Access Methods for Markovian Streams University of Washington Technical Report UW TR: #TR08-07-01

Julie Letchner [#1], Christopher Ré [#2], Magdalena Balazinska [#3], Matthai Philipose [*4]

[#]*Computer Science & Engineering Department, University of Washington*
Seattle, Washington, USA
[1]letchner, [2]chrisre, [3]magda@cs.washington.edu

[*]*Intel Research Seattle*
Seattle, Washington, USA
[4]matthai.philipose@intel.com

Abstract

Model-based views have recently been proposed as an effective method for querying noisy sensor data. Commonly used models from the AI literature (*e.g.*, the Hidden Markov Model) expose to applications a stream of probabilistic and correlated state estimates computed from the sensor data. Many applications want to detect sophisticated patterns of states from these *Markovian streams*. Such queries are called *event queries*.

In this paper, we present a new system, Caldera, for processing event queries over stored Markovian streams. At the heart of our system is a set of access methods for Markovian streams that can improve event query performance by orders of magnitude compared to existing techniques, which must scan the entire stream. These access methods use new adaptations of traditional B+ tree indexes, and a new index, called the *Markov-chain index*, to efficiently extract only those parts of the stream potentially relevant to the query while retaining the stream's Markovian properties. We have implemented our prototype system on BDB and demonstrate its effectiveness on both synthetic data and real data from a building-wide RFID deployment.

1 Introduction

Applications that make critical decisions based on sensor data are increasingly common, with sensor deployments now playing integral roles in supply chain automation [5, 39], environment monitoring [17], elder-care [25, 28], etc. Unfortunately, building applications on top of raw sensor data remains challenging because sensors produce inaccurate information, frequently fail, and can rarely collect data on an entire region of interest. As an example, consider a Radio Frequency IDentification (RFID) tracking application [38] in which RFID readers are distributed throughout an environment. Ideally, when a tag (carried by a person or attached to an object) passes in the vicinity of a reader, the reader detects and logs the tag's presence: *e.g.*, Bob's tag was sighted by reader A at time 7, reader B at time 8, etc. In practice, however, readers often fail to detect nearby tags [40], forcing applications to deal with sparse and noisy input streams.

The reduction of errors and gaps in sensor data streams is the focus of a large body of techniques developed in the AI community [34]. While a limited number of these techniques can be applied in real time, the most effective (*Bayesian smoothing* [13]) can be applied only as a post-processing step, after the raw data stream is stored on disk. Our goal is to support archive-based applications that leverage this smoothed data in order to provide the most accurate possible answers to historical queries (*e.g.*, "Was Bob in his office yesterday?", "Did Margot take her medication before breakfast every day last month?", etc.).

The result of any smoothing technique is a *probabilistic* stream, in which each timestep encodes not a single state, but a distribution over possible states. In the RFID tracking example, such a stream might indicate, for each timestep, the distribution over possible locations of a tag: *e.g.*, at time 7, Bob was in the hallway with probability 0.8 and in his office with probability 0.2. Additionally, states at consecutive timesteps can be correlated: *e.g.*, Bob's location at time 8 is correlated with

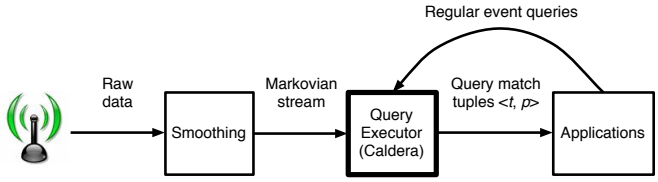


Figure 1: The flow of data in Caldera. Raw sensor data is first smoothed into Markovian streams and archived on disk. Caldera processes application-specified Regular event queries on these archived Markovian streams and returns the result tuples to the applications.

his location at time 7. We call these probabilistic, correlated streams *Markovian streams*. They are a materialized instance of a *model-based view* [10].

A natural class of queries on Markovian streams are *event queries* [1, 9, 33, 43], which find sophisticated patterns of states in streams. An example event query on an RFID stream is: *"When did Bob take a coffee break yesterday?"*, where "coffee-break" is defined as Bob going from the coffee machine directly to the lounge.

Because of the probabilities and correlations present in a Markovian stream, archived event queries on these streams cannot be handled by any existing system. Indeed, stream processing engines that handle event queries [1, 9, 43] ignore probabilities. Probabilistic databases [3, 8, 20, 42], on the other hand, handle data uncertainty but do not support event queries and many do not handle correlations [3, 8, 42]. The only existing system able to manage Markovian streams is Lahar [33], which we previously developed to process *real-time* event queries. In order to preserve correlations, Lahar must process *every* timestep of the input stream. This full-data scan is grossly inefficient in archived settings.

In this paper, we present Caldera, a novel system that efficiently processes event queries on archived Markovian streams. The challenge of this system is to identify and process only the relevant portions of a Markovian stream in a manner that preserves cross-timestep correlations, but *without* scanning the entire stream history on disk. Caldera achieves this goal by using a battery of access methods that leverage pruning and precomputation strategies.

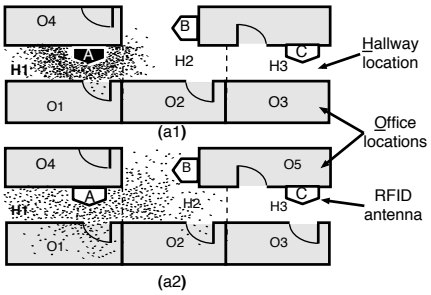
In Section 2, we formally define Markovian streams and the event queries processed by Caldera. An important contribution in this section is our division of these queries into two classes, which we call *fixed-length* and *variable-length* queries. We identify the challenges involved in processing each. Our algorithmic contributions are access methods optimized for each query class. Specifically, these contributions include:

1. An access method optimized for *fixed-length* queries, based upon a novel adaptation of standard indexing techniques (Section 3.1).
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 techniques to Markovian stream data, using standard B+ trees (Section 3.2).
3. A novel index structure (the Markov chain index) and associated access method for *variable-length* queries, for which standard indexing is insufficient (Section 3.3).
4. A discussion of practical issues relating to our three access methods, including predicate evaluation and a comparison of physical disk layouts (Section 3.4).

In Section 4 we demonstrate the efficiency of our four access methods on both real and synthetic data. We show on real data that Caldera's access methods deliver speedups of up to two orders of magnitude over a naïve stream scan. We then use synthetic data to push beyond the limitations of our real dataset and confirm that our algorithms continue to provide similar speedups under a wider variety of stream conditions. Finally, we identify unique characteristics of location-based Markovian streams, and we discuss the impact of these characteristics upon access method performance, disk layout choice, and the quality of a heuristic-based, approximate access method.

2 Preliminaries

In this section we give an overview of the queries and data processed by Caldera (see Figure 1). We first sketch one of the many processes by which Markovian streams can be generated from sensor data. We then describe event queries over Markovian streams and outline the major challenges to processing them in an archived setting.



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 2: (a) A sample-based distribution of Bob's location at time 7 (a1) where antenna A detects the Bob, and the next timestep 8 (a2) where the Bob is undetected. (b) shows the marginal distributions computed from the samples, for times 7 (b1) and 8 (b2) respectively. (c) shows the correlations between these two timesteps.

2.1 Hidden Markov Models and Markovian Streams

A Markovian stream is a post-processed version of a noisy sensor stream, in which errors have been smoothed out and gaps between sensor readings have been filled. Several techniques exist for transforming sensor data into a Markovian stream. Here we present one such technique, which we also use to generate the Markovian streams in our evaluation.

Consider again the example scenario in which RFID readers track Bob's location as he moves throughout a building. Because RFID readers are placed at discrete locations and because of noise and interference, the raw RFID stream representing Bob's location contains errors and gaps. Often, however, a system can use probabilistic *inference* to reduce errors and provide location estimates during times for which no sensor data is available.

Many of the methods for inferring these location estimates rely upon a simple and commonly-used graphical model called a *Hidden Markov Model* (HMM) [29]. The HMM is designed to infer a sequence of hidden states (*e.g.*, Bob's locations) from a sequence of observations (*e.g.*, RFID tag reads). The HMM incorporates both *physical constraints* (*e.g.'a person cannot go directly from office O1 to office O2'*, since this would involve walking through a wall; *'a tag is most likely no more than 5 feet from the RFID antenna detecting it'*, since antenna ranges are small) as well as *statistical likelihoods* (*e.g.'it is more likely that Bob will enter his own office, rather than his neighbor's'*. The construction of this model is orthogonal to Caldera's contribution, so we do not describe it here, but refer the reader to an excellent tutorial [29].

Standard probabilistic inference algorithms—including the one used in our experiments—can use these models to infer information missing from raw sensor streams. Fig. 2 illustrates the key idea behind a popular and intuitive class of these algorithms called *sample-based* inference techniques. The figure depicts a small RFID deployment in which readers A, B, and C are located in the corridors of a building. The figure shows two timesteps' worth of *samples*, which are simply guesses about Bob's location. The samples move through space at each timestep and congregate in areas consistent with sensor readings received by the system. To compute the marginal probability that Bob's tag is in a particular location, *e.g.* $H1$, at a particular time, *e.g.* 7, the system simply counts the number of samples in $H1$ and divides by the total number of samples. In this example, 80% of Bob's samples are in location $H1$.

In addition to these marginal distributions, probabilistic smoothing algorithms also provide correlations *between* these distributions. Such correlations are called conditional probability tables, or CPTs. To demonstrate their importance, imagine that at two later timesteps, say 50 and 51, Bob is in either office $O1$ or $O2$, each with probability 0.5. Using correlations, the probability that Bob is in $O1$ at time 50 and $O2$ at time 51 is $0.5 * 0.0 = 0.0$. The second term in this product reflects the physical impossibility of moving directly between these two offices, and reduces the final query probability appropriately to zero. Ignoring correlations, however, the probability that Bob moves from $O1$ to $O2$ is $0.5 * 0.5 = 0.25$! While Bob's ability to walk through walls bodes well for his career as a superhero, perhaps more relevantly it highlights the importance of the correlations in Markovian streams. Not surprisingly, maintaining these correlations is known to increase the *quality* of events [33]. We use the name *Markovian stream* for the correlated, temporal data produced by probabilistic smoothing on a raw input stream.

A *Markovian stream* with value attributes $A_1, \ldots, A_k$ (where A_i has domain D_i for $i = 1, \ldots, k$) is a pair (p_0, C). Here p_0 is a distribution over $D_1 \times \cdots \times D_k$, representing the initial distribution of the stream; C is a sequence of *conditional probability tables* (CPTs) $\mathsf{C}(A_1, \ldots, A_k, A'_1, \ldots, A'_k, T, P)$. Each tuple in C uniquely describes the probability of a specific state transition from time t to $t+1$. For example, a tuple $(a_1, \ldots, a_k, a'_1, ..., a'_k, t, p)$ in C indicates that p is the conditional probability that the state of the stream is $(A_1 = a'_1, \ldots, A_k = a'_k)$ at time $t+1$, given that the state at time t is $(A_1 = a_1, \ldots, A_k = a_k)$. The set of tuples sharing a value of $T = t$ together define the entire stream transition (CPT) between t and $t+1$. For a full formal

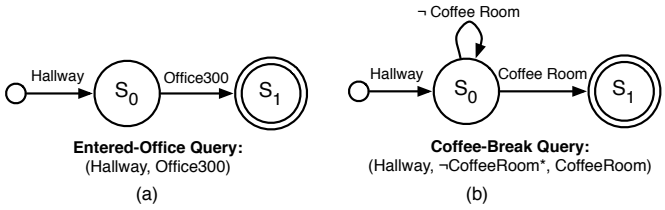


Figure 3: Two two-link Regular event queries in NFA form and written syntax. (a) Entered-Room query asking when Bob is in the Hallway and then immediately in Office300. (b) Coffee-Break query asking when Bob is in the Coffee Room at *any* time after leaving the Hallway.

semantic, see Ré *et al.* [33].

For performance reasons Caldera explicitly stores the marginal distributions of *each* timestep in addition to the CPTs, instead of only the distribution of the first timestep, p_0. Figures 2(b) and (c) show the relational representation of a Markovian stream in which the single attribute A_1 represents Bob's location.

2.2 Regular Queries on Markovian Streams

Streams lend themselves naturally to *event queries* [9], which ask questions about sequences of states. An example event query is, *"When did Bob enter his office (300)?"*. In this paper, we look at a subset of event queries called *Regular queries* [33]. These queries are analogous to regular expressions, or equivalently NFAs. An example Regular query is shown in Figure 3(a). This query is satisfied when Bob is first in a Hallway and then in Office300 in the next timestep. Regular queries comprise only *linear* NFAs, which means that the directed automaton edges form a total order.

Regular query NFAs differ from standard NFAs in that they transition when *predicates* on the input stream are satisfied, instead of on static alphabet symbols. For example, the NFA in Fig. 3(a) transitions from S_0 to S_1 on input timesteps in which Bob's location is the single location satisfying the Office300 predicate. If Bob is instead anywhere else, then no transition is possible and the automaton dies.

We define a *predicate* here as a Boolean function on stream attributes. For example, the CoffeeRoom predicate returns true when the input location is a Coffee Room. A *Regular query* comprises a list of such predicates organized in a way that respects the NFA's linearity. Concretely, a Regular 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.* (¬`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. Because Regular query NFAs are linear, a list of links uniquely determines a query. Fig. 3(a) and 3(b) show Regular query NFAs and their expression using this formalism.

A technical contribution of this work is to divide the set of Regular queries into two classes for optimization: (1) *fixed-length queries*, whose NFA representations are loop-free, and (2) *variable-length queries*, whose NFAs contain loops. Figs 3(a) and 3(b) show example fixed- and variable-length queries, respectively. The class of fixed-length queries is so-named because these queries can be satisfied only by fixed-length stream intervals (*i.e.* an n-link query can be satisfied only by stream intervals of length n). In contrast, intervals satisfying variable-length queries may span an arbitrary number of timesteps. As we demonstrate in the following sections, these distinctions are crucial for optimization.

The output of a Regular query is a sequence of pairs $\langle t, p \rangle$, where p is the probability that the query is satisfied at timestep t. Figure 4 graphically illustrates the output of the Entered-Room query detected in a real RFID data stream. The query probability spikes when Bob enters his office ($t \approx 1100$), and a much lower series of peaks around the false positives ($t \approx 1600$). Applications can use simple thresholding to detect these event peaks (*e.g.* Bob is entering an office if $p > 0.3$).

2.3 Archived Regular Query Challenges

In our previous work we presented Lahar, a system that can efficiently process *real-time* Regular queries. Lahar's algorithm requires as input an *entire* Markovian stream, making its use untenable in large-scale, archived scenarios where a large volume of stream data must be read from disk. In this section we outline the three major challenges addressed in our current work on efficient processing of *archived* Regular queries.

Our first challenge is that of *adapting classical techniques* (*e.g.* B+ tree indexing, merge joins, *etc.* [30, 44]) to the problem of processing large Markovian stream archives. This adaptation is non-trivial because only a subset of Regular queries can

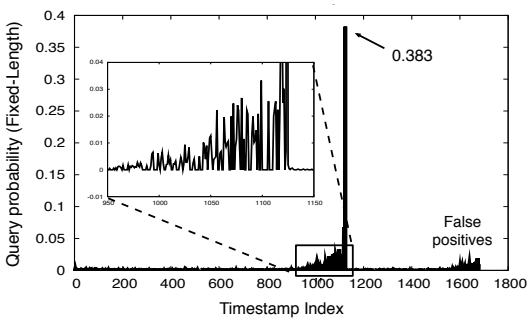


Figure 4: Probability of the Entered-Room query on a real RFID data stream. Bob actually entered his office only once, around timestep 1100, but he walked past the office door again around timestep 1600.

be processed using standard techniques. Thus the challenge is in identifying this class, and in mapping standard techniques onto the queries within it.

Our second challenge, *high-quality event retrieval*, arises from the abundance of low-quality events matching each query. The uncertainty in Markovian streams creates a large number of low-quality query matches for each high-quality peak in the query signal (see Figure 4). Top-k algorithms are a standard approach for dealing with such noise. The challenge in adapting top-k approaches to archived Markovian streams is that query match probabilities are a *function* of the data stream, but are not present directly in the stream data itself. As a result, effective top-k pruning conditions are non-obvious.

Our third challenge, *correlation retrieval*, arises from the correlations present in Markovian streams. Variable-length query matches can span intervals of arbitrary length, and the probabilities of these matches depend on correlations across the entire interval—computable only via a scan of all timesteps in the interval. The challenge in this case is to develop a data structure that makes effective use of precomputation to allow efficient lookup of correlations between *any* two timesteps, without requiring excessive storage space.

3 Access Methods for Markovian Streams

Caldera's query plans comprise three operators as illustrated in Figure 5(a). At the heart of each query plan is the `Reg` operator based on Lahar [33]. We treat `Reg` here as a black box that takes as input a Markovian stream and a Regular query, and produces as output the query probability at each timestep in the stream. The `Sort` operator is a standard operator, for use in top-k queries. It sorts the output of the `Reg` operator in order of decreasing probability. Finally, the `Ex` (extract) operator, which is the focus of this paper, extracts fragments of Markovian streams from disk using one of the access methods introduced shortly.

In this section, we present five physical implementations of `Ex`. The first, Algorithm 1, is a naïve stream scan. In the absence of indexes this is the only possible approach. This is the baseline against which we compare our optimized implementations.

Figure 5(b) outlines the space of Regular queries and the novel algorithms that we develop in this section. We first show how to leverage B+ tree indexes to efficiently answer fixed-length queries. We then show how to further leverage standard techniques to optimize top-k variants of these queries. For variable-length queries, we introduce a new type of index, called the Markov chain index (MC index) and an access method that uses this index to efficiently answer *any* Regular event queries.

We first introduce our algorithms under two simplifying assumptions: 1) we assume that all streams are defined on only a single attribute A_1; 2) we assume that all predicates are of the form `attribute op constant` (*e.g.* predicates select single attribute values). In Section 3.4 we show how our algorithms generalize beyond these assumptions.

3.1 Fixed-Length Regular Queries

As reflected by the first challenge in Section 2.3, one goal of this paper is to adapt standard indexing techniques (*e.g.* B+trees and merge joins) wherever possible in order to process Regular queries on Markovian streams. In the case of fixed-length queries, we are able to successfully adapt these classical techniques with only minor changes.

We first describe a B+ tree secondary index on a Markovian stream, defined in a straightforward manner. The B+ tree uses search keys of the form: `(attribute, time)`. Here `attribute` is the stream attribute being indexed and `time` is

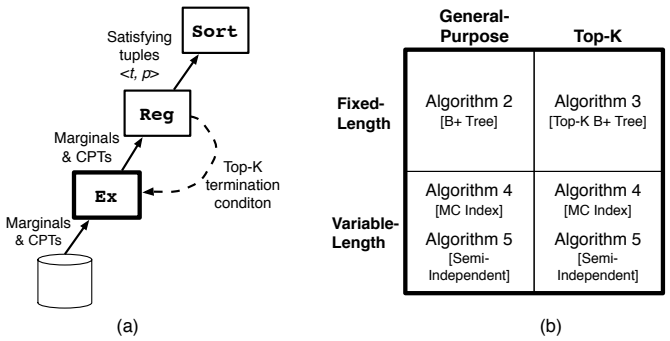


Figure 5: (a) A sample Caldera query plan. (b) A breakdown of the space of Regular event queries and the `Ex` implementations we develop for each quadrant.

Algorithm 1 Naïve Scan Access Method

- `Reg`.initialize(): Initializes the `Reg` operator with the first marginal in an interval.
- `Reg`.update(): Updates the `Reg` operator with the next conditional in an interval.
- `M`[i]: The i^{th} timestep in `M`.
- t.marginal: The marginal distribution at timestep t.
- t.cpt: The conditional distribution (CPT) at timestep t.

Input: Regular query Q_R comprising n links, Markovian stream `M`
Output: Probability that Q_R is satisfied at each timestep $t \in$ `M`
1: `Reg`.initialize(`M`[0].marginal, Q_R)
2: **foreach** timestep t in `M` **do** p = `Reg`.update(t.cpt)

the stream timestamp. For example, a query plan for the Entered-Office query in Figure 3(a) might leverage an index on `(location,time)`. 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 6.

Note that, because each timestep in a Markovian stream represents a distribution over states, each timestamp can appear multiple times in this index. For example, timestamp 7 in Figure 6 appears in three index keys, with locations H2, O1, and H1 (not shown).

Recall now that a fixed-length query Q_F comprising n links can be satisfied only by stream intervals of length n. 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. However, a standard equijoin between the resulting sets of timesteps is useless for identifying relevant intervals. Instead, we require a temporally-aware join that identifies contiguous *sequences* of timesteps. We implement such a join 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. We say that the cursors *intersect* when they together reference a sequence of n consecutive timesteps, in the appropriate ordering. Thus every *intersection* identifies a length-n 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.

An access method which we call the B+Tree approach is shown in Algorithm 2. Lines 1-2 initialize index cursors on 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. In the case where one or more predicates are not indexed, the definition of *intersection* is relaxed accordingly, but the algorithm is otherwise unchanged.

An example of the pruning done by the B+Tree approach can be seen on the stream segment pictured in Figure 6. On a two-link query $(H2, O2)$, the cursors on H2 and O2 first intersect on the interval (t_7, t_8), which is passed to `Reg`. In contrast, the H2 index entry for time t_8 has no intersecting entry in the O2 index at time t_9, so this interval is skipped.

One convenient feature of our 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 query $(O1, H2)$ on the data in Figure 6. The B+Tree algorithm will identify both (t_7, t_8) and (t_8, t_9) as requiring further processing.

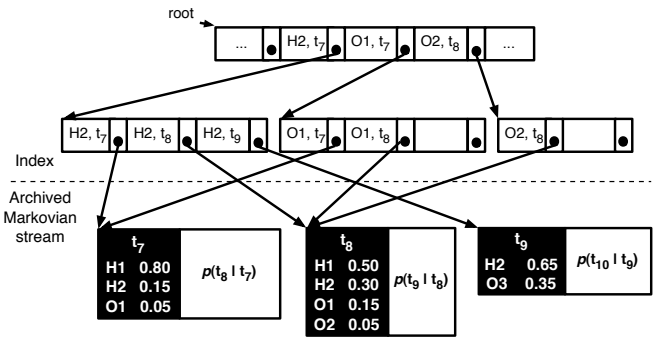


Figure 6: Bottom: Three timesteps' worth of the Markovian stream representing Bob's location, also shown in Figure 2. Top: A portion of BT_C on this stream.

Algorithm 2 General-Purpose, Fixed-Length Query Access Method

Note: For simplicity, merging of overlapping intervals is not shown.

Input: Fixed-length query Q_F comprising n links, 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_i in Q_F **do**
2: Initialize cursor C_i on predicate r_i in BT_C
3: **for each** interval I in the *intersection* of $(C_0, ..., C_n)$ **do**
4: `Reg`.initialize(I[0].marginal, Q_F)
5: **foreach** t in I **do** `Reg`.update(t.cpt)

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 degenerates into a naïve stream can, with additional overhead to handle the BT_C lookups.

3.2 Top-K Optimization for Fixed-Length Regular Queries

The B+Tree algorithm efficiently computes the probability of every query match in a stream; however, recall from Section 2.3 (Figure 4) 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-n interval, the marginal probability that the i^{th} link predicate is satisfied at the i^{th} timestep in the interval is an upper bound on the probability that the interval matches the query. As an example, consider the query $(H1, H2)$ on the interval (t_7, t_8) of Bob's location stream in Figure 2(b). Both $p(H1 \text{ at } t_7) = 0.8$ and $p(H2 \text{ at } t_8) = 0.3$ 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 an event query cannot be *more* likely than any of its individual components.

This observation implies that we can adapt the well-known Threshold Algorithm (TA) and its variants [11, 26] to our problem. The basic idea of our algorithm is to process fixed-length stream intervals in decreasing order of marginal probability. For each interval, we use the maximum marginal probability among all query predicates to determine the order.

More specifically, we introduce an additional secondary B+ tree index on the Markovian stream. We call this index the BT_P (p for "probability") index. It uses search keys of the form `(attribute,prob)`, where `attribute` is the stream attribute being indexed and `prob` is the marginal probability of all tuples satisfying the attribute value at the indexed timestep. Within BT_P keys sharing an attribute value, entries are ordered in *decreasing* order of marginal probability.

Algorithm 3 outlines an access method (the top-k B+Tree approach) that leverages the BT_P index and the TA technique to achieve efficient processing of 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 t_j 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 t_j) is below the probability of the best k query matches retrieved thus far (lines 5-6). If this condition is not met, the marginal probability of *each* predicate in the stream is examined (lines 8-9). If none of these is low enough (below the current top k match probabilities) to prune the interval, the interval is processed through the `Reg` operator (lines 10-11)

Algorithm 3 Top-K, Fixed-Length Query Access Method

Input: Fixed-length query Q_F comprising n links, Markovian stream `M`, B+ Tree index BT_P on the single attribute of `M`, k
Output: Top k timesteps at which Q_F is satisfied in `M`.

```
1: bestMatches.initializeEmpty
2: for each predicate r_i in Q_F do
3:    Initialize cursor C_i on predicate r_i in BT_P
4: for each timestep t_j referenced by any C_i do
5:    if t_j.marginal.probabilityOf(r_i)<= bestMatches.min then
6:       Terminate
7:    else
8:       I = [t_j, ..., t_{j+n}]
9:       if I[k].marginal.probabilityOf(r_k)> 0, ∀ 0 ≤ k < n then
10:         Reg.initialize(I[0].marginal, Q_F)
11:         foreach t_k in I do Reg.update(t_k.cpt)
12:         bestMatches.evaluate(p)
```

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-n 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` 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 B+Tree approach on queries with clear peaks, such as that shown in Figure 3(b). We further explore the relationship between these two algorithms in Section 4.

3.3 Variable-Length Regular Queries

The fixed-length access methods in the previous section are inapplicable to variable-length queries, because variable-length queries can match stream intervals of arbitrary length. A full stream scan seems necessary in this case, but can be avoided 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. match at least one query attribute). 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 each relevant timestep to the next. Thus we arrive at the final, *correlation retrieval* challenge: to develop an efficient (in both space and time) method for retrieving the correlations between distant stream timesteps.

We now introduce a novel indexing structure that provides efficient access to CPTs relating *any* pair of timesteps in a stream. We call this structure the Markov chain Index (*MC* index), and we develop an `Ex` implementation that leverages it to efficiently process variable-length queries.

3.3.1 Markov chain Index

The Markov chain (*MC*) index is a tree-structured index that provides efficient lookup and/or computation of the CPT relating any two Markovian stream timesteps. The index stores a small set of precomputed CPTs organized in the tree structure shown in Figure 7. The lowest level of the tree ($i = 0$) is simply the set of all M CPTs in the raw, 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 7 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 7, the two shaded index entries represent those whose product is the CPT relating t_0 and t_5. In the absence of the *MC* index, the entire raw stream would require scanning.

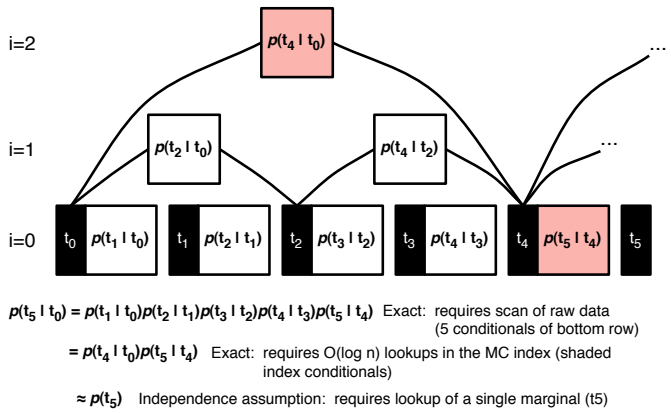


Figure 7: Markov chain index for $\alpha = 2$. The bottom index row ($i = 0$) is the raw data itself, abstracted here to hide numerical details. The two index entries required to compute the CPT between timesteps t_0 and t_5 are shaded; without the index, a scan of the raw data would be required. The semi-independence assumption requires lookup only of the marginal at time t_5, but computes an approximate result.

Algorithm 4 Variable-Length Query Access Method (MC Index)

Input: Variable-length query Q_V comprising n links, 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_i in Q_V **do**
3: Initialize cursor C_i on predicate r_i in BT_C
4: **for each** timestep t referenced by any C_i **do**
5: **if** !Reg.isInitialized **then**
6: p = Reg.initialize(t.marginal, Q_V)
7: **else**
8: cpt = MC.computeCPT(t_{prev}, t)
9: p = Reg.update(cpt)
10: $t_{prev} = t$

3.3.2 Variable-Length Algorithm

The Markov chain index naturally yields a simple algorithm (Algorithm 4, which we call the MC index approach) for variable-length queries.

In lines 2-3, a separate cursor is initialized on each query predicate. Line 4 advances these cursors forward in parallel, entering into the loop once for each timestep t satisfying *any* of the predicates. Upon the first entrance into this loop, execution jumps to line 6 where the `Reg` operator is initialized. Upon subsequent iterations,line 8 uses the MC index to compute the CPT between the previous relevant timestep (t_{prev}) and the current one (t). In line 7 this CPT is used to update the `Reg` operator. In this way the algorithm performs a single pass over the entire stream, using the MC index to efficiently summarize and process spans of irrelevant data.

When a variable-length query contains non-negated loop predicates (*e.g.* $(H2, O2*, O2)$, the MC index and algorithm as presented above appear 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 ($O2$ 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.* $O2$) 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 extremely similar to those presented for the general case, and we omit them due to space limitations.

3.4 Practical Details

Up to this point we have abstracted away several details of our access methods, which we now discuss. These include handling streams with multiple value attributes, more sophisticated predicates, and disk layout choices. After this discussion we close our algorithmic section with the presentation of a final, heuristic-based *approximate* access method for variable-length Regular queries for which the performance of the algorithms presented thus far is insufficient. This final algorithm provides no accuracy guarantees, but we show in Section 4 that in practice it can produce good probability estimates.

3.4.1 Indexing Predicates

Thus far we have assumed that all query predicates are expressed as trivial equality selections on a single stream attribute. However, Caldera can also support more sophisticated predicates. Caldera is designed to work with a star schema, analogously to data warehouses: the Markovian stream corresponds to the facts table and can have multiple value attributes. Additional dimensions tables provide extra information about stream attributes. Consider as an example the RFID-derived Markovian stream with schema `(tagID,locationID,time,probability)`. Additional information about individual locationIDs could be stored in a dimension table called LocationType, with schema `(locationID,locationType)` and tuples of the form ("Room300", "CoffeeRoom"), ("HallwaySegment3", "Hallway"), *etc.*.

As in a data warehouse, Caldera supports join indexes between the stream relation and the dimensions relations. These indexes speed up queries that join the stream relation M with a dimension table D and perform a selection on an attribute D.a of D (*e.g.*, "When did Bob go from a hallway to a coffee room?"). Conceptually, the join index extends the tuples in M with the attributes of D and indexes the resulting relation. In practice, we create the index directly on M without modifying it. Within our access methods, two types of join indexes are particularly useful for event queries: indexes with search keys of the form `(D.a,M.time)` and `(D.a,M.prob)`.

When the probability of a predicate P_i at a particular timestep t is required (*e.g.*, to create the index entries of BT_P for time t), it is computed by summing the marginal probabilities of all tuples whose attribute values satisfy P_i at t; this construction is valid because tuples with the same timestamp are disjoint. Conditional predicate probabilities are similarly combined via weighted averages.

In the case where a Markovian stream is defined over multiple attributes, it is possible that indexes exist on only a subset of these attributes. Our fixed-length access methods can simply use whatever indexes are available. Our variable-length access method, however, requires indexes on *all* attributes involved in any of the query predicates. If any of these indexes is unavailable, a naïve full data scan is the only processing option supported by Caldera.

Finally, with the exception of the top-k algorithm, all access methods support both equality and range predicates.

3.4.2 Physical Schema

In this section, we discuss the details of physically storing Markovian streams on disk.

Because a Regular query is defined over a single Markovian stream (*e.g.* Bob's location or Sue's location but not both at the same time), the most sensible high-level partitioning on disk is by stream. This ensures that all data relevant to a given query is localized to a single partition. Within a partition (Markovian stream), we choose to order data chronologically since the `Reg` operator needs to process data in that order; however, several disk layout fulfill this constraint. Recall from Section 2.1 that a Markovian stream is defined by a series of marginal and conditional distributions. There are at least two possible alternatives for storing these two types of information:

Separated Layout: The marginal and conditional sequences are each laid out chronologically, but in separate locations (*e.g.* separate files). This layout optimizes for situations where most accesses are made to one sequence or the other, but not to both. Indexes on this layout are constructed separately for each sequence.

Co-Clustered Layout: The marginal and conditional sequences are interleaved such that the marginal and conditional tuples representing a single timestep are placed together on disk, immediately before the marginal/conditional pair representing the chronologically-next timestep. This minimizes access cost for operations requiring access to both the marginal and conditional data for most timesteps. Indexes on this layout are constructed as shown in Figure 6.

While the optimal layout is workload-dependent, we find in Section 4 that for common queries on a Markovian stream derived from RFID data, the separated layout yields superior performance.

Algorithm 5 Approximate Variable-Length Access Method (Semi-Independent Algorithm)

Input: Variable-length query Q_V comprising n links, Markovian stream `M`, B+ Tree index BT_C on **ALL** attributes of Q_V
Output: **Approximate** probability that Q_v is satisfied at each timestep $t \in$ `M`

```
1:  t_prev = Ø;
2:  for each predicate r_i in Q_V do
3:      Initialize cursor C_i on predicate r_i in BT_C
4:  for each timestep t referenced by any C_i do
5:      if !Reg.isInitialized then
6:          p = Reg.initialize(t.marginal, Q_V)
7:      else
8:          if t.index == t_prev.index + 1 then
9:              p = Reg.update(t.cpt)
10:         else if t.index > t_prev.index + 1 then
11:             p = Reg.update(t.marginal)
12:     t_prev = t
```

3.4.3 Variable-Length Algorithm (Approximate)

We close our algorithmic section with the acknowledgement that, while our MC index algorithm provides applications with an elegant space/time tradeoff,some disk-hungry or time-critical applications may not be satisfied by any point in this tradeoff space. For these applications we provide a final access method that is both more efficient than the semi-independent approach, and does not require use of the *MC* index; however, it returns only *approximate* results. We call this algorithm the semi-independent approach because it assumes independence between some—but not all—timesteps in a Markovian stream.

Algorithm 5 outlines the semi-independent approach. Lines 1-3 initialize a separate cursor for each query attribute. In line 4, the cursors are advanced in parallel and the loop is entered once for each timestep satisfying *any* query attribute. Upon the first entrance into this loop, execution continues to line 6 where the `Reg` operator is initialized. Upon subsequent iterations, a second test is applied (line 8): if t is adjacent in the input stream to the previous timestep t_{prev}, then `Reg` is updated with the correlations between the two timesteps, read directly from the raw stream data (line 9). Otherwise (line 10), the correlations are approximated by an independence assumption (line 11). A pointer to the previous timestep is maintained throughout (line 12).

We call this approach semi-independent instead of simply independent because it does *not* assume independence between adjacent timesteps. The CPTs relating adjacent timesteps can be read using a single disk access, equivalent in cost to the access of the marginal required to make the independence assumption. The costs of semi-independence and full independence are therefore the same, and we choose the algorithm that preserves as much correlation information as possible.

4 Evaluation

In this section, we evaluate Caldera's access method strategies for event queries on Markovian streams. We implemented Caldera in Java, using Berkeley DB [27] as our data storage back-end. All experiments were conducted on a 2.00GHz Linux machine with 16GB of RAM.

We demonstrate on both synthetic and real data that standard B+ tree indexing techniques (the B+Tree [Algorithm 2] and top-k B+Tree [Algorithm 3]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. We further demonstrate that the novel Markov chain index provides the same magnitudes of speedup on variable-length queries. Finally we demonstrate that the approximate semi-independent approach outperforms even the MC index approach, and we elaborate on its tradeoff between efficiency and accuracy.

4.1 Setup

We evaluate Caldera on both synthetic and real Markovian streams. In both cases, the data domain is the RFID-based location tracking application used as our primary motivation throughout the paper.

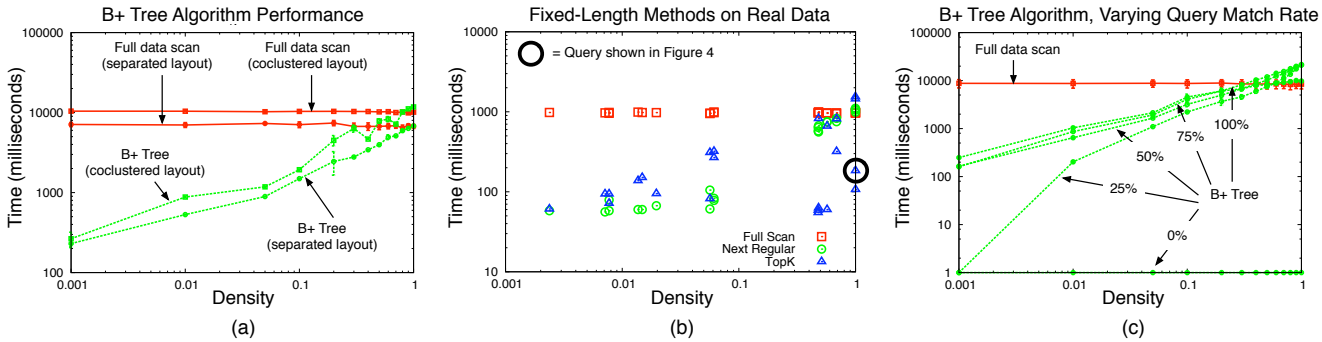


Figure 8: Evaluation of access methods optimized for fixed-length queries. (a) Worst-case performance of the B+Tree algorithm using two disk layouts, on synthetic data. (b) Performance of fixed-length approaches on real-world data. (c) Performance of the B+Tree algorithm on increasingly-favorable datasets.

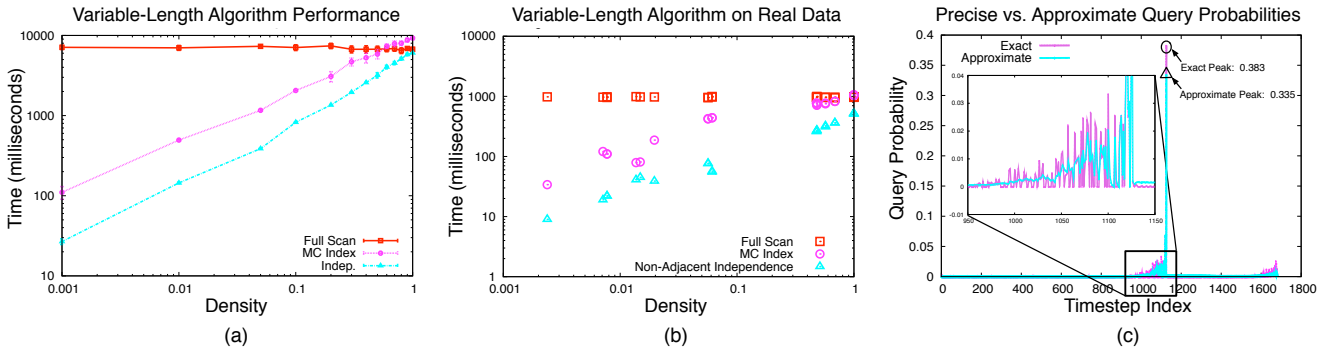


Figure 9: Evaluation of access methods optimized for variable-length queries. (a) Performance of the three algorithms as the number of relevant timesteps varies. (b) Performance on a real-world stream. (c) Breakdown on a real-world query of the approximation error of the semi-independent approach.

4.1.1 Synthetic RFID Data

We evaluate basic performance of our algorithms on a synthetic set of Markovian streams. Each stream comprises 8 hours of data (30,000 timesteps). To maintain realistic properties in these streams, we constructed them by concatenating together various 30-second stream "snippets" generated from an RFID simulator reflecting the physical layout of our building and RFID deployment. 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 control the relevant properties of each stream with respect to our test queries.

4.1.2 Real RFID Data

Our real RFID dataset was collected using our building-wide RFID deployment by eight volunteers carrying 58 tags as they went through one-hour versions of typical daily routines. These routines visited rooms across two floors, spanning an area of roughly 10,000 square feet discretized into 352 locations. The 38 antennas on these two floors were placed only in the corridors (antennas were not placed in any room/office/lab/etc.). We show performance on eight of these traces, selected for their length and realistic reflection of location uncertainty (*i.e.*, some tags were never or almost never detected and we did not use the resulting traces).

We find that, on our real RFID dataset, the fraction of timesteps containing data relevant to any given query 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 method.

4.2 Access Methods for Fixed-Length Regular Queries

We first evaluate the performance of the B+Tree algorithm (Algorithm 2). In this experiment, we evaluate performance on a two-link Entered-Room query and compare the results to the naïve full stream scan. We perform this comparison using both the separated and co-clustered disk layouts in order to assess the benefits of each type of data organization.

4.2.1 Performance Overview

Logscale performance numbers for both algorithms running on our synthetic data are shown in Figure 8(a). We note that the B+Tree algorithm performance here is worst-case, because in this specific synthetic data trace, *each* relevant timestep 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. As expected, when the data density is low, even in the worst case the B+Tree algorithm significantly outperforms a full stream scan—in this case, by 1-2 orders of magnitude. Conversely when most stream timesteps are relevant to the query, the B+Tree approach degenerates into a full data scan with additional overhead for the B+ tree lookups.

Figure 8(a) also demonstrates that both a full stream scan and the B+Tree algorithm perform more efficiently on streams archived using a separated disk layout. In the case of a full scan this is not surprising, since only CPTs (no marginals) are required. More interesting is the superior performance of the B+Tree approach on a separated layout. In theory, when data density is low, relevant stream intervals should be isolated and the B+Tree approach will process roughly as many marginals as CPTs, within a small constant factor. A co-clustered layout will certainly outperform a separated layout in this case, since only a single disk I/O is required to access both marginal and CPT data. In real-world Markovian streams, however, and in our synthetic traces, intervals of relevant data are not isolated but instead overlap each other in long (*e.g.*, 30 timesteps or more) stretches. This overlap increases the number of CPTs read from disk for each marginal, and as this number increases, the separated layout dominates performance.

In order to show the best performance, we evaluate all algorithms throughout the remainder of this section using a separated disk layout. We note however that other types of Markovian streams may not exhibit the same locality of relevant timesteps, and for these streams the choice of optimal disk layout may be different.

4.2.2 Real-World Query Performance (Fixed-Length)

Figure 8(b) shows the performance of the three fixed-length query algorithms on a set of 22 different Entered-Room queries in one real, 28-minute stream. Each of the 22 queries is responsible for three plotted points (one for each algorithm). Not surprisingly, the plot confirms results in Fig. 8(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 8(b) reflects the bimodality of data density.

In the set of queries with low data density, the top-k B+Tree approach (Algorithm 3, 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. Additionally, 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 3(b) exhibits such behavior, and indeed this query is responsible for the top-k B+Tree point highlighted in Fig. 8(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 large number of relevant timesteps.

4.2.3 Broader Performance Trends

Figure 8(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

		Stream: James Q: Entered-Office # Subgoals in query:			Stream: Sally Q: Entered-Office # Subgoals in query:			Stream: Pat Q: Coffee-Room # Subgoals 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
NEXT	# query matches	26	26	11	2	1	1	3	1	1
NEXT	Time: B+ Tree (msec)	174	232	266	59	67	132	67	71	135
NEXT	Time: Top-K B+ Tree (msec)	56	75	214	67	98	165	95	366	1970
BEFORE	# query matches	145	188	188	2	1	1	3	21	21
BEFORE	Time: MC Index (msec)	290	424	678	32	43	117	187	237	902
BEFORE	Time: Semi-Indep. (msec)	114	156	379	9	12	22	39	68	398

Figure 10: Algorithm and stream statistics from our real data set.

query matches. The curve for a query match rate of 100% is precisely the curve from Figure 8(a).

For any single curve in Figure 8(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 similarly causes a proportional increase in performance. 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.

4.2.4 Performance on Longer Queries

We demonstrate the scalability of the B+Tree algorithm on queries comprising more than two links using the real data. Figure 10 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 (we do not expect real-world queries to require more than 4 links). Queries with 3 and 4 links require a tag's presence at multiple specific hallway locations outside a room before the room is entered.

The Reg operator slows exponentially with each additional query link, as can be seen in the fourth row of Fig. 10. Because the B+Tree approach is able to avoid many of these 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 indexes 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 indexes, independent of the number of predicates indexed. A second-order effect reflects improved performance when the number of predicate indexes decreases, simply because the index overhead is reduced.

4.3 Access Methods for Variable-Length Queries

We first evaluate the performance of the MC index (Algorithm 4, α=2) and semi-independent (Algorithm 5) approaches versus a naïve stream scan, on synthetic data and the now-familiar Entered-Room query. Figure 9(a) shows the results. Data density is again the dominant factor in the performance of both algorithms, which exhibit the same trends as the B+Tree approach. Figures 9(a) and 8(a) are directly comparable. The approximate, semi-independent approach is consistently faster than the precise MC index approach, by a factor of roughly 8.

4.3.1 Real-World Query Performance (Variable-Length)

The performance of the three variable-length query algorithms on real-world data is shown in Figure 9(b). As on the synthetic data in Figure 9(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. The semi-independent approach continues to show performance improvements over the MC index algorithm measuring just under an order of magnitude.

The queries shown in Figure 9(b) are precisely the queries plotted in Figure 8(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). The two algorithms perform very similarly, yielding an interesting discussion about when one or the other might be preferable. The MC index approach is more general in that it can handle any kind of Regular query; however, it is applicable only when all stream attributes are indexed, and when the *MC* index is available. If either of these conditions are not met, then the B+Tree can be applied, but only to fixed-length queries.

4.3.2 Accuracy vs. Efficiency

The semi-independent approach is faster than MC index, but this performance gain comes at the cost of accuracy. The approximate, semi-independent algorithm makes no guarantees about the magnitude of the errors it may incur. Furthermore, recall from Section 2 that ignoring correlations as the semi-independent approach does can have a large impact on the resulting query results. Thus any application interested in precise results must avoid the approximate approach.

Many applications, however, are more flexible and can imprecision. For such applications, the tradeoff between efficiency and accuracy depends on the types of approximation errors typically present in the Markovian streams of the application domain. An example of such errors on a real-world Markovian stream is shown in Figure 9(c). In this case, the approximate query probabilities track the magnitudes of the precise probabilities fairly well. The semi-independent algorithm correctly identifies the maximum-probability timestep (this is not guaranteed or even likely in some datasets), incurring a relative error of roughly 13%. Applications in many circumstances may be willing to use these approximate results in return for an order-of-magnitude speedup.

Anecdotally, we find the approximations in Figure 9(c) to be a fairly optimistic sampling of the output of the semi-independent algorithm. The semi-independent approach on other of our real-world Markovian streams failed to properly identify the maximum-probability timestep, giving errors with raw magnitudes up to 0.286.

4.4 Markov Chain Index

The purpose of the Markov chain index is to provide efficient access to (precomputed) correlations between distant Markovian stream timesteps. Figure 11 provides details of the index's tradeoff between storage space and lookup speed.

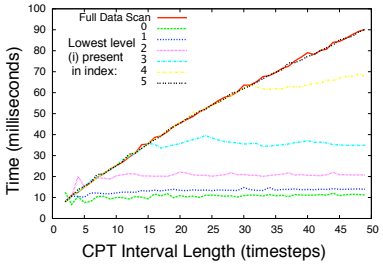
Figure 11(a) shows the average time required to compute correlations between two timesteps separated by intervals of varying size. Because the placement of these intervals relative to stored index entries is important, reported results are the averages over all placements. Each higher curve in the graph represents performance when an additional index level is removed (this is a proxy for 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 indexes 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, we 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 Caldera will find this an acceptable tradeoff; however, disk-starved applications can leverage the MC index within their resource limitations by incresaing α. Storage requirements for various α on streams of varying lengths are shown in 11(b).

5 Related Work

Management of large-scale archived RFID data sets has received significant attention, with much work focusing directly on the storage, processing, and analysis of raw, noisy RFID data [14, 16, 24, 31, 39]. A complementary body of work addresses the cleanup of noisy RFID streams and presents methods for producing cleaned—but still deterministic—streams [19, 31, 39]. By contrast, Caldera preserves the uncertainty (probabilities and correlations) at the levels of both data and queries. In previous work, we proposed probabilistic RFID cleaning [21] and event extraction [41] techniques. These approaches, however, ignore correlations, do not support Kleene Closures, and extract events by executing incremental SQL queries.

The problem of probabilistically inferring high-level information (e.g. location, activity) from low-level sensor data has a long history in the AI community. Temporal probabilistic models such as Hidden Markov Models [29] and Dynamic Bayesian Networks [23, 25] deal specifically with uncertain, temporal data. Particle filtering [2] (and associated smoothing algorithms [22, 7]) is a simple and popular way of performing inference in these models, and has also recently been used in



(a)

α	# 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)
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 11: (a) Time required to compute 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.

the databases community to perform real-time inference inside a DBMS [20] and to infer the locations of RFID tags when both tag and antenna locations are unknown [37].

While inference in probabilistic models is highly developed in the AI community, work on answering relational and event-style queries on the output of these models is in its infancy. Deshpande *et al.*, propose treatment of these models within a DBMS as a model-based view [10] and explore answering selection and aggregation queries on these views [20]; Caldera in effect treats its stream archive as a materialization of such a view and explores event-based queries. The Data Furnace project [12] similarly proposes to perform model-based inference directly inside a DBMS; however to the best of our knowledge no technical results on this project have been published. Probabilistic databases like MystiQ [8] and Trio [3, 42] address a related problem space but cannot model the sequential, correlated data streams at the heart of Caldera.

Automaton-based processing of event queries on RFID or similar data streams has recently been explored in the Cayuga [9] and SASE [43] systems, but these systems assume a deterministic input stream. The Lahar [33] system adapts this style of processing to probabilistic input. We use the Lahar approach as a component of our current work.

Markovian streams are related to time series and time evolving data, since they are data sequences with time information. Caldera's workload (*i.e.*, event queries on probabilistic streams), however, differs significantly from the workload of time-series databases (*i.e.*, primarily similarity searching) and temporal databases (*i.e.*, selections of different object versions or objects valid within certain time intervals). Indexing techniques for time series [15] and temporal [35] data are thus orthogonal to Caldera's techniques.

There has been recent work on top-k query processing of probabilistic relational systems[32, 36, 4]; however, none consider the problem of indexing streaming sequential data. Separately, there is work on probabilistic streams [18, 6], but this work does not consider access paths for the streams.

6 Conclusion

In this paper we presented Caldera, a system for executing event queries over stored Markovian streams. In contrast to previous systems that must process the entire Markovian stream to answer an event query, Caldera selectively processes only relevant parts of the stream, thus achieving significant performance gains. At the same time, Caldera preserves result accuracy by retaining the Markovian properties of the stream while skipping data.

To achieve high performance, Caldera 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, Caldera uses novel adaptations of standard B+ tree indexes. For variable-length queries, it leverages a new type of index, the MC index, to effectively summarize unimportant parts of a stream. Additionally, Caldera 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 Caldera offers performance that can be orders magnitude better than 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.

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proc. of the SIGMOD Conf.*, June 2008.

[2] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for on-line non-linear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, Feb. 2002.

[3] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. Uldbs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964, 2006.

[4] D. Burdick, P. M. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. Olap over uncertain and imprecise data. *The VLDB Journal*, 16(1):123–144, 2007.

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

[6] G. Cormode and M. N. Garofalakis. Sketching probabilistic data streams. In *SIGMOD Conference*, pages 281–292, 2007.

[7] R. G. Cowell, S. L. Lauritzen, A. P. David, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[8] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Prov. of the 30th VLDB Conf*, 2004.

[9] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. of 10th EDBT Conf.*, Mar. 2006.

[10] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *Proc. of the SIGMOD Conf.*, June 2006.

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

[12] Garofalakis et. al. Probabilistic data management for pervasive computing: The Data Furnace project. *IEEE Data Engineering Bulletin*, 29(1), Mar. 2006.

[13] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, 2003.

[14] H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and analyzing massive RFID data sets. In *Proc. of the 22nd ICDE Conf.*, Apr. 2006.

[15] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., second edition, 2005.

[16] 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.*, Sept. 2005.

[17] Incorporated Research Institution for Seismology (IRIS). Seismic Monitor. `http://www.iris.edu/hq/`.

[18] T. S. Jayram, A. McGregor, S. Muthukrishnan, and E. Vee. Estimating statistical aggregates on probabilistic data streams. In *PODS*, pages 243–252, 2007.

[19] Jeffery et al. Adaptive cleaning for RFID data streams. In *Proc. of the 32nd VLDB Conf.*, Sept. 2006.

[20] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. In *Proc. of the 24th ICDE Conf.*, June 2008.

[21] N. Khoussainova, M. Balazinska, and D. Suciu. Towards correcting input data errors probabilistically using integrity constraints. In *Proc. of Fifth MobiDE Workshop*, June 2006.

[22] M. Klaas, M. Briers, N. de Freitas, A. Doucet, S. Maskell, and D. Lang. Fast particle smoothing: if I had a million particles. In *Proc. of the 23rd ICML*, pages 481–488, New York, NY, USA, 2006. ACM.

[23] S. L. Lauritzen. *Graphical Models*. Number 17 in Oxford Statistical Science Series. Clarendon Press, Oxford, 1996.

[24] 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.*, June 2008.

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

[26] A. Marian. Evaluating top-k queries over web-accessible databases. In *Proc. of the 18th ICDE Conf.*, 2002.

[27] 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.

[28] M. Philipose, K. P. Fishkin, M. Perkowitz, D. J. Patterson, D. Fox, H. Kautz, and D. Hahnel. Inferring activities from interactions with objects. *IEEE Pervasive Computing*, 3(4):50–57, 2004.

[29] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. pages 267–296, 1990.

[30] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Science Engineering, third edition, 2002.

[31] J. Rao, S. Doraiswamy, H. Thakkar, and L. S. Colby. A deferred cleansing method for RFID data analytics. In *Proc. of the 32nd VLDB Conf.*, Sept. 2006.

[32] C. Re, N. Dalvi, and D. Suciu. Efficient Top-k query evaluation on probabilistic data. In *ICDE*, 2007.

[33] C. Ré, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *Proc. of the SIGMOD Conf.*, June 2008.

[34] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[35] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, 1999.

[36] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *ICDE*, pages 896–905, 2007.

[37] T. Tran, C. Sutton, R. Cocci, Y. Nie, Y. Diao, and P. Shenoy. Probabilistic inference over RFID streams in mobile environments. Technical Report 07-59, University of Massachusetts at Amherst, 2007.

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

[39] F. Wang and P. Liu. Temporal management of RFID data. In *Proc. of the 31st VLDB Conf.*, Sept. 2005.

[40] E. Welbourne, M. Balazinska, G. Borriello, and W. Brunette. Challenges for pervasive RFID-based infrastructures. In *IEEE PERTEC 2007 Workshop*, Mar. 2007.

[41] 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 *Proc. of the Sixth MobiSys Conf.*, June 2008.

[42] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc of the 2nd CIDR Conf.*, 2005.

[43] E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing over Streams. In *Proc. of the SIGMOD Conf.*, June 2006.

[44] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann, 1997.