

Run-Length Encoding Markovian Streams

Jennifer Wong ^{#1}, Julie Letchner ^{#2}, Magdalena Balazinska ^{#3}

[#]*Computer Science & Engineering Department, University of Washington
Seattle, Washington, USA*

{¹jkwong, ²letchner, ³magda}@cs.washington.edu

Abstract—Markovian streams, a common class of imprecise data streams, have proved to be excellent models for uncertain, sequential data found in sensor readings. Radio Frequency Identification (RFID) information particularly lends itself to a Markovian model; a Markovian stream derived from one RFID trace represents all paths that an RFID tag traveled during an interval of time. Processing these types of streams, however, poses several challenges: the amount of disk space these streams use and the time it takes to process them. In this paper, we address both these challenges through run-length encoding (RLE) of Markovian streams. We introduce our algorithms for stream RLE compression and decompression, and study the effects on storage and timing efficiency and query error through experiments on real RFID-derived Markovian streams. The results of our tests prove not only that these streams are highly compressible, but also that the effects of this method of compression are ideal. RLE compression significantly reduces streams’ file size and marginally improves query processing time, all while managing the resulting query error.

I. INTRODUCTION

From audio recordings to sensor traces, people and computers record exabytes of sequential data everyday. Many different applications benefit from access to archives of this data; location tracking in business or hospital environments, activity monitoring, and web-based audio search are just several examples of such applications [5]. But, few applications are capable of using audio and sensor data directly. Search engines, for instance, cannot use radio podcasts until the audio data is translated into sentences. Many applications must therefore rely on higher-level information computed from the raw data.

The process of extracting higher-level information from low-level data, such as writing transcripts for audio speeches, is called *inference*. A significant consequence of this process is information that is *imprecise*. For example, due to the recording method or the process of inference itself, it might be difficult for speech recognition systems to infer whether “are paired” or “our pears” was spoken. Instead, those applications accommodate this imprecision by returning several guesses for what was actually said, each with a different probability. These uncertain sequences are commonly represented as *Markovian streams*, which are the focus of our paper. A search engine, for instance, would be able to use an audio-derived Markovian stream containing sentences inferred from a recording to retrieve the the probability that either “are paired” or “our pears” were spoken.

Though Markovian streams are important models of sequential data, using these streams presents two challenges. Depending on their length and complexity, Markovian streams are potentially expensive in terms of *disk space* and *processing time*. For instance, one Markovian stream representing just ten minutes of sensor data used nearly 11MB. As a solution to these inefficiencies, we present in this paper *run-length encoding* (RLE) of Markovian streams. Through the compression of redundant data, we seek to improve the storage and querying efficiency of these streams.

For this study, we worked with RFID-derived Markovian streams. Such streams contain the possible paths that an RFID tag traveled, inferred from the tag’s time and location records. RFID-derived Markovian streams are ideal for RLE compression because of several factors:

- These streams are potentially very long. If a stream infers an RFID tag’s location for every second, 10 minutes of recording will result in 600 items of data; a full day’s worth would include 86,000 pieces of data.
- RFID data is often highly redundant. For example, an RFID tag attached to an office worker will likely not move for extended periods of time. The result of the worker sitting in a meeting for 45 minutes or using a computer for two hours is an interval of repeated data in the Markovian stream. The information for this interval can then be summarized by one piece of data during compression.

In Figure 1, we highlight the areas of a real RFID-derived Markovian stream when the RFID tag remained in the same location for two seconds or more. We can observe in this example that extended portions of this stream contain redundant pieces of information.

In our research we developed an RLE algorithm for compressing and decompressing streams that exploited these in-

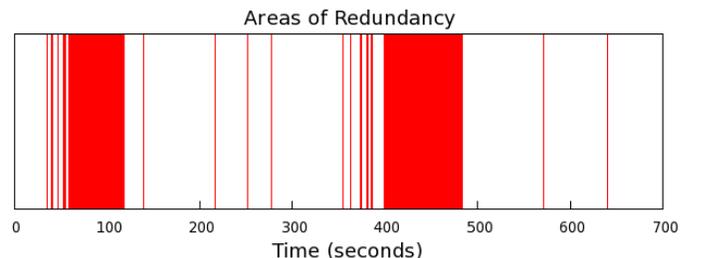


Fig. 1. Visualization of areas of stream compression for a real RFID-derived stream titled Tag3. The x-axis is time in seconds. Areas in the stream that contain redundant information are highlighted in red.

intervals of redundant information. During compression, our algorithm reads the Markovian stream, looking for areas of redundant data. When an interval of redundancy is encountered, we summarize the entire interval as one item of data. After the entire stream is read and compressed, we save a new stream containing those summarized items to disk. This stream thus contains very similar information to the original Markovian stream, but in a more compact form.

After compression, users can decompress and query the saved compressed streams. Our process of decompressing these streams involves reading the compressed stream, identifying those summarized pieces of data, and copying them an appropriate number of times to return the stream to its original length.

In the rest of this study we explore these algorithms in more detail and examine their effects on stream storage, processing time, and query accuracy:

- Section II: We explain Markovian streams in more depth and introduce the Lahar system. We provide a detailed description of the contents of Markovian streams, and explain how they are queried.
- Section III: We detail how we applied RLE compression to Markovian streams (Section III-A) and decompressed/queried those streams (Section III-C). We also provide a brief theoretical analysis proving that worst-case error bounds on a compressed stream vary according to how the stream is being queried, but such error is manageable.
- Section IV: We briefly introduce an interface that allows users to easily compress, decompress, and query a stream.
- Section V: We conduct an empirical study of the effects of compression on several RFID-derived and one synthetic Markovian streams. We prove that compression benefits both storage and query time, while incurring low query errors.
- Section VI- VII: We briefly present other research related to Markovian stream compression, conclude this study with a summary of our results, and give several suggestions for future research.

II. BACKGROUND

In this section, we overview previous research on Markovian streams, the Lahar system, and the process of querying these streams. Because our study expands on these concepts, it is important to understand them in detail.

A. Markovian Streams

Markovian streams are a class of uncertain data streams commonly used in many imprecise-sequence management systems. These streams contain high-level information inferred from *sequential, imprecise* data. Examples of raw data that have been previously modeled as Markovian streams are audio podcasts and RFID tracking information [4] [5]. In representing these real-world data sets as Markovian streams, all probable sentences that were spoken or locations that were visited are summarized in a compact model.

Markovian streams are made up of a series of *imprecise timesteps*, each of which represent an instant of time. In the

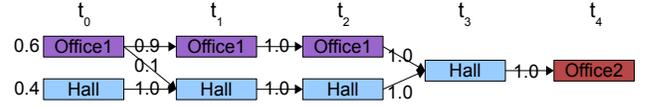


Fig. 2. Visualization of a simplified RFID-derived Markovian stream. Boxes show the probability that Bob was in a specific room that the given time. Arrows show the conditional probabilities of each room given the previous room.

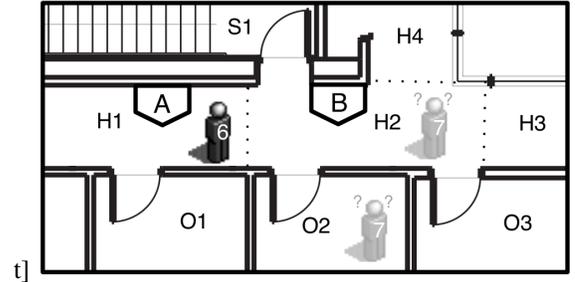


Fig. 3. Example of RFID ambiguity [6]: At time 6 Bob is in hallway H1 and is detected by Antenna A. So, Bob's true location is known. At time 7, for a reason such as a blind spot or a physical obstruction, Bob is not read by any antenna. So, we do not know where Bob is located at that moment. He is either in O2 or H2, each with some probability; his location is thus *uncertain*. [6]

case of an RFID-derived Markovian stream, one timestep will contain several locations and the probabilities that each of those locations was the actual location of the RFID tag. All imprecise timesteps have three pieces of information: 1.) a sequence identification (*SeqID*), to specify the time in the stream, 2.) a Conditional Probability Table (*CPT*), to give the correlations between distributions at this timestep and the next timestep, and 3.) a *marginal*, to provide the probability distribution over all possible locations at the given instant. A timestep's marginal is calculated by taking the product of the probabilities of each of the previous marginal's tuples and the corresponding tuples in the *CPT*.

A Markovian stream's *domain* is the set of possible locations a tag can ever be. The *length* is the number of timesteps contained in the stream.

One simple Markovian stream derived from an RFID-tag attached to theoretical employee Bob is illustrated in Figure 2. This stream has a domain size of three (Office1, Office2, and Hall) and a length of five. Each timestep in this stream contains a probability distribution over Bob's possible locations at that moment in time. These correlations are conditional probability distributions indicating the likelihood of Bob's location at time $t + 1$ given his location at time t . For instance, the second timestep in the stream in Figure 2 would have the following properties:

- 1) SeqID: 1
- 2) CPT:
 - From Office1 to Office1 = 1.0
 - From Hall to Hall = 1.0
- 3) Marginal:
 - Office1: $0.6 * 0.9 = 0.54$
 - Hall: $(0.6 * 0.1) + (0.4 * 1.0) = 0.46$

The first piece of information, the SeqID, states that this timestep represents time 1. The second piece of information,

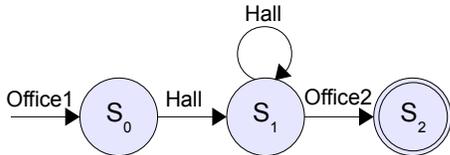


Fig. 4. An NFA for the example event query, “When did Bob move from Office1 to Office2?” This query is satisfied for the Markovian stream in Figure 2 at time 4. The length of this query is 3.

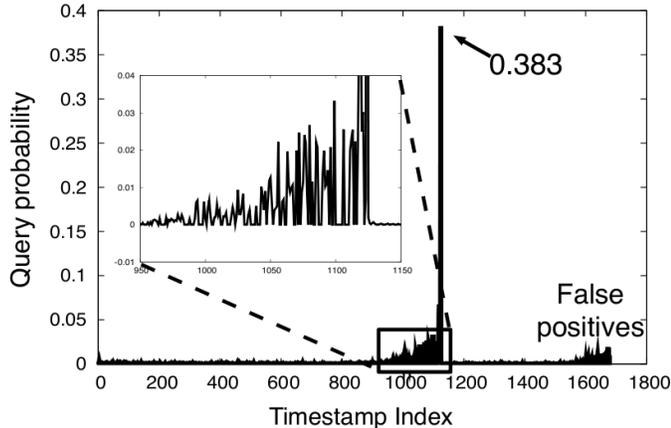


Fig. 5. Output of an event query. The x-axis indicates the sequence ID and the y-axis shows the probability that the user’s query was satisfied at that time.

the CPT, states that if Bob was in Office1 at time 1, he stayed in Office1 at time 2. Otherwise, if he was in Hall at time 1, Bob stayed in Hall at time 2. The third piece of information, the marginal, states that the probability that Bob was in Office1 or Hall at time 1 is 0.54 and 0.46, respectively.

For the rest of our study we use real RFID-derived Markovian streams, similar to our aforementioned example stream for Bob. Our RFID data was produced by mounting RFID readers in various hallways in our campus building. As a person or object wearing an RFID tag moved through the building, its time and location was recorded. The result is a large amount of sequential, noisy data – sequential, because this data is temporal; and noisy, because antennas were ambiguously placed or tags were not read as explained in Figure 3. For every tag, Markovian streams were then inferred from the recorded tuples using a probabilistic inference technique called a particle filter [1]. Our real Markovian streams thus represent the probability distributions over the possible paths each tag could have taken through the building.

B. Event Queries

Markovian streams are commonly queried using a class of query called *event queries*. Event queries find instances of specific patterns in a stream [6]. Some examples of event queries are questions such as, “When did Bob enter the hallway?” or “When did Bob go from the office to the hall?” These queries are equivalent to regular expressions and, subsequently, nondeterministic finite automata (NFA). An example of this type of query for our RFID data is provided in Figure 4. The

```

1. SELECT INSTANTS
2. FROM RFID
3.   WITHKEY = Tag3
4. EVENT E1 NEXT E2 BEFORE E3
5.   WHERE E1.loc = Office1
6.     AND E2.loc = Hall
7.     AND E3.loc = Office2;

```

Fig. 6. Lahar’s query syntax for an event query. Line 1 specifies the type of query. Lines 2-3 specify the stream being queried. Lines 4-7 specify the pattern to search for.

length of a query is the number of states in the corresponding NFA. The query in this figure has a length of three.

The output of an event query is a list of probabilities, one for every timestep in a stream, indicating the probability with which the query pattern was satisfied at that instant. [2] An example of a generic query result is in Figure 5.

There are two types of event queries: *fixed-length* and *variable-length* [3]. *Fixed-length* queries do not have loops in their NFA’s. *Variable-length* queries have loops in their corresponding NFA’s.

C. The Lahar System

Markovian streams are warehoused through a system called *Lahar* (prior work on Lahar includes further system detail [4] [5]). Using this system, users are able to submit event queries on Markovian streams. An example of a Lahar query is given in Figure 6. This query is asking when Tag3 moved from Office1 to Office2 and is equivalent to the NFA in Figure 4 if Tag3 were attached to Bob.

A brief summary of Lahar’s querying process, with emphasis on the *Ex* and *Reg* Operators, is as follows:

- 1) Lahar accepts an event query from the user.
- 2) Lahar opens the appropriate Markovian stream, which is stored on disk, and extracts the imprecise timesteps using the *Ex*(tract) Operator.
- 3) Starting with the first extracted timestep, Lahar uses regular expressions to calculate the probability that the query was satisfied using the *Reg*(ular Expression) Operator. Except for the first timestep, Lahar does not use the marginals to make these calculations; *only the CPT’s*, from which marginals can be computed, are used. Lahar also reads timesteps *one at a time* and *in order*, features that we later exploit in our compression algorithms.
- 4) Lahar performs any appropriate post-processing, such as aggregation, lineage, etc. [2] We do not discuss any of these techniques in this paper.

III. COMPRESSION ALGORITHMS

In this section we introduce the purpose of our research: the compression and decompression processes. Both of these processes work with the topics explained in Section II to store and query compressed streams. We also define new concepts, including *compressed Markovian streams*, *compressed imprecise timesteps*, and *compression functions*.

A. The Compression Process

The compression process is the process of removing redundant timesteps in a Markovian stream and saving the resulting new stream. It begins by accepting a Markovian stream and identifying areas of redundancy. Redundant timesteps are then summarized in a single new timestep, and the new stream is saved to disk. Before this process is explained in more detail, we will first define compressed Markovian streams, imprecise timesteps, and compression functions:

- **Compressed Markovian Stream:** When a Markovian stream is compressed, it is saved as a compressed Markovian stream. This class of stream is nearly identical to normal Markovian streams, but contains a new type of timestep called compressed imprecise timesteps. Lahar also knows to query compressed Markovian streams using a different algorithm.
- **Compressed Imprecise Timesteps:** Compressed imprecise timestep are used to represent multiple redundant timesteps in a Markovian stream. This is a subclass of imprecise timestep, but has an added property called `_count`. This value indicates the number of imprecise timesteps this new timestep represents.
- **Compression Functions:** A compression function is used during the compression process to identify redundant timesteps. These functions accept two timesteps and return a Boolean indicating *compressibility*. Each function determines compressibility in a unique manner. For more detail on how these functions work and how compressibility is determined for this study, see the “difference function” in Section III-B.

The process of compression may now be explained in greater depth:

- 1) The user specifies a Markovian stream to be compressed.
- 2) A new, empty compressed Markovian stream is created, to be populated with compressed imprecise timesteps as the process continues.
- 3) Starting at the beginning of the original stream and reading two adjacent timesteps at a time, timesteps t_a and t_{a+1} are given to a compression function (described in detail shortly). The function returns a Boolean indicating compressibility.
 - While the function returns *false* (i.e. the timesteps were not compressible), the first timestep, t_a , is saved to the compressed Markovian stream as a new compressed imprecise timestep with a `_count` value of 1.
 - When the function returns *true* (i.e. the timesteps were compressible), t_a and subsequent timesteps, t_{a+2} , t_{a+3} , t_{a+4} , ..., and t_{a+n} are given to the compression function while it returns true. When a timestep that cannot be compressed with t_a is encountered, $t_{a+(n+1)}$, the function will return false. Timesteps t_a , t_{a+1} , t_{a+2} , t_{a+3} , t_{a+4} , , and t_{a+n} are then summarized in a new compressed imprecise timestep preserving t_a 's marginal and CPT and a `_count` value of n . The process returns to reading adjacent timesteps, starting with $t_{a+(n+1)}$.

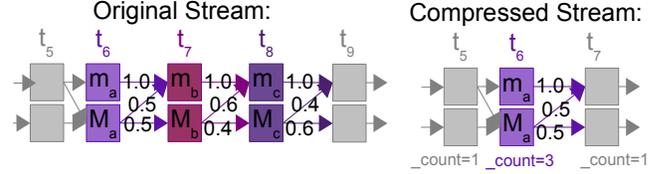


Fig. 7. Visualization of the compression process over the parameter 1.0. m_{a-c} and M_{a-c} represent the marginal values. The difference function returns true for timesteps t_6 , t_7 , and t_8 , so those timesteps were compressed.

- 4) When the entire original Markovian stream is processed, the new compressed Markovian stream is saved to disk, to be queried later.

An example of a portion of this compression processes is illustrated in Figure 7. Our algorithm gives t_5 and t_6 to the compression function, which returns false. t_5 is thus saved to the new compressed stream. t_6 and t_7 are given to the compression function, which again returns true. t_6 and t_8 are given to the compression function, and it returns false. t_6 to t_8 are then summarized in a compressed imprecise timestep with t_6 's marginal and CPT, and a `_count` value of 3. After that new timestep is saved to disk, the compression process returns to reading adjacent timesteps, starting with t_9 .

B. The Difference Function

We developed several different types of compression functions, each with a unique method of determining compressibility. For the remainder of this study, however, we focus on the *difference function*. Like all compression functions, this function accepts two timesteps and returns a Boolean. The difference function, however, also accepts a value called the *threshold parameter*, T , that controls the level of similarity required for two timesteps to be “redundant.” In the rest of this section we explain how the threshold parameter is used, outline how the difference function determines compressibility, provide some real examples of compression using the difference function, and make several claims concerning the effects of this compression method.

1) *The Threshold Parameter:* The difference function considers two probabilities, a and b , to be within the acceptable threshold for compression if they satisfy following inequality:

$$((1 - T) \cdot a) \leq b \leq (\frac{1}{1-T} \cdot a)$$

This inequality ensures that b is within T percent of a ; for lower threshold, the probabilities must be very similar, and for higher probabilities, the probabilities may be dissimilar. Note that the parameter can approach, but never be equal to, 1.0.

2) *Determining Compressibility:* When two timesteps are accepted, the difference function performs the following tests using the aforementioned inequality to determine compressibility:

- 1) Do both timesteps' marginals contain the same tuples?
 - If yes, is the probabilities of each marginals' tuples within the inequality?

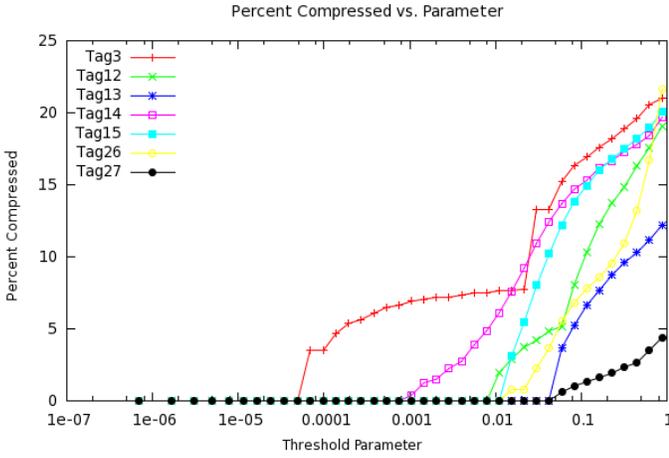


Fig. 8. Graph of the percent of timesteps removed during compression versus the maximum probability difference for several RFID-derived streams.

2) Do both timesteps' CPTs contain the same tuples?

- If yes, are the probabilities of the tuples in each CPT within the inequality?

If the timesteps pass all of these tests, the difference function returns true. If any of these tests returns false, the two timesteps are not compressible and the function returns false.

3) *Difference Function on Real Data:* In Figure 8, we observe how different RFID-derived streams respond to a change in the difference function's parameter. As the maximum probability difference increases, the percent of timesteps removed during compression generally increases for all RFID tags. In other words, *all of these streams experience greater compression when the difference function's threshold parameter is larger*, as expected.

We also observe that when the parameter is set to roughly 0.01 to 0.1, all of these streams experience a jump in compression. For example, Tag3 jumps from only 7% compressed when the parameter is approximately 0.001, to 16% when the parameter is around 0.1. Furthermore, none of these streams achieved 100% compression, even for high threshold values. Since only a certain number of timesteps in a stream usually pass questions 1) and 2) in Section III.A1.c, *most real RFID-derived streams cannot be compressed to 100%*.

There are, however, discrepancies between streams' compressibility: the lowest parameter at which a stream experiences any compression varies (Tag3 is compressed when the parameter is 0.0001, but Tag27 is not compressed until 0.005), and the maximum percent a stream is compressed depends on the tag (Tag2 is compressed up to 30%, but Tag27 cannot be compressed more than 5%). This inconsistent maximum percent compressed demonstrates that though all streams are compressible, there are different degrees of variance in their marginals and CPT's.

4) *Effects of Compression:* Based on the results of compression on real data, we know that the greater the threshold parameter, the greater the compression. We may then predict that the larger the threshold, 1) the less time it will take to compress the stream, and 2) the less disk space the resulting compressed stream will use. Since I/O time during our compression process is mostly writing compressed timesteps to

disk, saving fewer timesteps will improve compression speed. And because the file size of a Markovian stream depends linearly on the number of timesteps the stream contains, a smaller compressed stream will take up less disk space. Our compression algorithm thus allows users to *improve both storage and compression time through the manipulation of the difference function's threshold parameter*. The empirical results of compression are provided in Sections V-A and V-B.1.

C. The Decompression/Querying Process

Querying a compressed Markovian stream is very similar to querying a normal Markovian stream, save for the Reg Operator (all processes, including the Ex Operator and any necessary post processing, are performed like normal). The Reg Operator decompresses a compressed imprecise timestep with a `_count` value of n by making n copies of said timestep.¹ An example of this process is given in Figure 9. The exact process of calculating the query results for these copied timesteps varies slightly depending on the type of query. Decompression/querying is explained in more detail:

- 1) Lahar accepts a query of length m .
- 2) The Ex Operator extracts a compressed imprecise timestep.
- 3) The Reg Operator calculates the probability that the query was satisfied at the timestep using its CPT:
 - If the timestep has a `_count` value equal to 1, the Reg Operator treats the timestep like normal and returns the probability that the query was satisfied at that time.
 - If the timestep has a `_count` value greater than 1, the Reg Operator stops the Ex Operator from extracting any more timesteps. Instead, the Reg Operator uses the same timestep n times and returns n probabilities. This exact process is explained shortly.
- 4) This process of extraction and calculations continues until the end of the compressed stream is reached.

The manner in which the Reg Operator calculates query results for timesteps with `_count` values greater than 1 is based on the query type, m (query length), and n (`_count` value):

- **Fixed-length Querying ($m < n$):** The Reg Operator processes the first m of the identical timesteps and calculates the query results using the CPT's, returning probabilities p_1 through p_m . After the m^{th} calculation, the Reg Operator stops calculating and simply returns $n - m$ times the probability p_m . This reduces querying error, as explained in Section III-D. Using the stream in Figure 9, if the query length were two, the Reg Operator would only calculate two query results, p_6 and p_7 , for times 6 and 7 of the decompressed stream. For time 8, the operator would return p_7 .
- **Fixed-length Querying ($m \geq n$):** The Reg Operator reads all n of the identical timesteps and calculates the query

¹Note that copying a timestep n times produces a Markovian stream that is inconsistent. The marginals of a timestep are normally computed using the previous timesteps' CPT's. However, since we were merely copying a timestep, the marginals and CPT's will not be consistent.

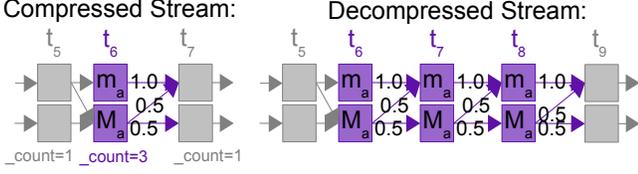


Fig. 9. Visualization of the decompression process. m_{a-c} and M_{a-c} represent the marginal values. Since t_6 had a $_count$ value of 3, that timestep was copied three times.

results using the CPT's, returning probabilities p_1 through p_n . Using the stream in Figure 9, if the query length were three, the Reg Operator would calculate query results, p_6 , p_7 , and p_8 , for times 6 through 8 of the decompressed stream.

- **Variable-length Querying:** The Reg Operator reads all n of the identical timesteps and calculates the query results using the CPT's, returning probabilities p_1 through p_n . The error for this type of query is much higher, as explained in Section III-D. Using the stream in Figure 9, if the query length were three, the Reg Operator would calculate query results, p_6 , p_7 , and p_8 , for times 6 through 8 of the decompressed stream.

1) *Effects of Decompression:* This method of decompressing a Markovian stream should affect the streams in several ways during the querying process. First, the more a stream was compressed, the less time it should take to complete a fixed-length query. More specifically, the Ex Operator's timing is affected because the less timesteps that are read from disk, the less time this operator takes. The Reg Operator's timing is also improved because of the way that fixed-length queries are processed. Second, because compressed timesteps do not contain exactly the same information as the original timesteps, we know that the more a stream was compressed, the less accurate the query results will be. The empirical results of decompressing a stream are given in Sections V-B.2 and V-C.

D. Query Error

In this section, we describe formal bounds on the error induced by compressing Markovian stream intervals using the difference function described in Section III-B. Recall from that section the criteria for determining whether a true probability r (real value) is similar enough to a second probability a (approximate value) to be approximated using the value of a , under a particular compression threshold T : $(1-T)a \leq r \leq \frac{1}{(1-T)}a$. This difference function guarantees a relationship between r and a that yields automatic bounds on how far the real value can be from its approximation.

The error bounds on single probability values (marginal or conditional probabilities) in each timestep of a compressed Markovian stream can be extended to define bounds on the error of the probability of each deterministic sequence, or subsequence, encoded in a compressed Markovian stream. To do this, we note that the probability of any deterministic sequence (or subsequence) of length n in a Markovian stream is the product $p_m p_{c_2} \dots p_{c_n}$, where p_m is the marginal

probability of the first element in the sequence, and p_{c_i} is the conditional probability of the i^{th} element, given the value of the previous element. Because sequence probabilities and single-probability error bounds are both multiplicative, we can multiply the error bounds on individual probabilities to obtain an error bound on the probability of an entire sequence: $p_{true} \geq (1-T)p_m(1-T)p_{c_2} \dots (1-T)p_{c_n} = (1-T)^n(p_m p_{c_2} \dots p_{c_n})$ $p_{true} \leq \frac{1}{(1-T)}p_m \frac{1}{(1-T)}p_{c_2} \dots \frac{1}{(1-T)}p_{c_n} = \frac{1}{(1-T)^n}(p_m p_{c_2} \dots p_{c_n})$

Here, the values p_m and p_{c_i} are the approximate values used in the compressed Markovian stream, and p_{true} is the true value of the deterministic sequence $p_m p_{c_2} \dots p_{c_n}$. Because the approximate values are of course accessible directly from the compressed stream representation, error bounds can be computed simultaneously with sequence probabilities with almost no additional work.

Note that the error factors $(1-T)^n$ and $\frac{1}{(1-T)^n}$ get smaller with the length of the sequence; however, sequence probabilities themselves generally become smaller as sequence lengths increase, due to repeated multiplications of probabilities between 0.0 and 1.0. Thus, even for small sequences, the theoretical error bounds quickly become very loose. The result is that the *relative* error within the theoretical error bounds increases with sequence length, despite the fact that the absolute range of the error bounds is shrinking. Thus, even for small sequences, the theoretical error bounds quickly become very loose.

As an example, consider a sequence of probabilities equal to 0.36, each approximated to the value 0.4 using a threshold parameter of 0.1. For a sequence of length 1, error bounds are: $0.36 \leq r \leq 0.44$, (the true value of r is 0.36, and the approximated value is 0.4). The relative size of the error range here is $(0.44-0.36)/0.36 = 0.234$, for a relative error range of 23% of the true sequence value. By contrast, for a sequence of length 5, the lower bound on error is $0.4^5 * (1-0.1)^5 = 0.00604 \leq r$, and the upper bound on error is $r \leq 0.4^5 * (1-0.1)^{-5} = 0.01734$. The relative size of the error range here is $(0.01734-0.00604)/0.006 = 1.882$, for a relative error range of 188% of the true sequence value. For even a short approximated stream of only five timesteps, this error bound is too loose to be meaningful. Thus, for the remainder of the paper, we focus on empirically-measured error, which is much smaller than theoretical error bounds suggest (Section V).

The above discussion focuses on sequence error, not query error. If a query is satisfied by only a single deterministic sequence in the input stream, then these two errors are the same. However, it is common for many deterministic sequences to satisfy a query, which makes error bounds even looser (the probabilities of each deterministic sequence satisfying a query are summed to obtain the final probability that the query is satisfied at a given timestep). Because empirical error is so much smaller than even the worst-case error of a single sequence, we do not explore the theoretical error bounds on query probabilities here.

IV. RLE SYSTEM OVERVIEW

In this section we briefly introduce the User Interface (UI) used to implement the algorithms described in Section III and

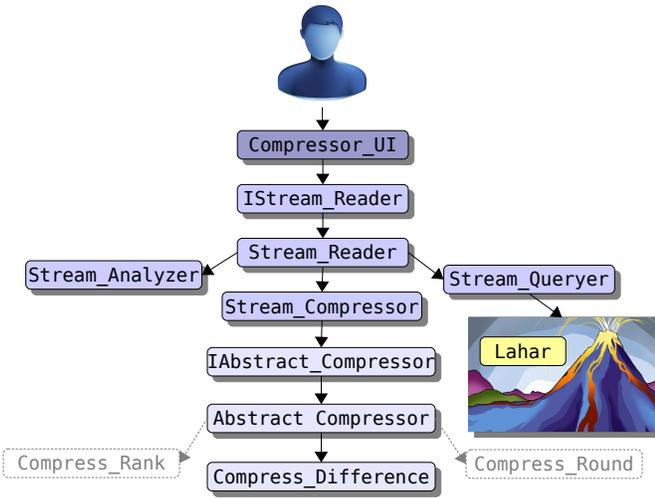


Fig. 10. Visualization of the code architecture. The user perform all processes through Compressor_UI. Compress_Rank and Compress_Round are other compression functions that are not included in this study.

the architecture of its code.

A. The User Interface

In order to allow users to easily compress and query compressed Markovian streams, we designed a simple UI. This UI is capable of performing the following processes:

- Compressing a Markovian stream, as explained in Section III-A.
- Loading a compressed Markovian stream into Lahar.
- Decompression and querying a compressed Markovian stream, as explained in Section III-C

The UI is also capable of performing the following testing processes:

- Performing simple analysis of compressed Markovian streams, such as printing the stream or its length.
- Pseudo-compressing a Markovian stream, which performs the compression processes without saving the compressed stream to disk and is useful for testing.
- Comparing the results of querying two Markovian streams.

B. Code Architecture

The architecture of the code for our UI is outlined in Figure 10. The user accesses all processes through the main class, Compressor_UI. This class helps users to select a process, then creates the classes necessary to complete that process. In the remainder of this section, we explain said classes.

The Stream Readers: *IStream_Reader* is an interface responsible for managing most Markovian stream interactions, such as compression and querying. Classes that implement this interface must be able to 1) specify and retrieve a normal or compressed Markovian stream, 2) return timing statistics, such as total compression or query time, and 3) perform their respective process. *Stream_Reader* is the abstract class that implements *IStream_Reader* and provides functionality to the latter two of the aforementioned tasks. Classes that extend

Stream_Reader are responsible for providing functionality to the third task, which might be compressing a stream, loading or querying a stream using Lahar, or analyzing a stream.

The Compressors: *Abstract_Compressor* is an interface responsible for managing the compression process. Classes that implement this interface must be able to start compression. *Abstract_Compressor* is the abstract class that implements *IAbstract_Compressor* and provides functionality to the compression starter, which actually compresses the stream. Any class that extends *Abstract_Compressor* is a compression function (as explained in Section III-A) and is responsible for comparing two timesteps for compressibility.

Other Classes: Many of our classes interacts directly with Lahar’s original code. *Timestep_Imprecise_Compresed*, which represents a compressed imprecise timestep, and *RegCorrelated_Instants_Compres sed*, which is the Reg Operator for compressed streams, are some of the many subclasses that extend original Lahar classes.

V. EMPIRICAL STUDY

In this section, we examine the impact of RLE compression on stream file size, compression and query runtime, and query accuracy. For most of our tests, we used Tag3, a real 10-minute, RFID-derived Markovian stream.

A. Storage Study

In this section we measure the effects of compression on file size. To observe how compressing a stream affects the stream’s file size on a real stream, we compressed Tag3 multiple times using increasing parameters. The files size of each resulting compressed stream is visualized in Figure 11. We can see from this graph that as the parameter was increased and more timesteps were removed from the stream during compression, the compressed stream’s file size became smaller. These results demonstrate that *compression improves Markovian streams’ storage efficiency*; in reducing redundant areas of streams into a single, compressed timestep, the Compressed Markovian Streams are stored on less disk space, as expected.

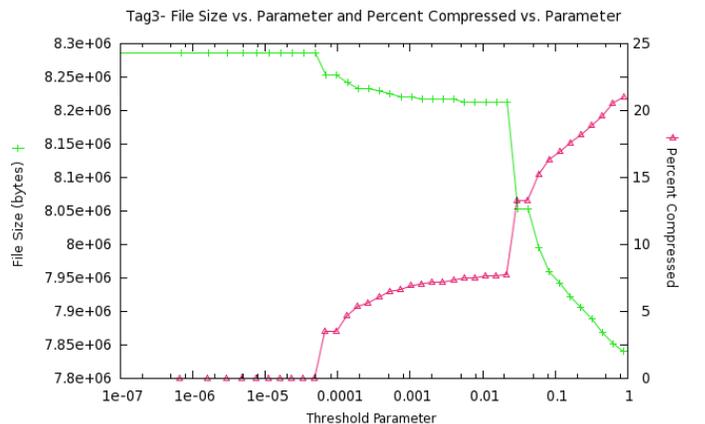


Fig. 11. Graph of Tag3’s file size versus parameter. Overlaid is a graph of Tag3’s percent compressed versus parameter. As the percent of this stream removed during compression increases, the file size of the compressed stream decreases.

Parameter	Length	% Comp.	Avg Comp. Time
1×10^{-7}	710 timesteps	0.0%	1.42 seconds
1×10^{-6}	710 timesteps	0.0%	1.36 seconds
1×10^{-5}	710 timesteps	0.0%	1.37 seconds
1×10^{-4}	685 timesteps	3.52%	1.36 seconds
0.001	662 timesteps	6.76%	1.30 seconds
0.01	656 timesteps	7.61%	1.25 seconds
0.1	590 timesteps	16.90%	1.21 seconds
0.999	543 timesteps	23.52%	1.08 seconds

Fig. 12. Compression statistics for Tag3 over eight parameters. As the parameter increases, the compression time generally decreases.

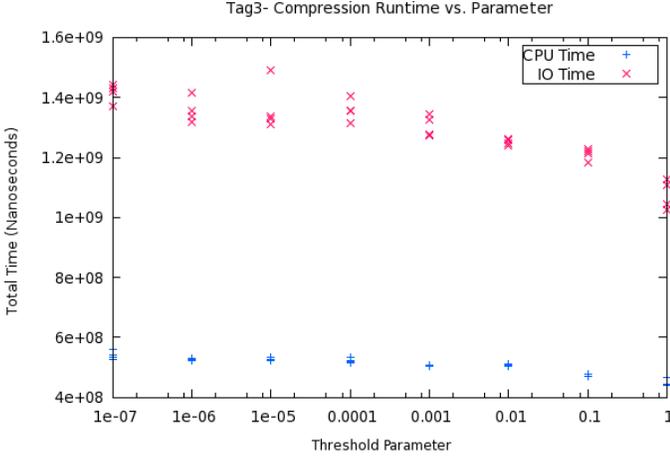


Fig. 13. Graph of the compression runtimes for Tag3 versus the parameter. I/O time changes but CPU time is consistent.

B. Runtime Study

In this section we again use the RFID-derived stream, Tag3 to study compression runtime as well as the effects of compression on querying time.

1) *Compression Runtime*: To observe how changing the parameter, and consequently changing the amount a stream is compressed, affects compression runtime for a real stream, we compressed Tag3 eight times over increasing parameters: 1×10^{-7} , 1×10^{-6} , 1×10^{-5} , 1×10^{-4} , 0.001, 0.01, 0.1, and 0.999.² The results of these compressions are summarized in Figure 12. We can see from this table that *as the parameter is increased and Tag3 became more compressed, compression time I/O was improved.*

Our results are visualized as a graph in Figure 13. For this stream, the I/O times are slightly improved as the parameter is increased. Recall from Section III that the higher the parameter, the shorter the compressed stream, and the less time it takes to write the compressed timesteps to disk. Thus, before the parameter equals 0.0005 and when no compression occurred, approximately 1.5 seconds is spent on I/O during compression. However, when the maximum probability difference is set to 0.999 and the greatest compression occurs, the I/O time is around 1.2 seconds. Though I/O time generally decreases as the parameter increases, the CPU time for all parameters is approximately 0.5 seconds for Tag3. We can therefore

²Our timings statistics were not always consistent due to discrepancies caused by our machines, so each test was run multiple times for each parameter.

Parameter	Length	% Comp.	Avg Query Time
1×10^{-7}	710 timesteps	0.0%	0.352 seconds
1×10^{-6}	710 timesteps	0.0%	0.350 seconds
1×10^{-5}	710 timesteps	0.0%	0.349 seconds
1×10^{-4}	685 timesteps	3.52%	0.349 seconds
0.001	662 timesteps	6.76%	0.339 seconds
0.01	656 timesteps	7.61%	0.340 seconds
0.1	590 timesteps	16.90%	0.321 seconds
0.2	582 timesteps	16.90%	0.314 seconds
0.3	577 timesteps	16.90%	0.310 seconds
0.4	572 timesteps	16.90%	0.309 seconds
0.5	568 timesteps	16.90%	0.310 seconds
0.6	565 timesteps	16.90%	0.305 seconds
0.7	563 timesteps	16.90%	0.304 seconds
0.8	562 timesteps	16.90%	0.305 seconds
0.9	560 timesteps	16.90%	0.307 seconds
0.99	543 timesteps	23.52%	0.300 seconds

Fig. 14. Query statistics for Tag3 over sixteen parameters. As the parameter increases, the query time generally decreases.

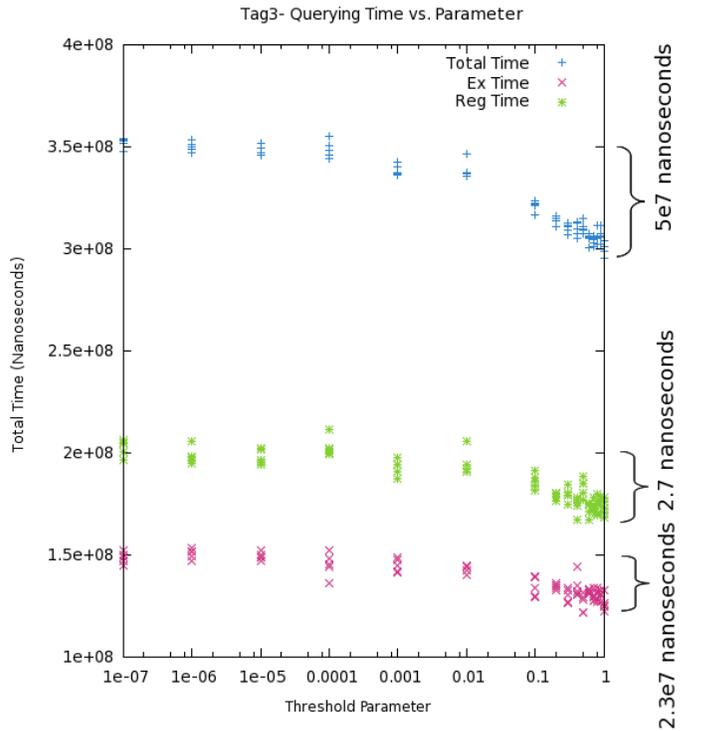


Fig. 15. Graph of total query time versus the parameter. On the x-axis is the parameter at which the stream was compressed and the y-axis is the time in nanoseconds. The duration of the Ex and Reg Operator are also included in this graph. As the parameter is increased, the total Ex and Reg times are decreased.

conclude that *a reduction in I/O time was responsible for the increase in compression speed for Tag3 as the parameter was increased.* This is consistent with our expectations for compression time.

2) *Query Runtime*: For our experiment we used the eight compressed streams from Section V-B.1 as well as eight additional streams using the following parameters: 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9. We queried each of these sixteen streams multiple times using the following fixed-length query:²

```

SELECT INSTANTS
FROM RFID
  WITHKEY = Tag3
EVENT E1 NEXT E2
  WHERE E1.loc = Office1
  AND E2.loc = Office1;

```

These streams' results are summarized in Figure 14. From this table we can see that *the more the Tag3 was compressed, the faster the query was completed.*

We visualize this study in Figure 15, a graph of the total time, Ex time, and Reg time versus the parameter. Before Tag3 was significantly compressed, when the parameter was approximately 0.001, the total and Ex and Reg Operator times were relatively consistent. However, when the stream was compressed more than 5% and the parameter was greater than 0.1, the query time was clearly reduced; the difference between a stream compressed by 0% and a stream compressed by almost 24% was approximately 0.05 seconds. In other words, *increasing Tag3's compression improved both the Ex and Reg time, and ultimately the total query time.* This is consistent with our predictions for fixed-length query time.³

C. Error Study for Fixed-Length Queries

For all queries, compressed streams will produce less accurate results than uncompressed streams. However, as explained in Section III-D, the maximum amount of error between a compressed and uncompressed stream depends on the type and length of the query. In this section we conduct an empirical study that examines the query error seen on real data. For this study we ran the same fixed-length query in Section V-B.2 on Tag3, compressed using the same sixteen parameters in Section V-B. We then compared the results of that query with those of an uncompressed Tag3. The resulting query errors are visualized in Figure 16. We can see from this graph that as the threshold parameter was increased and the stream became more compressed, the percent difference between the compressed and uncompressed query results increases. However, for the majority of the compressed streams (when the parameter was 90% or less) this error is less than 3% on average. Clearly, *the error of this fixed-length query for Tag3 is low.*

D. Analysis of Study Results

Using the results of our study on storage, runtime, and error for Tag3, we can come to several conclusions. Firstly, *file size, compression and query runtimes, and error are directly related to the percent Tag3 was compressed.* As the stream became more compressed, storage and runtime improved and error increased. Secondly, in order to optimize these results so that there is a reasonable amount of error (less than 1%) while still improving storage and runtime are made more efficient, *the parameter for Tag3 should be set to approximately 0.1.* For this value, Tag3 experiences a large jump in percent compressed, a

³Though we were able to improve query time for Tag3, our results were not very dramatic because this tag cannot be compressed by more than 24%. For more significant results, we briefly study a synthetic stream's results in Section V-E.

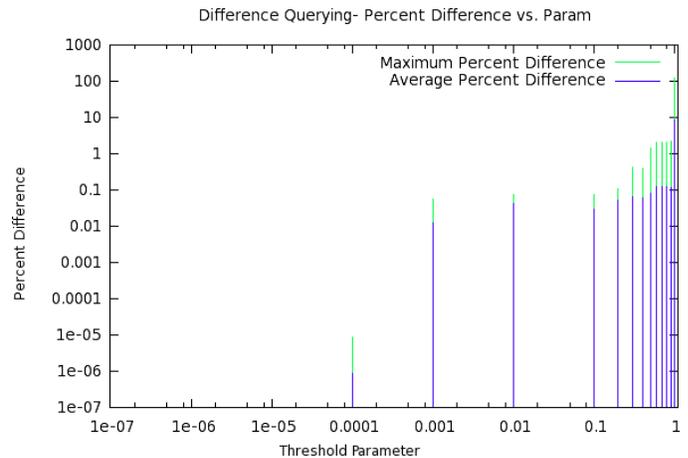


Fig. 16. Graph of the fixed-length query error versus the parameter. As the parameter increases, the discrepancies between the compressed and uncompressed stream increases.

significant drop in file size (by 28.7%, compared to the original stream), a slight decrease in compression and query time (by 0.21 and 0.03 seconds, respectively, compared to the original stream), and an average query error of only 0.07%.

In general, using these empirical results for Tag3 as well as the graph of other streams in Figure 8, we can assert that *most streams' compression will be optimized when the parameter is set between 0.001 and 0.01.* Since most streams experience a jump in compression for those threshold values, we can predict that there will be an improvement in storage and runtime while maintaining a manageable amount of error. Furthermore, we can assert that a parameter value less than 0.0001 and greater than 0.9 are not useful values; they either produce too little compression or too much error for most streams.

E. Synthetic Stream Testing

In addition to RFID-derived streams, we also briefly studied the results of compression on a synthetically-generated Fully Connected, Uniform CPT stream. A *Fully Connected, Uniform CPT* stream contains identical timesteps for which all tuples have the marginal probability $(\text{domainsize})^{-1}$ and the CPT point to all tuples with the probability $(\text{domainsize})^{-1}$. An example of this type of stream is given in Figure 17. This type of stream is ideal for testing the extreme examples of compression; because all timesteps are identical, Fully Connected streams are nearly *100% compressible* for all parameters. For our tests, we generated a stream called FullyConnected500, which had a length of 500 timesteps and a domain size of 10 tuples. The results of compressing this stream, including data on storage and compression runtime, are in Figure 18.

We then submitted the following fixed-length query to FullyConnected500:

```

SELECT INSTANTS
FROM RFID
  WITHKEY = FullyConnected500
EVENT E1 NEXT E2
  WHERE E1.loc = Room1
  AND E2.loc = Room1;

```

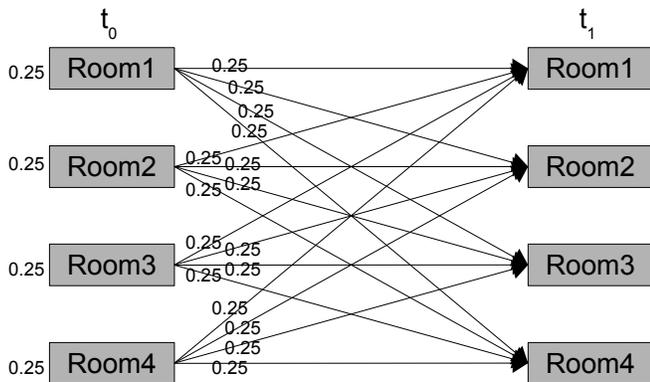


Fig. 17. A Fully Connected, Uniform CPT stream of length 2 and domain size 4.

	Uncompressed	Compressed
Length	500 timesteps	1 timestep
% Comp.	0%	99.8%
File Size	7.3 MB	5.3 MB
Avg Comp. Time	n/a	0.29 seconds

Fig. 18. Compression statistics for synthetic stream FullyConnected500, compressed and uncompressed.

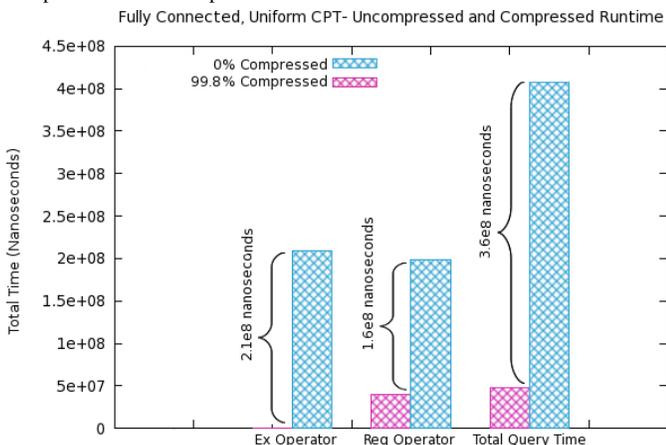


Fig. 19. Graph of the query runtimes for synthetic stream FullyConnected500, compressed and uncompressed.

The timing statistics of this query are in Figure 19. Querying the compressed stream took a total of approximately 0.047 seconds to complete, which was 8.5 times faster than the uncompressed stream. Ex and Reg times were 2,743 and 4.5 times faster respectively with the compressed stream. The Ex Operator took much less time to complete because the compressed stream required only one timestep to be extracted. Similarly, the Reg Operator time was improved because the results of this fixed-length query were copied, not calculated, for most of the stream.

VI. RELATED WORK

As we mentioned in Section II, there have been many studies on Markovian streams, event queries, and Lahar. There have, however, also been various other studies on compressing Markovian streams. Other compression techniques include *Independence Approximation* and *MAP Approximation*

studied by Letchner et al. [2]. While our RLE algorithm performs compression by looking at multiple timesteps at a time, these techniques remove information from every timestep. Independence Approximation discards the temporal correlations in a Markovian stream and essentially only uses timesteps' marginals. MAP Approximation represents a Markovian stream with its single most likely deterministic sequence and essentially only looks at the most probable timesteps in a stream.

VII. CONCLUSION

In this study we sought to improve the file size and processing time of Markovian streams. We introduced RLE compression as a solution to this challenge, and defined techniques for performing such compression on Markovian streams. Using real Markovian streams, we found that RLE compression works on RFID-derived streams, greatly reduces file size, and slightly improves query time, all while maintaining a manageable amount of query error. We also used a synthetic Markovian stream to show that our compression techniques have a greater potential to reduce processing time than what was demonstrated with a real stream.

While the results in this paper are promising, there are remaining challenges for Markovian stream compression. First, longer and more varied Markovian streams have not yet been tested. In this paper we only examined a ten minute RFID-derived Markovian stream; the results of compressing a stream representing an hour or more or the results of compressing an audio or GPS-derived stream may produce more interesting results.

Second, other compression functions that improve the percent compressed or query accuracy have yet to be developed. There are an endless number of methods for determining compressibility, and it would be very simple to write other compression functions. In addition to varying how a function determines compressibility, it is also possible to develop a function that examines three or more timesteps at a time.

Finally, a more advanced UI would allow users to explore Markovian stream compression in more depth. For example, a UI that allows users select other compression functions, or helps to identify the optimal threshold parameter would be extremely useful.

REFERENCES

- [1] A. Doucet, S. Godsill, and C. Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and Computing*, 10(3):197–208, 2000.
- [2] J. Letchner. Warehousing markovian streams (ph.d thesis), 2010.
- [3] 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.
- [4] J. Letchner, C. Re, M. Balazinska, and M. Philipose. Lahar demonstration: Warehousing markovian streams. *VLDB*, 2009.
- [5] J. Letchner, C. Re, M. Balazinska, and M. Philipose. Approximation trade-offs in a markovian stream warehouse: An empirical study. *ICDE*, 2010.
- [6] 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.