one and two project the element e' out of the graph while maintaining connectivity. Step 3 removes from the graph edges that are no longer necessary because they reflect information (connectivity) that is now represented by the new edges added in step two. When a new edge ($e \leftarrow e''$) is added to the graph (step two), its probability is the product of the probabilities of the two edges ($e \leftarrow e'$) and ($e' \leftarrow e''$). When such a new edge connects two nodes that are already connected, the probability of the existing edge is simply augmented by the probability of the new edge, to avoid duplicate edges.

An important feature of this projection algorithm is that it alters the lineage graph so that it is no longer Markovian after projection. Specifically, the edges that are added to the graph in step two of the projection process can span an arbitrary number of instants. Thus, not only is the projected graph not guaranteed Markovian, but its vertices have degree $O(|V|^2)$ instead of the $O(|V|W)$ degree of nodes in an unprojected lineage graph, where W is the maximum number of vertices associated with a single instant in the lineage graph. The $O(|V|^2)$ nature of the edges in the projected lineage graph means that the algorithm to produce a projected graph has running time quadratic in $|V|$ (note that edges added during projection may later be projected away themselves).

Conceptually, the set of lineage graph edges that must be projected away to satisfy a particular query can be processed in any order (of course, edges added by the projection algorithm obviously cannot be projected away until after they exist). Both the correctness and the complexity of a sequence of projection transforms is independent of the ordering in which they are applied. 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. In this scheme, all projections of edges starting at instant i must be completed before projections of edges that start at instant $i + 1$ (recall that edges in the lineage graph point backwards in time). Because the projection process only adds edges that start at the instant *after* the edge being projected, projection 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 in step two/three. All experiments in Section 6.5 use this sliding-window, single-pass approach to projection.

Recall from Section 6.3 that construction of a lineage graph occurs in two phases: the first phase constructs a superset of the graph, while the second phase prunes away dead-end graph branches corresponding to partial query matches. Although it is conceptually cleaner to imagine projection applied to a completed lineage graph, in practice projection can 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 I/O costs for reading/writing of elements that are eventually projected out. Of course, the combined construction-plus-projection approach followed by pruning is not always superior to the straightforward construction-pruning-projection approach; the latter is better when significant pruning occurs, since time is otherwise wasted projecting elements that will eventually be pruned away. We explore the tradeoffs of these two approaches to projection further in Section 6.5.

6.5 Experimental Evaluation

In this Section, we evaluate the performance of Lahar's algorithms for constructing lineage graphs, performing projection on them, and enumerating the top k lineage paths from them. All experiments were performed on the Lahar implementation described in Section 3, on a 2.0GHz Linux machine with 16GB of RAM.

6.5.1 Setup

We evaluate the performance of Markovian lineage queries on two real-world data sets, both of which are described in detail in Section 3.3.

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 and is 8.2MB in size. On average, the trace reflects 10.6 locations per instant that the person was in with some non-zero probability. During the 12 minutes represented in this trace, the individual walked through the hallways of an office building and visited several different offices for roughly one minute each; the example queries we evaluate in this section, also shown in Figure 6.3 are patterns specified over these offices and hallways.

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 in this section comprises 915 instants, 1.3MB, and lists, on average, 4.9 words per instant with non-zero probability. The uncertainty in the audio stream is less than the uncertainty in the RFID stream, and we will demonstrate that the skew is correspondingly different (larger/better on audio queries). The three audio queries we use to evaluate performance are each three-word phrases, including the phrase, “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 present our analysis in terms of these queries, we performed these experiments on a wider range of queries and using a broader set of input streams, and verified that the results are within the ranges highlighted here. We thus omit these results for clarity of presentation.

6.5.2 Constructing & Enumerating Lineage

We begin by studying Lahar’s performance for generating and enumerating lineage without projection. Recall that, in this case, Lahar performs three sequential passes over the lineage graph: a forward pass to generate the pre-lineage graph, a backward pass to prune the graph, and a final forward pass to compute the top k set for each query match. The total time required for each of these passes is shown in Figure 6.7 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. Note that the four queries shown here from the RFID domain are precisely queries Q_A, Q_B, Q_C, and Q_D from Figure 6.3(a-d), respectively.

Total Cost of Lineage Is Moderate: In both domains, the cost of *each* lineage-related pass is less than twice the cost of the single-pass Boolean processing algorithm, and in many cases each lineage-related pass requires only a fraction of the Boolean processing time. In any case, all queries completed in under two seconds. For a worst-case overhead of roughly 4x, Lahar is able to compute the top ten full-length lineage sequences associated with every query match in the input stream. This

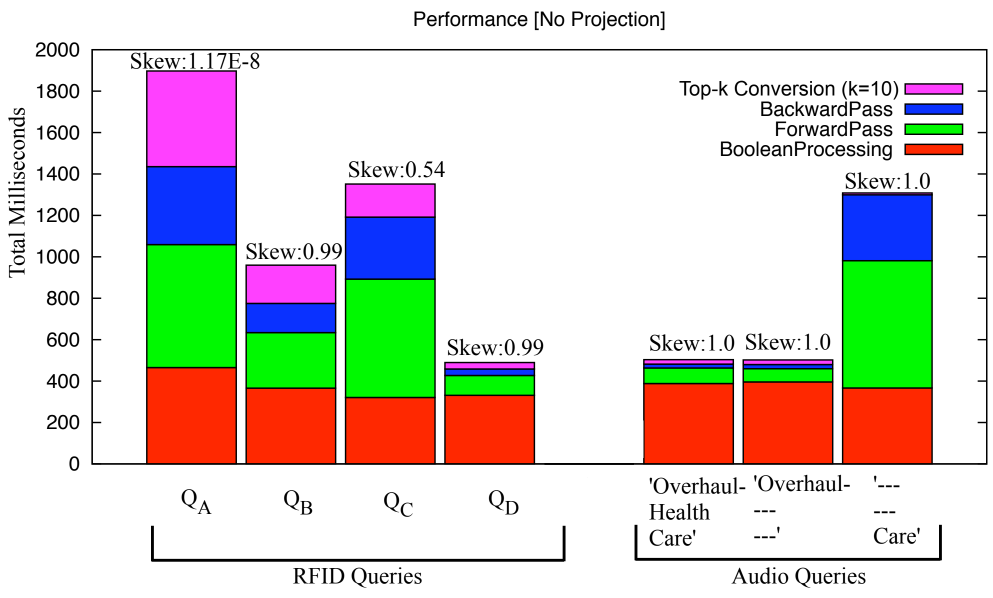


Figure 6.7: Performance of constructing the lineage graph and computing the top k paths for each match, shown for four real-world queries in an RFID-derived stream (left), and three real-world queries in an audio domain (right). Skew values and lineage graph sizes for these queries, computed using k=10, are shown in Figure 6.8.

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 sizes are shown in Figure 6.8(b)). 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 roughly 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

RFID Data

	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

Audio Data

"OHC"	"O**"	"**C"
1.0	0.88	1.0
1.0	1.0	1.0
1.0	1.0	1.0
1.0	1.0	1.0

Skew

(a)

	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/stream size (number of nodes)

"OHC"	"O**"	"**C"
20	24	9920
20	24	20
4526	4526	4526

Graph/stream size (number of nodes)

(b)

Figure 6.8: (a) Skew values and (b) lineage graph statistics, for queries Q_A through Q_D. Performance of these queries is shown in Figure 6.7.

size of the *final* lineage is small, as can be seen in Figure 6.7 for queries Q_C and the "* * care" audio query. on which many nodes are generated in the forward pass only to be eliminated during pruning (actual graph sizes are shown in Figure 6.8(b)).

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 being the dominant factor in performance. Figures 6.7 and 6.8 clearly show the natural correlation between the size of the lineage and the cost of top-k conversion: 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 the value of 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 6.7(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 5x for a 10x 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 (for k=10) are shown above the performance bars in Figure 6.7, and skew values for varying k are shown in Figure 6.8(a). 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. Indeed, skew is always 1.0 on the "Overhaul Health Care" query, on any input data, because only one word sequence can ever match this query. In the RFID domain, the queries with high skew (Q_B and Q_D) have lineage that contains more than 10 paths, but in this domain the Markovian stream is itself skewed such that only a few lineage paths have high probability. Indeed, we determined experimentally that the value of k can be reduced to 4 for Q_B and 1 for Q_D while still maintaining a skew value over 0.99.

RFID queries Q_A and Q_C, however, which include loop predicates specifying arbitrarily-long intervals of time in which Bob was in a hallway or a room, 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. Although the Markovian stream itself is skewed such that only 1-3 locations have any significant probability in each instant, the high connectivity of the lineage graph for Q_A means that each of these locations, at each stream instant, participates in many lineage paths, leading to low skew. On the other hand, the low skew of Q_C is due almost entirely to uncertainty about the starting time of each query match.

Overall, five of the seven representative queries shown in Figure 6.7 have skew of at least 0.99 for k=10. Independently of the cause of this high skew, its common appearance on real-world queries and data indicates that projection/top-k optimizations that leverage high skew will be useful in practice. We demonstrate this point further in the next section.

Projection must be performed before top-k enumeration, and thus must be performed by Lahar: Consider the skew of query Q_A in Figure 6.8(a): 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 in Lahar (and thus before top-k enumeration), the skew of query Q_{A1} rises to 0.79 for k=10.

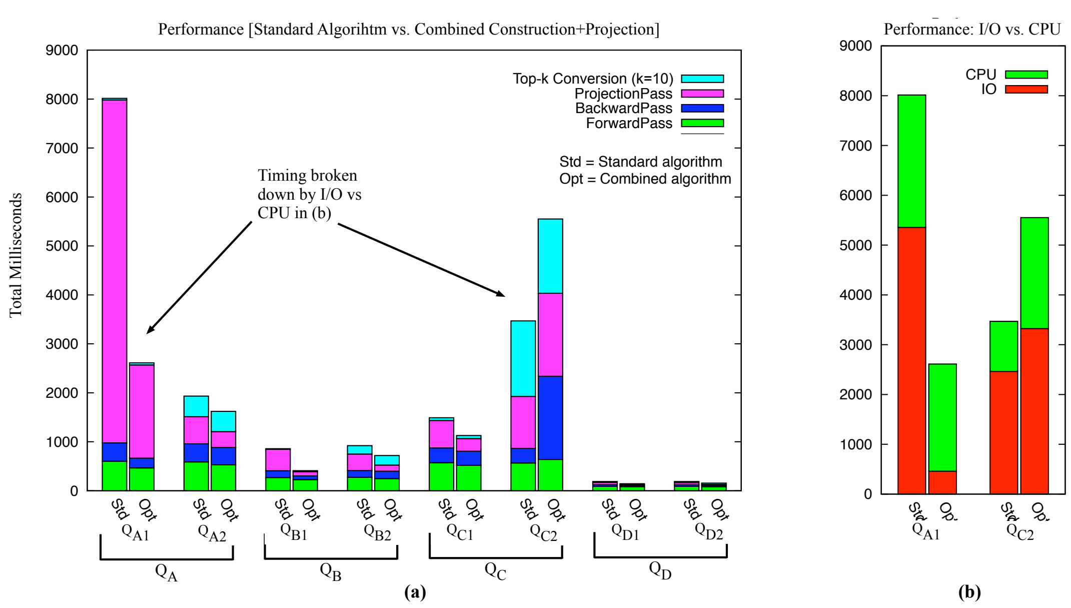


Figure 6.9: (a) Performance of lineage queries including projection on RFID data. The left bar of each pair shows performance of an algorithm using a separate pass over the data for projection; the right bar of each pair shows performance of an algorithm that combines lineage construction and projection in a single data pass. (b) Performance of queries Q_{A1} and Q_{C2} broken down into I/O and CPU components

6.5.3 Performance: Projecting Lineage

Figure 6.9(a) shows the performance of eight representative lineage queries that include projection. Recall that each query is a variation of one of the four queries whose performance is shown in Figure 6.7. The left bar of each pair of bars in Figure 6.9(a) 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 6.9(a) shows that the overhead of performing projection is manageable in practice: of the 8 representative queries, all completed in under 8 seconds, and all finished

	Q_A		Q_B		Q_C		Q_D	
	Q_{A1}	Q_{A2}	Q_{B1}	Q_{B2}	Q_{C1}	Q_{C2}	Q_{D1}	Q_{D2}
Skew for K=1	0.26	1.3E-9	0.99	0.63	0.16	0.16	0.99	0.99
Skew for K=3	0.49	3.8E-9	0.99	0.98	0.28	0.28	0.99	0.99
Skew for K=5	0.63	6.2E-9	0.99	0.99	0.36	0.36	0.99	0.99
Skew for K=10	0.79	1.2E-8	0.99	0.99	0.54	0.54	0.99	0.99

Figure 6.10: Skew values for lineage projection queries for varying k.

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 cost of projection might 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 (from I/O cost reduction): On seven out of eight of the example queries whose performance is profiled in Figure 6.9(a), the combined construction+projection algorithm yields better performance than the algorithm that performs each using separate passes over the data. In the case of query Q_{A1} the difference in performance is dramatic, reducing query completion time from 8 seconds down to 2.5. Lahar's performance on query Q_{A1} is also shown in the first pair of bars in Figure 6.9(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 the lineage graph before projection is very large, and the graph after projection is small. By performing projection as the graph is constructed, in memory, Lahar 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 algorithm that processes the lineage graph in multiple passes. Two factors can cause the combined algorithm to perform poorly, and both contribute to its poor performance on query Q_{C2}.

The first factor is visible in Figure 6.9(a), and is 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 roughly the same size in both algorithms (9168 vs. 9059 nodes), but the average degree of each node is 55 in the projected graph constructed using combined construction+projection, and only 2.15 in the unprojected graph to which backward pruning is applied *before* projection in the multi-pass algorithm. This 25x increase in connectivity greatly increases the CPU costs associated with the backward pruning pass of the algorithm, as can be seen in Figure 6.9(b).

A secondary reason for the poorer performance of the construction+projection algorithm on query Q_{C2} is that projection in this query removes relatively few nodes (less than two percent), so the optimized version is not able to improve performance by reducing I/O costs as it does in the other queries.

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.

6.6 Conclusions

In this chapter, we have outlined a set of Markovian stream lineage queries that provide applications with detailed information about *how* a query was matched in an input stream. We formally defined a lineage graph to capture this information, and provided single-pass algorithms for constructing the lineage graph, performing projection on it, and enumerating the top-k sequences for any particular query match. We evaluated the performance of these algorithms on real-world data sets from two domains to demonstrate that lineage queries can be answered efficiently in practice.

The work in this chapter leaves open some important questions. First, question of when to apply projection before pruning and when to apply it afterward is left to future work. One approach to this problem might be to collect a small sample of statistics (size, connectivity, number of dead-end nodes, *etc.*.) at the beginning of query processing, and to use these to determine how to approach processing of the remainder of the query/stream. Additional open problems include development of a language for specifying lineage queries (and projections in particular), and optimizations for specific types of lineage query such as duration queries interested only in the start and end time of each match.

Algorithm 4 Algorithm for the forward-pass phase of lineage construction, in which the pre-lineage graph is created.

Input: Markovian stream $M = (\vec{\mathtt{M}}, \vec{\mathtt{C}})$ with length N; unambiguous DFA query Q

Output: Pre-lineage graph $\langle pV_Q^M, pE_Q^M \rangle$

```
1:  $pV = \{\emptyset\}$; $pE = \{\emptyset\}$
2:  for each $i = 0 \rightarrow N - 1$ do
3:    /* Lines 3-9 create the beginning of sequences starting at instant $i$ */
4:    add node $v_{start}(i)$ to $pV$
5:    for each $(d, p_m) \in \mathtt{M}_i$ do
6:      if there exists a DFA transition $s_0 \rightarrow s' \in Q$ whose predicate is satisfied by $d$ then
7:        add node $v_d^s(i)$ to $pV$
8:        set $(v_{start}(i) \leftarrow v_d^s(i))$.probability = $p_m$
9:        add edge $(v_{start}(i), v_d^s(i))$ to $pE$
10:   /* Lines 10-20 continue or end sequences beginning before instant $i$ */
11:   add node $v_{final}(i + 1)$ to $pV$
12:   for each $v_d^s(i)$ in $pV$ do
13:     for each edge $d \rightarrow d'$ in $\mathtt{C}_i$, with probability $p_e$, do
14:       for each DFA transition $s \rightarrow s'$ satisfied by $d'$ do
15:         add node $v_{d'}^{s'}(i + 1)$ to $pV$ if it does not exist
16:         set $v_d^s(i) \leftarrow v_{d'}^{s'}(i + 1)$.probability = $p_e$
17:         add edge $v_d^s(i) \leftarrow v_{d'}^{s'}(i + 1)$ to $pE$
18:         if $s'$ is an accepting DFA state then
19:           set $v_{d'}^{s'}(i + 1) \leftarrow v_{final}(i + 1)$.probability = 1.0
20:           add edge $v_{d'}^{s'}(i + 1) \leftarrow v_{final}(i + 1)$ to $pE$
```

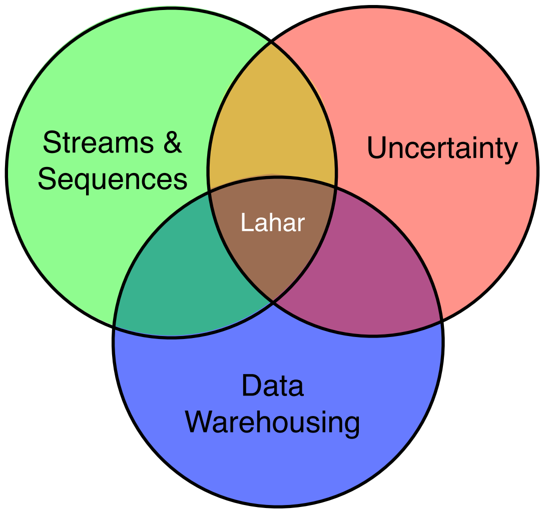
Chapter 7
RELATED WORK

Lahar draws heavily from three major areas of databases research: sequence/stream management, uncertain data management, and data warehousing. The connection to the first two areas is clear in light of the sequential, uncertain nature of Markovian streams. Lahar draws on existing data warehousing work to achieve scalability, and also in its support of aggregations and lineage queries. The relationships between these three major areas and Lahar are depicted in Figure 7.1, which provides the structure for the following discussion of related work.

7.1 Stream/Sequence Management

Early support for sequential data management developed as extensions to standard SQL. These extensions include PREDATOR [110], SRQL [88], and SEQUIN [109]. None of these systems fully supported event queries, because SQL joins cannot express Kleene closure. Later, SQL-TS [104] was proposed with support for Kleene closure. As sensors grew in popularity, real-time data stream management systems emerged, including Aurora/Borealis [2,1], SnoopIB [4], STREAm [10], TelegraphCQ [19] and others [7]. These systems focused on filtering individual stream elements, or on relational sliding-window queries; none supported event queries or imprecise inputs.

More recently, streaming systems have emerged that directly support continuous event queries, including Kleene closure, using finite automata. These include the SASE/SASE+ [128, 48, 5], Cayuga [35,36,13], and ZStream [78] systems. Cayuga, designed to optimize simultaneous processing of many continuous queries, returns only the final timestamp of each pattern match. In contrast, SASE/SASE+ additionally returns the set of atomic input events that contribute to each match. In order to maintain scalability, SASE requires that query matches be limited to a maximum duration; Cayuga (and Lahar) impose no such constraints. Cayuga, however, uses interval-based temporal semantics instead of the atomic instants-based semantics of SASE and Lahar. The recently-proposed ZStream system [78] processes streaming event queries (with lineage) using a tree structure in lieu of



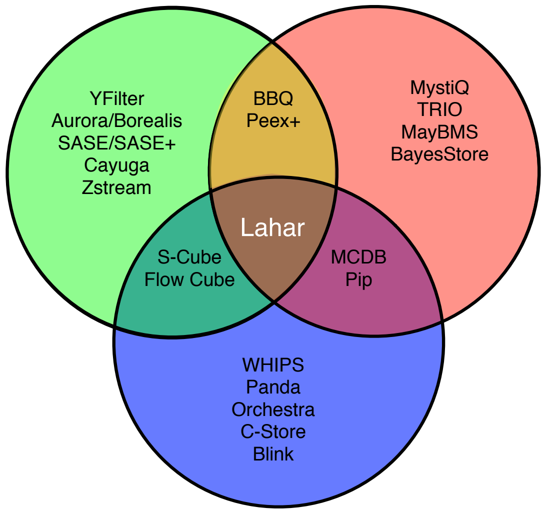


Figure 7.1: (a) High-level view of the areas of database research related to Lahar. (b) Research prototype systems in the areas related to Lahar.

NFAs, allowing it both the flexibility to optimize its query plans and the ability to express predicates over multiple input events. None of these systems support imprecise input streams.

Lahar's event query language is based on that of Cayuga, chosen for its operators' clear semantics. As in Cayuga, Lahar's event query results include only the timestamps at which the event query was matched, except in the case of lineage queries, where the return model more closely resembles that of SASE/SASE+. Lahar moves beyond the capabilities of existing stream/sequence management systems by 1) supporting imprecise input streams and 2) supporting processing techniques beyond sequential, scan-based approaches which are optimized for streaming settings rather than archives.

7.2 Uncertain Data Management

Uncertain data has been a topic of study in both the artificial intelligence and database communities for some time. The AI community has been historically concerned with efficient *inference* and *learning* of probabilistic models (factored joint probability distributions), including hidden Markov models (HMMs) [87], conditional random fields (CRFs) [70], and Bayesian networks [30]. Inference in these models is restricted to the computation of marginal probabilities conditioned on evidence. Inference is important to Lahar because it is the process by which its input (Markovian) streams are generated from raw data; however, probabilistic inference is performed outside of the Lahar system.

In the past several years, the database community has produced several probabilistic data management systems (pDBMS's), including MystiQ [34, 92, 97, 95], Trio [127, 12, 6], and MayBMS [69, 8]. All three systems support relational queries using a possible-worlds semantics: MystiQ using *safe plans* [34], Trio by tracking and processing Boolean lineage formulas [106], and MayBMS using its own probabilistic relational algebra [69]. Like Lahar, these systems operate on imprecise base tables whose imprecise values are given as input (in contrast to the set of systems described next, which may determine these values internally). Unfortunately none of these systems easily supports sequential, imprecise data, and none can express Kleene closure.

An alternate and increasingly-popular approach to uncertain data management uses probabilistic inference, rather than relational operators, to perform query processing. Systems using this approach represent imprecise base tables and the relationships (correlations) between them as a graphical model. Operators add edges and/or nodes to the model such that the modified model contains one or more nodes that represent the query result. Standard probabilistic inference is then used to compute the marginal distributions over these result nodes. This approach has been advocated in the Data Furnace project [42]; by Sen & Deshpande [107, 108], who accelerate inference by identifying commonly-replicated sub-graphs in the probabilistic model; by Kanagal & Deshpande [61,62], who formalize model transforms corresponding to relational and sequencing operators; in the BayesStore system [120]; and by Wang et al. [119] who frame inference as a recursive SQL query on deterministic relations.

Lahar is fundamentally different from these systems because it does not perform online inference. Not only is inference too slow to perform online at warehouse scale, but it would be unnecessary and redundant on Lahar's append-only data model. Lahar is thus closer in spirit to MystiQ or Trio than to inference-based systems.

Finally, a significant amount of work around uncertain data management has focused on *top-k* queries [29, 113, 93], in which only the k most likely query results are computed/returned. The notion of top-k queries originated in traditional databases, but is particularly appropriate in probabilistic databases where every tuple carries an implicit score (its probability), and full result sets may be large and contain an overwhelming number of answers with probabilities so small as to be negligible. Lahar optimizes top-k queries using pruning conditions based on probability-ordered rankings, similar to the pruning used by Fagin et al. [40] on tuple scores.

7.3 Data Warehousing

Data warehouses are repositories for huge archives of historical data. The key challenge of a data warehousing system is to provide rich analytics on its archives [21]. Addressing this challenge requires a data warehouse to provide two components: 1) languages/algorithms for expressing and computing these analytics, and 2) optimization tools that allow these algorithms to run efficiently at scale. Relational data warehouses use a cube model [47] to support rich analytics including aggregations, drill-downs, and in some cases, lineage. They achieve scalability using a variety of techniques including compression, approximation, precomputation, and indexing. The following discussion touches on those aspects of data warehousing that are most relevant to Lahar: aggregation, lineage, compression, and indexing.

7.3.1 Aggregation

Cube-based aggregation is the key concept on which traditional data warehouses are built. In a data cube [47], a *measure* attribute is aggregated at various levels of granularity along each of many *dimensions*. This model has proved highly successful for business analytics and has spawned many variations, including the S-Cube for sequence data [77] and, separately, a cube model for imprecise relational data [17] (discussed in detail in Sections 7.4.2 and 7.4.3, respectively). A major feature of the data cube is its effective use of precomputation, applicable only to *summarizable* data/aggregations [77]. Here, the term summarizable is used in a technical sense to describe aggregations that can be computed at coarse levels of granularity using aggregated results from finer levels (but without touching the underlying base data), such as SUM, MAX, or MIN (but not MEDIAN). Markovian streams are unfortunately not summarizable in this technical sense because of their temporal correlations, and for this reason Lahar does not adopt a cube model in its implementation.

7.3.2 Lineage

The lineage (alternately called provenance) of a query is an expression of a relationship between database tuples, or "source" tuples, and the tuples produced as the output of the query. Different types of lineage express different relationships. For example, "why" lineage expresses links between result tuples and the database tuples that are used to derive the result, while "how" lineage additionally expresses the operations (select, project, join, *etc.*) by which each result tuple is derived from source tuples [24]. The recently-introduced Panda system is intended as a platform for work that unifies the various types of lineage [52].

In relational databases, lineage is often used to determine the derivation of a tuple (i.e. why a particular tuple is present in a query result). Such lineage is useful for view maintenance: Given a query that updates a view, lineage can be used to propagate view changes back to the raw data tables from which the view is derived. The relationship between view maintenance and lineage is explored in the WHIPS warehouse [31,32,33]. Similarly, lineage is useful in systems that associate metadata (e.g. probabilities or confidence scores) with each database tuple, because lineage encodes the computation by which metadata for result tuples is derived. One such system is the probabilistic database Trio, which uses lineage to propagate tuple probabilities through query plans [127].

In some cases, tuple derivation lineage is interesting to users in its own right, particularly when query results contain unexpected values [16, 20]. In this spirit, the recently-proposed ProQL language allows users to query relational lineage directly (for example, to identify the set of result tuples that were generated via a particular sequence of operators) [64]. The WHIPS warehouse also contains a "drill-through" operation that allows users to retrieve lineage for the purpose of examining it.

In scientific databases or data exchange settings, where data is imported or shared across many different sources, lineage is more often called provenance and it is used to identify the original data sources that contribute to the presence of a result tuple [54, 15]. When various data sources are trusted with different levels of confidence, or are updated at different times/frequencies, identification of the sources upon which a result tuple depends is critical. The ORCHESTRA system is a collaborative data sharing system built for such settings; it uses provenance to track tuple derivations [53].

The lineage work most closely related to the work in this thesis is that of Shen et al. [111] and Kimelfeld & Ré [68], who produce subsequence matches of event queries on imprecise streams. Shen et al. consider only *independent* sequences, however. 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 proposed in Chapter 6, 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 [63] generates lineage for *conjunctive* queries on Markov sequences, but does not consider lineage of event queries, or projection of such lineage.

7.3.3 Compression

Scalability is a major concern in data warehouses, and compression has historically been an important tool for achieving it. Warehouse compression is unique in that its goal is to reduce query processing time, which may or may not be achieved simply by reducing data size. The most successful compression techniques are those that produce compressed representations that can be queried directly, eliminating decompression costs. These include the run-length-encoded (RLE) columns of C-Store [3] and the "constant-time" tables of Blink [89, 90]. These compression techniques exploit the relational model's lack of tuple order, and are thus not applicable to Markovian streams, which are inherently ordered. Lahar's compression strategy of materializing multiple (approximately) compressed versions of each Markovian stream is similar at a high level to the work of Chen et al. [23] (who study relational data), however, and Lahar's use of approximate compression techniques to create a performance/accuracy tradeoff is similar to the work of Apaydin et al. [9], who apply approximate compression to relational indices.

7.3.4 Indexing

Indexing is another fundamental tool for achieving scalability in both OLAP and OLTP databases. Work on indexing is vast, and Lahar's use of existing work is fairly straightforward. The most closely-related work is highlighted in the next section.

7.4 Cross-Boundary Related Work

This section describes a more focused set of related works/systems that build on multiple of the three areas related to Lahar. These discussions correspond to the three almond-shaped “intersection” regions of Figure 7.1, and highlight the work most closely related to Lahar.

7.4.1 Uncertain Sequence Management

The two lines of work on uncertain sequence management that are most closely related to Lahar are those by Shen et al. [111] and by Kanagal & Deshpande [59,60,61,63]. Shen et al. effectively extend SASE+ [48] to handle imprecise input streams. The resulting system processes event queries—including lineage—on imprecise input streams, and in this sense is very much like Lahar. However, this work supports only streams with *independent* uncertainty across input timesteps. The temporal correlations of Markovian streams add considerable complexity to size, generation, and querying of Lahar’s lineage.

Kanagal & Deshpande also address Markovian stream management, including event query processing. Their early work [60, 59] proposes a particle filtering method for generating Markovian streams; this is one of many ways by which a Markovian stream might be produced as input to Lahar. Later work proposed a relational algebra for SPJ, aggregate, and sequencing queries on Markovian streams [61]; however, this algebra is executed via probabilistic inference and supports only *fixed-length* event queries (Section 4). More recent work proposed a hierarchical index, INDSEP [62], that accelerates online inference in the graphical models produced by their algebra. This index is a step toward the scalability sought by Lahar, and in fact INDSEP builds upon and generalizes Lahar’s *Markov Chain Index* [73] (Section 4). The most recent work by Kanagal & Deshpande proposes a lineage-based algorithm for computing conjunctive query results on imprecise, correlated data, including streams, but this work does not extend to event queries.

Additional work on uncertain sequence management addresses problems related to Lahar’s Markovian stream management, but tackle different problems or make different assumptions about the data. Much of this work has grown out of efforts to manage spatiotemporal (“moving objects”) or continuously-updated sensor databases. The spatiotemporal work tends to focus on snapshot queries (as opposed to sequence queries), and bounds uncertainty intervals without necessarily assigning a probability distribution over possible worlds [86, 25, 26]. Sensor data management has drawn heavily from the AI community and tends toward a model-based approach in which high-level streams of interest are inferred from noisy input streams using sequential models such as HMMs [87], CRFs [70], or DBNs [76]. Systems in this category have been developed in the context of textual information extraction [119], wireless sensor networks (the BBQ system [37]), and many other types of streaming sensors [38, 115, 60, 59]. The online inference performed in these systems is ideal for real-time, continuous updates but is too slow for the archived warehouse setting targeted by Lahar; furthermore these inference-centric systems lack specialized processing for event queries.

The increasing ubiquity of RFID has led to some efforts to "clean" noisy RFID streams (e.g. the SMURF project [56] and others [14, 22, 91]) or to detect events probabilistically directly on the raw input (e.g. the PEEX [66, 67] and Cascadia [126] systems). These efforts use ad-hoc approaches with unclear semantics, and additionally detect sequences using relational operators (SQL) instead of finite automata.

Finally, some work on uncertain stream processing is related to Lahar only tangentially. In particular, efforts to compute statistical aggregates [55], sketches [28], or synopses [57] over the elements of imprecise streams use a streaming processing model, but do not support ordered streams (i.e. the order in which stream elements arrive is unimportant in these models).

7.4.2 Warehousing Sequences

The sequence warehousing system most closely related to Lahar is the S-Cube by Lo et al. [77]. S-Cube's novel contribution is its support for pattern-based sequence aggregation, which effectively allows `GROUP-BY`s on pattern predicates specified at arbitrary granularities. In contrast to Lahar, S-Cube supports only *fixed-length* event queries (Section 4) and deterministic input sequences; however, Lahar's cube-based aggregation model follows very much in the spirit of S-Cube. Both systems extend cube models to *non-summarizable* data, forcing them to compute all queries directly on the lowest-level data, without leveraging precomputed views.

The majority of remaining work on scalable, archived sequence management has appeared in the context of supply-chain RFID management. These approaches leverage RFID-specific properties that do not hold for general Markovian streams. Hu et al. [50] exploit the hierarchical nature of supply chain objects (products inside cases inside pallets), while Lee & Cheung [72] exploit the shared-prefix nature of supply chain trajectories, to achieve significant data compression. The work of Wang [121, 122] and separately, Gonzales [44, 45] achieve significant speedup by compressing consecutive detection records into "stay" intervals. Gonzales develops this notion into a cube-based warehousing model for RFID streams (FlowCube [43]) in which each cube cell contains a representative trajectory, and if necessary, a list of significant exceptions. The underlying notion of FlowCube—that presenting an easily-understood view of data is more important than presenting a full, precise enumeration—is adopted in Lahar's approach to lineage.

A small amount of sequence warehousing work targets indexing techniques for sequential data, although such work focuses on streaming, rather than archived, settings. The goal of this work is efficient retrieval of the system state that must be updated based on the current stream input. For example, Tran et al. [115] leverage indices to identify a subset of RFID tags whose locations must be updated, while the Cayuga system [35] leverages indices to identify the subset of NFAs that are affected by the current stream input. Lahar does not currently support streaming queries and thus these indexing techniques solve an orthogonal problem to that of Lahar.

7.4.3 Warehousing Uncertain Data

Several systems have recently emerged that support some combination of aggregation and/or lineage in a scalable manner on imprecise data. None of these systems support sequential data and thus they are not directly applicable to Markovian stream warehousing; however, Lahar draws on the novel ideas and techniques used in these systems.

Aggregation & Uncertainty

Cube-based aggregation on imprecise data is more complex than in deterministic settings, because pre-existing or query-induced correlations must be correctly handled. Three cube-based or similar models for imprecise, relational data have been recently proposed: First, Burdick et al. [17] study the problem of OLAP on relational warehouses containing imprecision in the *dimension* attributes. This work focuses on methods for assigning probability distributions over these imprecise dimensions; it does not discuss algorithms for query execution. Second, Li et al. propose a Sampling Cube [75] for performing OLAP over sampled data; here, the focus is on generalizing cube cells until they contain a meaningful number of samples, without altering the statistical properties of aggregated query results. Finally, Jampani et al. propose MCDB, which is not strictly a cube but can support cube-style analytics using its sampling-based query execution model. MCDB is similar to inference-based systems in that it can easily represent arbitrarily-correlated data (including Markovian streams), but it performs inference via sampling instead of exact techniques. Although MCDB does not support event queries, its support for "what-if" queries conditioned on hypothetical probability distributions is a novel and interesting form of imprecise data analytics.

Lineage & Uncertainty

Several imprecise data management systems track lineage, not only to support lineage-related queries, but as a tool for computing the probabilities associated with query results. Trio [6], Pip [65], and work by Ré et al. [96] all track lineage for *relational*, imprecise data. This lineage takes the form of boolean expressions, which can be evaluated either exactly or approximately to determine the probability of a query result. This work addresses lineage for conjunctive queries on relational data, and does not extend to event queries on sequences like Markovian streams.

Indexing & Uncertainty

Although indexing techniques have been heavily studied for traditional databases and warehouses, relatively little work has focused on development of similar techniques specifically for imprecise data. The two most relevant works are the INDSEP index developed by Kanagal & Deshpande [62] (already discussed in Section 7.4.1) and two indices proposed by Singh et al. [112] for categorical, uncertain data. Singh et al.'s first index is a B+ tree equivalent to Lahar's BT_P index (because they do not support sequential data, Singh et al. of course do not explore the use of this index for efficiently supporting event query computation). The other is a modified R-tree that supports efficient retrieval of tuples based on their entire uncertain distribution. This index is useful for selection of tuples with "similar" distributions, which are not a focus of Lahar.

Additional work on stream indexing includes that of Reiss et al. [99], who focus on indices for streaming data that can be efficiently updated as archives continuously grow; and Cheng et al. [27]'s novel R-tree that allows moving objects with (uniform) uncertain location to be indexed by the probability with which they intersect various spatial regions. Tao et al. [114] extend this index to support arbitrary probability distributions; however, Lahar's event queries do not require support for the type of region-based queries targeted by these indices.

Compression & Uncertainty

Compression of imprecise data is only sparsely studied. The most relevant work is that of Tran et al. [115] who compress discrete spatial distributions into more compact, continuous distributions (e.g. Gaussians) when possible. Compressing Markovian streams in this way is an interesting possibility for Lahar, although event query processing is currently supported only on streams with uncertainty over discretely-valued domains.

Chapter 8
CONCLUSION & FUTURE WORK

This dissertation has described the challenges associated with sophisticated processing of imprecise sequence archives, and has presented a set of algorithms and data structures that make this processing efficient. We propose a Markovian stream representation for modeling imprecise sequences, and introduce the Lahar database for managing and querying these streams. Lahar includes a SQL-like query language and programmatic API for interfacing with applications. Internally, it supports efficient event query processing using a combination of matrix-based processing, B+ tree and novel indices, and novel stream approximations. Lahar also supports novel, sophisticated Markovian stream queries including aggregations within and across streams, and lineage queries including those with projection. Concretely, the novel contributions of this thesis are as follow:

- **A Markovian stream approach to imprecise sequence management.** This thesis proposed the use of materialized Markovian streams to separate the probabilistic inference and data management/querying portions of imprecise data management. Central to this approach is the Lahar database for managing Markovian streams efficiently; the technical contributions of this thesis are implemented in a Lahar prototype.

- **Markovian stream indices.** We developed novel adaptations of B+ tree indices and a novel Markov Chain index, which together improved Markovian stream processing speeds by up to two orders of magnitude on real-world Markovian streams.

- **Markovian stream approximations.** We proposed and evaluated four different Markovian stream approximations, some of which can improve performance by two orders of magnitude. We broadly characterized the error incurred by these approximations, demonstrating that error is sensitive to both the approximation and the type of query.

- **Markovian stream lineage queries.** We defined a set of lineage queries and developed a novel lineage graph structure to represent the results. We also developed $O(N)$ and $O(N^2)$ algorithms for constructing the lineage graph and projecting away arbitrary sets of its edges, respectively. We demonstrated that in practice, the overhead of tracking event query lineage is often a fraction of the total query processing cost, and that the overhead can in some cases be reduced by applying projection at different points in the lineage processing pipeline.

Future Work

This thesis has demonstrated that Markovian stream processing is not only useful, but can be performed practically in a system like Lahar. Nevertheless, there are many topics related to Markovian stream processing that are not covered in this thesis, and which represent interesting areas for future work. Here we outline a few of these topics, beginning with narrowly-scoped challenges specific to improving Lahar's functionality, and ending with broader challenges relevant to imprecise stream processing in general.

Narrowly-Scoped Challenges

We have identified several challenges that, if addressed would significantly improve the performance and functionality of Lahar. We outline these challenges here.

- **Automatic Query Optimization:** Currently, Lahar's query optimizer doesn't compare multiple query plans: it relies on user input, given through the programmatic API, to specify which indices or which approximations (or both) should be used to execute a query. Ideally, the optimizer would compare these alternatives using a cost model and select the optimal plan on its own.

  Automatic query optimization in Lahar faces several challenges. First, to properly select an approximation scheme, Lahar requires access to a cost function that includes estimates of both accuracy and efficiency, as well as some user-supplied measure of equivalence between these two factors. As mentioned in Chapter 5, a method for reliably predicting the accuracy of various approximation schemes on different queries is difficult, and an area for future work.

  Indexing, too, presents challenges for automatic query optimization in Lahar. Recall that the

MC index can be used only when every predicate in a query is indexed. Currently, Lahar does not create indices on the fly, but an optimizer might make a real-time decision about whether creating a missing index on the fly in order to leverage the MC index in a query might be worthwhile.

- **Query Language:** Lahar's current query language is limited in several important ways. First, it allows expression only of linear NFAs, when the basic query processing algorithm can process any type of NFA. Furthermore, the current language only expresses NFAs in which any loop predicate for a given state s is identical to the predicate on the forward (non-loop) edge entering s. There is an inherent trade-off between a language's expressiveness and its simplicity, and we designed the current language to err on the side of simplicity (note that applications can submit arbitrary NFAs for processing through the programmatic API). Development of a simple, flexible way to specify arbitrary NFA structures is an interesting are for future work in many stream processing systems, including those that process deterministic streams.

  Lahar's query language also suffers from an inability to allow users to specify many details of lineage queries. In particular, the language currently provides no way for applications to specify projection on lineage queries. Currently this must be done through the programmatic API only. The same restriction applies for specifying the value of k in lineage queries.

- **Distributed Markovian Stream Processing:** In practice, real-world Markovian stream archives may include thousands of individual streams, each of which may represent days, weeks, or even months' worth of data. Scalable management of such an archive requires a distributed system, for both storage and query processing.

  The current Lahar prototype is a single-threaded, single-machine system. Although the processing of multiple, independent Markovian streams is an embarrassingly parallel problem, several challenges are involved in creating a parallelized algorithm for processing a single Markovian stream. One primary challenge is that Lahar's query processing algorithms for variable-length queries all rely on a sequential scan of the data—even when the MC index is used to skip over intervals, the instants that require processing must be processed in sequential order. In order to effectively parallelize query processing, a distributed algorithm must be able to split a single stream into multiple units that can be processed in parallel. Straightforward, independent processing of individual stream segments is incorrect in the presence of Markovian correlations; instead, a parallelized algorithm must process each segment conditionally, and later use the results from earlier segments to condition the results from later ones. Algorithms for performing this conditioning efficiently are an area for future work.

Distributing Lahar across multiple machines and disks presents additional challenges. A distributed system must include policies to determine the distribution of Markovian streams on disk: Should all similar Markovian streams be co-located (e.g. all of Bob's streams, or all crash cart streams) on a single disk, or is it more effective to co-locate snippets of streams that represent a wide variety of individuals/objects but that represent the same interval in time? Answering these questions requires analysis of real-world query workloads, and it is possible that applications requirements vary widely based on differences in their workloads.

Broadly-Scoped Challenges

In this section, we highlight two broadly-defined challenges that we believe are the primary remaining barriers to widespread adoption of the Markovian stream approach to imprecise sequence management. These challenges are:

- **Query signal utility model.** Recall Figure 6.1, which shows an event query signal. For each instant *i* in an input stream, the query signal indicates the probability that the event query in question is satisfied at instant *i*. Although this signal is a precise representation of the query answer, it is difficult for applications to make use of this signal without additional context. For example, noise in the signal results in many instants having a non-zero but very small probability of satisfying the query. Applications can handle this noise by thresholding the query output so that only instants with probability above a certain threshold value are considered. One challenge inherent to this approach, however, is how the threshold value should be chosen. Appropriate threshold values can change depending on differences in the input stream domain, the correlations in the data, and the length of a given query. Currently, application thresholds must be calibrated by a human user in a trial-and-error approach. A better approach, and one for consideration in future work, is to build a utility model into Lahar itself. Doing so might allow Lahar to suggest appropriate thresholds to applications, based on statistics gathered on Markovian stream data and/or past query workloads.

Lahar's current query signal output additionally causes difficulty for applications wishing to correlate query match *intervals* with real-world events. Consider the query signal in Figure 6.1: a single real-world event at instant 1104 creates an interval of signal peaks beginning around instant 900 and ending around instant 1150. Each of the instants in this interval satisfies the query with non-trivial probability because, although the real-world event occurred only once, the exact instant at which it occurred is highly uncertain. Another interesting challenge for future work is the development of techniques to allow Lahar to correlate high-probability intervals in the query signal with real-world events. One possible way to provide this information is using lineage: stream instants with overlapping lineage are more likely to be mutually exclusive and to correspond to a single real-world event. Another possible technique for handling these intervals is to smooth them out using post-processing on the query signal itself. In either case, the question of how to assign probabilities to the result—and of what format the result should take—are open for future work.

- **Markovian stream generation.** Both the Markovian stream approach to imprecise sequence management, and the Lahar database, are agnostic to the method used to generate Markovian streams from imprecise inputs (e.g. sensor streams). However, Markovian streams are currently difficult to generate without sophisticated knowledge of probabilistic models, and without knowledge of the Markovian stream domain. Because this knowledge is relatively rare even among technical populations, conversion of imprecise sequence data into Markovian streams is a significant barrier to the adoption of Lahar—even if Lahar itself is highly intuitive and easy to use.

  Although many algorithms and toolkits exist for generating Markovian streams (e.g. the HTK toolkit [81], the HMM toolkit [79], *etc.*), these toolkits require significant knowledge about probabilistic models and inference. In order for Lahar to achieve widespread adoption, a simplified, intuitive tool for generating Markovian streams is required. Of course, development of such a tool is challenging because successful probabilistic inference requires an appropriate data model as well as appropriate parameters for describing sensor noise and transition probabilities (likelihoods of various word sequences, for example, or the connectivity between various locations in a building).

We believe that development of a simplified, intuitive tool for Markovian stream generation is possible, and might be developed by making several assumptions. The first assumption is that data can be appropriately modeled using a Hidden Markov Model, the simplest form of graphical model that can produce a Markovian stream. Although more sophisticated models might produce higher-quality Markovian streams, providing a default structure alleviates the need for users to explicitly specify a model. The second assumption is that the parameters of the HMM can be learned from training data. Taking this approach requires that users provide a small amount of input data labeled with true values (e.g. the actual locations visited, or words spoken), but it removes the need for users to measure or characterize sensor noise, or to provide language models or building maps. The quality of Markovian streams generated using models constructed or learned in this manner is an interesting question for future analysis.

Conclusion

Markovian streams are well-suited to model a huge range of low-level, imprecise data, including speech, location sensors, smart homes, *etc.*. The higher-level information (location, activity, *etc.*) contained in these streams is valuable to applications wishing to perform a variety of tasks, including keyphrase searches in audio data, operations support in hospital or office buildings, *etc.*. Supporting such analytics efficiently on Markovian streams requires novel algorithms and systems. The algorithms and data structures developed in the context of Lahar, as well as the Lahar system itself, are a first step toward the goal of extracting utility from Markovian stream data. While the event query processing, aggregations, and lineage processing supported by Lahar are important, many research challenges remain to be addressed before applications can leverage the full power of Markovian stream data. We hope that in time, low-level stream data will be as easy and efficient to analyze as relational data is today.

BIBLIOGRAPHY

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proc. of the 2nd CIDR Conf.*, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *Proc. of the 29th VLDB Conf.*, 12(2):120–139, 2003.

[3] D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. of the SIGMOD Conf.*, pages 671–682, Chicago, IL, USA, 2006.

[4] R. Adaikkalavan and S. Chakravarthy. Snoopib: interval-based event specification and detection for active databases. *Data Knowl. Eng.*, 59(1):139–165, 2006.

[5] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proc. of the SIGMOD Conf.*, pages 147–160, New York, NY, USA, 2008. ACM.

[6] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *Proc. of the 32nd VLDB Conf.*, pages 1151–1154, 2006.

[7] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[8] L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing incomplete information with probabilistic world-set decompositions (demonstration). In *Proc. of the 23rd ICDE Conf.*, 2007.

[9] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A. S. Tosun. Approximate encoding for direct access and query processing over compressed bitmaps. In *Proc. of the 32nd VLDB Conf.*, pages 846–857. VLDB Endowment, 2006.

[10] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Springer, 2004.

[11] P. Bahl and V. N. Padmanabhan. Radar: an in-building rf-based user location and tracking system. In *Proceedings of IEEE Infocom*, volume 2, pages 775–784, 2000.

[12] O. Benjelloun, A. D. Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. Technical Report 2007-26, Stanford InfoLab, 2007.

[13] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *Proc. of the SIGMOD Conf.*, pages 1100–1102, New York, NY, USA, 2007. ACM Press.

[14] 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.

[15] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *Proc. of the SIGMOD Conf.*, pages 539–550, New York, NY, USA, 2006. ACM.

[16] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proc of the 8th ICDT Conf.*, pages 316–330, 2001.

[17] D. Burdick, P. M. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. Olap over uncertain and imprecise data. *Proc. of the 33rd VLDB Conf.*, 16(1):123–144, 2007.

[18] V. Chakka, A. Everspaugh, and J. Patel. Indexing large trajectory data sets with seti, 2003.

[19] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[20] A. Chapman and H. V. Jagadish. Why not? In *Proc. of the SIGMOD Conf.*, 2009.

[21] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. 26(1):65–74, 1997.

[22] S. S. Chawathe, V. Krishnamurthy, S. Ramachandran, and S. E. Sarma. Managing rfid data. In *Proc. of the 30th VLDB Conf.*, pages 1189–1195, 2004.

[23] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *Proc. of the SIGMOD Conf.*, pages 271–282, New York, NY, USA, 2001. ACM.

[24] 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.

[25] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluation of probabilistic queries over imprecise data in constantly-evolving environments. *Inf. Syst.*, 32(1):104–130, 2007.

[26] R. Cheng and S. Prabhakar. Managing uncertainty in sensor database. *Proc. of the SIGMOD Conf.*, 32(4):41–46, 2003.

[27] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proc. of the 30th VLDB Conf.*, pages 876–887, 2004.

[28] G. Cormode and M. Garofalakis. Sketching probabilistic data streams. In *Proc. of the SIGMOD Conf.*, pages 281–292, New York, NY, USA, 2007. ACM.

[29] 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.

[30] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer, 1999.

[31] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. *Proc. of the 16th ICDE Conf.*, 0:367, 2000.

[32] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *Proc. of the 27th VLDB Conf.*, pages 471–480, 2001.

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

[34] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proc. of the 30th VLDB Conf.*, pages 864–875, 2004.

[35] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. of the EDBT Conf.*, pages 627–644, 2006.

[36] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. pages 412–422, 2007.

[37] 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.

[38] Y. Diao, B. Li, A. Liu, L. Peng, C. Sutton, T. Tran, and M. Zink. Capturing data uncertainty in high-volume stream processing. 2009.

[39] A. Doucet, S. Godsill, and C. Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and Computing*, 10(3):197–208, 2000.

[40] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the 20th PODS Conf.*, 2001.

[41] J. Forney, G.D. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268 – 278, march 1973.

[42] M. N. Garofalakis, K. P. Brown, M. J. Franklin, J. M. Hellerstein, D. Z. Wang, E. Michelakis, L. Tancau, E. W. 0002, S. R. Jeffery, and R. Aipperspach. Probabilistic data management for pervasive computing: The *data furnace* project. *IEEE Data Eng. Bull.*, 29(1):57–63, 2006.

[43] H. Gonzalez, J. Han, and X. Li. Flowcube: Constructing rfid flowcubes for multi-dimensional analysis of commodity flows. In *Proc. of the 32nd VLDB Conf.*, 2006.

[44] H. Gonzalez, J. Han, and X. Li. Mining compressed commodity workflows from massive rfid data sets. In *Proc. of the 15th CIKM Conf.*, pages 162–171, 2006.

[45] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and analyzing massive rfid data sets. In *Proc. of the 22nd ICDE Conf.*, page 83, Washington, DC, USA, 2006. IEEE Computer Society.

[46] H. Gonzalez, J. Han, and X. Shen. Cost-conscious cleaning of massive rfid data sets. In *Proc. of the 23rd ICDE Conf.*, pages 1268–1272. IEEE, 2007.

[47] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Proc. of the 3rd KDD Conf.*, 1(1):29–53, 1997.

[48] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. In *Proc. of the 24th ICDE Conf.*, pages 1391–1393, 2008.

[49] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. pages 389–400. ACM, 2007.

[50] Y. Hu, S. Sundara, T. Chorma, and J. Srinivasan. Supporting rfid-based item tracking applications in oracle dbms using a bitmap datatype. In *Proc. of the 31st VLDB Conf.*, pages 1140–1151. VLDB Endowment, 2005.

[51] IDC. The expanding digital universe: A forecast of worldwide information growth through 2010. An IDC White Paper sponsored by EMC., March 2007.

[52] R. Ikeda and J. Widom. Panda: A system for provenance and data. In *Proc of the 2nd TaPP Workshop.* Stanford InfoLab, 2010.

[53] Z. G. Ives, T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira. The orchestra collaborative data sharing system. *Proc. of the SIGMOD Conf.*, 37(3):26–32, 2008.

[54] M. Jayapandian, A. Chapman, V. G. Tarcea, C. Yu, A. Elkiss, A. Ianni, B. Liu, A. Nandi, C. Santos, P. Andrews, B. Athey, D. J. States, and H. V. Jagadish. Michigan molecular interactions (mimi): putting the jigsaw puzzle together. *Nucleic Acids Research, Database Issue*, 35:566–571, 2007.

[55] T. S. Jayram, A. McGregor, S. Muthukrishnan, and E. Vee. Estimating statistical aggregates on probabilistic data streams. In *Proc. of the 26th PODS Conf.*, pages 243–252, 2007.

[56] S. R. Jeffery, M. Garofalakis, and M. J. Franklin. Adaptive cleaning for rfid data streams. In *Proc. of the 32nd VLDB Conf.*, pages 163–174. VLDB Endowment, 2006.

[57] C. Jin, K. Yi, L. C. 0002, J. X. Yu, and X. Lin. Sliding-window top-k queries on uncertain streams. *Proc. of the 34th VLDB Conf.*, 1(1):301–312, 2008.

[58] M. I. Jordan, editor. *Learning in graphical models*. MIT Press, Cambridge, MA, USA, 1999.

[59] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. Technical report, University of Maryland, May 2007.

[60] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. In *Proc. of the 24th ICDE Conf.*, pages 1160–1169, 2008.

[61] B. Kanagal and A. Deshpande. Efficient query evaluation over temporally correlated probabilistic streams. In *Proc. of the 25th ICDE Conf.*, pages 1315–1318, 2009.

[62] B. Kanagal and A. Deshpande. Indexing correlated probabilistic databases. In *Proc. of the SIGMOD Conf.*, 2009.

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

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

[65] O. Kennedy and C. Koch. PIP: A database system for great and small expectations. In *Proc. of the 26th ICDE Conf.*, 2010.

[66] N. Khoussainova, M. Balazinska, and D. Suciu. Probabilistic rfid data management. Technical report, University of Washington CSE, June 2007.

[67] N. Khoussainova, M. Balazinska, and D. Suciu. Probabilistic event extraction from rfid data. In *Proc. of the 24th ICDE Conf.*, pages 1480–1482, 2008.

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

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

[70] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. of the 18th ICML Conf.*, pages 282–289, 2001.

[71] T. Lancaster and F. Culwin. Towards an error free plagarism detection process. In *Proc of the 6th ITiCSE Conf.*, pages 57–60, New York, NY, USA, 2001. ACM.

[72] C.-H. Lee and C.-W. Chung. Efficient storage scheme and query processing for supply chain management using rfid. In *Proc. of the SIGMOD Conf.*, pages 291–302, New York, NY, USA, 2008. ACM.

[73] 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.

[74] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi. An introduction to the application of the theory of probabilistic functions of a Markov process to automatic speech recognition. *Bell Sys. Tech. J.*, 62:1035, 1983.

[75] X. Li, J. Han, Z. Yin, J.-G. Lee, and Y. Sun. Sampling cube: a framework for statistical olap over sampling data. In *Proc. of the SIGMOD Conf.*, pages 779–790, New York, NY, USA, 2008. ACM.

[76] L. Liao, D. J. Patterson, D. Fox, and H. A. Kautz. Learning and inferring transportation routines. *Artif. Intell*, 171(5-6):311–331, 2007.

[77] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung. Olap on sequence data. In J. T.-L. Wang, editor, *Proc. of the SIGMOD Conf.*, pages 649–660. ACM, 2008.

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

[79] K. Murphy. The hmm toolkit. http://www.cs.ubc.ca/ murphyk/Software/HMM/hmm.html, 1998.

[80] A. Nemmaluri, M. D. Corner, and P. J. Shenoy. Sherlock: automatically locating objects for humans. In *Proc. of the 6th MobiSys Conf.*, pages 187–198, 2008.

[81] J. Odell, D. Ollason, P. Woodland, S. Young, and J. Jansen. *The HTK Book for HTK V2.0*. Cambridge University Press, Cambridge, UK, 1995.

[82] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[83] New Oregon Hospital Adopts IR-RFID Hybrid System.
http://www.rfidjournal.com/article/view/4846/1, May 2009.

[84] T. J. Parr and R. W. Quong. ANTLR: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[85] 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.

[86] D. Pfoser and C. S. Jensen. Querying the trajectories of on-line mobile objects. In *Proc. of the 2nd MobiDE Workshop*, pages 66–73, 2001.

[87] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[88] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, and M. Krishnaprasad. Srql: Sorted relational query language. In *Proc. of the SSDBM Conf.*, pages 84–95. IEEE Computer Society, 1998.

[89] V. Raman and G. Swart. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *Proc. of the 32nd VLDB Conf.*, pages 858–869. VLDB Endowment, 2006.

[90] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *Proc. of the 24th ICDE Conf.*, pages 60–69. IEEE, 2008.

[91] J. Rao, S. Doraiswamy, H. Thakkar, and L. S. Colby. A deferred cleansing method for rfid data analytics. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 175–186. Proc. of the 32nd VLDB Conf., 2006.

[92] C. Ré, N. Dalvi, and D. Suciu. Query evaluation on probabilistic databases. In *IEEE Data Engineering Bulletin*, volume 29, pages 25–31, 2006.

[93] C. Ré, N. Dalvi, and D. Suciu. Efficient Top-k query evaluation on probabilistic data. In *Proc. of the 23rd ICDE Conf.*, 2007.

[94] 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.

[95] C. Ré and D. Suciu. Management of data with uncertainties. In *Proc. of the 16th CIKM Conf.*, pages 3–8, New York, NY, USA, 2007. ACM.

[96] C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *Proc. of the 34th VLDB Conf.*, 1(1):797–808, 2008.

[97] C. Ré and D. Suciu. Managing probabilistic data with mystiq: The can-do, the could-do, and the can't-do. In *Proc of the 2nd SUM Conf.*, pages 5–18, Berlin, Heidelberg, 2008. Springer-Verlag.

[98] G. Reeves, J. Liu, S. Nath, and F. Zhao. Managing massive time series streams with multiscale compressed trickles. 2(1):97–108, 2009.

[99] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling real-time querying of live and historical stream data. In *Proc of the 19th SSDBM Conf.*, page 28, Washington, DC, USA, 2007. IEEE Computer Society.

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

[101] RFID Update. Reva Launches RTLS Solution Based on Passive RFID.
http://www.rfidupdate.com/articles/
index.php?id=1774, Apr. 2009.

[102] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. In *Proc. of the ICCV Conf.*, pages 59–66, 1998.

[103] S. J. Russell and P. Norvig. *Artificial intelligence : a modern approach*. Prentice Hall, 2nd edition, 2003.

[104] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM TODS.*, 29(2):282–318, 2004.

[105] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. In *INFORMATION PROCESSING AND MANAGEMENT*, pages 513–523, 1988.

[106] 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.

[107] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *Proc. of the 23rd ICDE Conf.*, pages 596–605, 2007.

[108] P. Sen, A. Deshpande, and L. Getoor. Exploiting shared correlations in probabilistic databases. *Proc. of the 34th VLDB Conf.*, 1(1):809–820, 2008.

[109] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proc. of the 22nd VLDB Conf.*, pages 99–110. Morgan Kaufmann, 1996.

[110] P. Seshadri and M. Paskin. Predator: an or-dbms with enhanced data types. 26(2):568–571, 1997.

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

[112] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. E. Hambrusch. Indexing uncertain categorical data. In *Proc. of the 23rd ICDE Conf.*, pages 616–625, 2007.

[113] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *Proc. of the 23rd ICDE Conf.*, pages 896–905, 2007.

[114] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *Proc. of the 31st VLDB Conf.*, pages 922–933, 2005.

[115] T. Tran, C. Sutton, R. Cocci, Y. Nie, Y. Diao, and P. Shenoy. Probabilistic inference over rfid streams in mobile environments. In *Proc. of the 25th ICDE Conf.*, 2009.

[116] University of Washington. RFID Ecosystem. `http://rfid.cs.washington.edu/`.

[117] R. van der Togt, E. J. van Lieshout, R. Hensbroek, E. Beinat, J. M. Binnekade, and P. J. M. Bakker. Electromagnetic interference from RFID inducing potentially hazardous incidents in critical care medical equipment. *Journal of the American Medical Association*, 299:2884–2890, 2008.

[118] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel. Sphinx-4: A flexible open source framework for speech recognition. Technical report, 2004.

[119] 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.

[120] D. Z. Wang, E. Michelakis, M. N. Garofalakis, and J. M. Hellerstein. BayesStore: managing large, uncertain data repositories with probabilistic graphical models. *Proc. of the 34th VLDB Conf.*, 1(1):340–351, 2008.

[121] F. Wang and P. Liu. Temporal management of rfid data. In *Proc. of the 31st VLDB Conf.*, pages 1128–1139. VLDB Endowment, 2005.

[122] F. Wang, S. Liu, P. Liu, and Y. Bai. Bridging physical and virtual worlds: Complex event processing for rfid data streams. In *Proc. of the EDBT Conf.*, pages 588–607, 2006.

[123] `http://hr.dop.wa.gov/eln/`.

[124] E. Welbourne, M. Balazinska, G. Borriello, and W. Brunette. Challenges for pervasive rfid-based infrastructures. In *Proc. of the PerCom Conf.*, pages 388–394. IEEE Computer Society, 2007.

[125] E. Welbourne, L. Battle, G. Cole, K. Gould, K. Rector, S. Raymer, M. Balazinska, and G. Borriello. Building the internet of things using rfid: The rfid ecosystem experience. *IEEE Internet Computing*, 13(3):48–55, 2009.

[126] E. Welbourne, N. Khoussainova, J. Letchner, Y. Li, M. Balazinska, G. Borriello, and D. Suciu. Cascadia: a system for specifying, detecting, and managing rfid events. In D. Grunwald, R. Han, E. de Lara, and C. S. Ellis, editors, *Proc. of the 6th MobiSys Conf.*, pages 281–294. ACM, 2008.

[127] J. Widom. Trio: A system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data*. Springer, 2008.

[128] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. of the SIGMOD Conf.*, pages 407–418, New York, NY, USA, 2006. ACM Press.