

Probabilistic Event Stream Processing with Lineage

Zhitao SHEN[†], Hideyuki KAWASHIMA^{†,††}, and Hiroyuki KITAGAWA^{†,††}

[†] Graduate School of Systems and Information Engineering

^{††} Center for Computational Sciences

University of Tsukuba

Tennoudai 1-1-1, Tsukuba, Ibaraki, 305-8573 Japan

Abstract Many sensor network applications such as the monitoring of video camera streams or the management of RFID data streams require the ability to detect composite events over high-volume data streams. Sensor data inputs from the physical world are usually noisy, incomplete and unreliable. Thus they are usually expressed with probability. To manage this kind of data, probabilistic event stream processing systems are a natural consequence. In this paper, we propose a query language to support probabilistic queries for composite event stream matching. The language allows users to express Kleene closure patterns for complex event detection in the physical world. We also propose a working framework for query processing over probabilistic event streams. Our method first detects sequence patterns over probabilistic data streams using AIG, a new data structure, which handles record sets of active states with an NFA-based approach. Our method then computes the probability of each detected sequence pattern on their lineage. With the benefit of lineage, the probability of an output event can be directly calculated without considering the query plan. We conduct a performance evaluation of our method comparing it with a naive method. Results clearly confirm the effectiveness of our approach.

Key words Probabilistic Databases, Composite Event Processing, Lineage

1. Introduction

Network and sensor device technologies are progressing rapidly. They include a variety of sensor devices such as network cameras, wireless sensor nodes and RFID readers. These devices generate massive sensor data streams, and a variety of primitive physical events can be generated by analyzing them. Since sensor data streams are continually generated, the primitive events are also generated continuously. The continual events can be considered a sequence of primitive events, and recognition of the sequence^(注1) provides us a high level event, which is easy to understand in the physical world. For example, we want to detect the sequence event “Alice leaves a room” rather than the primitive events “Camera 1 detects Alice and then camera 2 detects Alice.” The purpose of this research is to develop an event processor that detects sequence events.

To process a sequence event, an event processor should

cope with two problems. The first problem is the processing of a sequence of primitive events. Although event processing techniques are discussed in depth in the field of active databases [12], processing for a sequence of events has not been clarified. Beyond that, the fatal problem with previous work is the strict limitation of sequence length. In other words, an unbounded length of events could not be considered. The second problem is essential and difficult. It is the uncertainty nature of a primitive event. As described above, a primitive event is generated from sensor data streams. It is widely known that sensor data streams include noise, so the interpretation has more than one possible candidate. Such uncertainty is usually treated with probabilities, so an event processor should have a framework to deal with probability.

For the first problem, recent work with complex event processing and language over certain streams is extensive [2], [7], [8], [13]. Especially, SASE [13] is a promising project to process primitive event streams. SASE designs a new language rather than adhering to SQL based on NFA. A recent paper [8] of SASE presents a rich declarative event language called SASE+. SASE+ can be used to define a wide variety

(注1): In this paper, we call a sequence of primitive events a “sequence event.”

of Kleene closure patterns,^(注2) which can be used to extract a finite yet unbounded number of events with a particular property from the input stream. Such patterns always appear in event streams in RFID and sensor network applications.

For the second problem, much work with probabilistic databases has been done. Probabilistic databases have received much attention because of their ability to deal with uncertainty resulting from sensor data, data cleaning tasks and information extraction. A general kind of probabilistic database model based on possible world semantics is described in [6]. The main idea in a probabilistic database is that the state of the database, i.e. the instance I , is unknown. Instead the database can be in any one of a finite number of possible instances I_1, I_2, \dots , called possible worlds, each with some probability and the probabilities of all the possible worlds must sum up to 1. Since the confidence computation requires massive cost, the concept of lineage was incorporated to accelerate the computation in ULDB [3].

As described above, the purpose of this research is to design a sequence event processor. From the standpoint of event processing, SASE+ shows more promise than SQL. It does not, however, consider the probabilistic standpoint. On the other hand, from the standpoint of probabilities, ULDB is promising because it supports the widely accepted possible worlds model and has a method to dramatically accelerate confidence computation, which requires massive cost when using a naive method. The problem with ULDB is that it supports only the relational data model, and no other models are considered. Since the theoretical model of SASE+ is NFA, no efficient methods with lineage based confidence computation have yet been considered.

We propose a model that computes confidence based on lineage for NFA. Our model is the extension of the SASE+ model with probabilities using the possible worlds model. Our model also computes confidence for each event based on the possible worlds model using lineage, which dynamically accelerates the computation. The novel point of this research is the realization of lineage-based confidence computation for a probabilistic sequence event. To deal with a sequence event, SASE proposed a data structure Active Instance Stack (AIS). However, AIS cannot be applied for our model because of its simplicity. Instead of AIS, we propose Active Instance Graph (AIG) to manage the record of the set of active states based on Nondeterministic Finite Automata (NFA) over an unbounded probabilistic event sequence.

The rest of this paper is organized as follows: Section 2

introduces the probabilistic event stream model and query language for our system. Section 3 introduces a decoupled framework for probabilistic event stream processing. Section 4 proposes a special data structure, called Active Instance Graph (AIG), to compute confidence using lineage. Section 5 presents the experimental results that confirm the efficiency of AIG. Section 6 concludes the paper.

2. Related Work

In this section we provide a brief introduction of the related works. First we will discuss the event processing engines which recently have attracted attention by the community because of their ability to find matches to event patterns defined by users from high speed data streams. Then, we will go through the topic of probabilistic data management.

Event processing engines such as Cayuga [4], [7], SASE [8], [13] and HiFi [9] are closet in spirit to our work. Cayuga [4], [7] is a prototype event stream processing system, which enables high-speed processing of large sets of queries expressed in the CEASAR algebra. SASE [8], [13] describe events in a formalism related to regular expressions and use some variant of a NFA model. In the community of sensor networks and the HiFi project [9] in particular propose a new architecture for processing complex high level events on the basis of data outputted by receptor devices at the edges of High Fan-In system. But most of event processing engines does not consider the uncertainty always appearing in real world sensors.

Probabilistic database systems such as Trio [1] and [5] propose new data models, extending the tradition relational model, able to capture incomplete or imprecise data usually appearing in physical world. Recently Siciu in [5] introduces possible worlds model and studies the problem of efficiently answering queries on probabilistic databases. A systematic effort is made to enable traditional relational operators to handle values with probabilities. Trio project [1] defines a probabilistic model called ULDB [3]. They exploit lineage in ULDB [10] to avoid the “safe” problem in [5] and make confidence computation efficient.

3. Probabilistic Event Matching Language

This section presents the underlying probabilistic stream models and briefly overviews our query language using an example.

3.1 Probabilistic Event Stream Model

Inputs: The input to the event processing system is a probabilistic event stream, that is, an infinite sequence of probabilistic events. Each probabilistic event in a stream represents an atomic occurrence with uncertainty. A prob-

(注2): Often used in regular expressions

abilistic event is a set of one or more alternatives, which can be regarded as different observations of an occurrence. Each alternative has a probability representing the possibility of a certain state of events. Thus different alternatives of a probabilistic event represent mutually exclusive states for the event. The sum of probabilities of all alternatives in one probabilistic event must be equal and less than 1. When the sum is less than 1, the probability that the event does not occur is $(1 - \delta(e))$, where $\delta(e)$ is the sum of the probabilities of a probabilistic event e 's alternatives.

Outputs: The output of an event processing system is also a probabilistic event stream. The difference between an input stream and an output stream is that probabilistic events in an output event stream have only one alternative, and the event types in an output stream are fixed by a certain query.

Note that our ultimate goal is to make the models of input and output identical. In so doing, the output stream of a query can be used as another query's input naturally. This topic is an important area of our future work.

3.2 Overview of the Language

In this paper, we extend a query language on SASE+ [8] with probability. We introduce our query language using an example. Figure 1 is the first query example.

```
PATTERN SEQ(Camera-in c1, Camera-out c2)
WHERE partition_contiguity(c1,c2) { [name] }
WITHIN 5 sec
HAVING Conf(*)>0.5
RETURN name
```

Figure 1 Query Example 1

The query in example 1 is based on SASE+ [8] and we have one extension for probability. We briefly describe SASE+ and then describe our extension.

a) SASE+ Language

The input stream is a probabilistic event stream containing face recognition results from multiple video cameras. Event type Camera-in notates the recognition results from a camera inside the room. Camera-out contains the result for outside. The pattern SEQ (Camera-in c1, Camera-out c2) represents the event pattern for a certain person leaving a room. Function partition_contiguity() means the events are conceptually partitioned based on a certain condition. The predicate [name] shows the partition condition requiring that all relevant Camera-in and Camera-out events have equal attribute values in the name. WITHIN is expressed as a sliding window. In Example 1, sliding window 5 sec means that the

capturing times of the two cameras must be within 5 seconds.

In Example 1, the predicates in HAVING require that the probabilities of an output result must be larger than 0.5. The distinction between WHERE and HAVING here is analogous to that in SQL. The WHERE clause gives the condition only for input streams. The difference is that the HAVING clause here is applied to each matched sequence pattern, while the HAVING clause in SQL is applied to each group created by GROUP BY.

The RETURN clause finally translates each matched pattern into a result uncertain event. "name" is the name of the person. Therefore, the output of result events shows the timestamps and the person names as a probabilistic event stream.

b) Probabilistic Extension

Our extension on probability is shown as Conf(*) in example 1. It is an in-build function introduced to calculate the probability of output composite event streams. Superficially, our language is mostly the same as SASE+, the difference being the confidence computation based on the possible worlds model. However, its internal processing mechanism differs significantly. We describe the differences in the remainder of this paper.

4. Query Processing

It turns out that the lineage information tracked can be used to compute probabilistic values for SQL based possible worlds [3]. We are inspired by this work. Theoretically, this work [3] focuses only on relational algebra and, therefore, it is not clear whether the technique can be applied for other data processing models. We propose to incorporate their lineage based confidence computation method to NFA, which is a theoretical model of SASE+.

We think a decoupled framework can be used for query processing to divide the data computation and confidence computation. For probabilistic event streams processing, we decouple the query processing into two steps:

1. Pattern Matching, in which we match probabilistic event streams with the sequence pattern specified in a query without dealing with probabilities of the events, and translate matched pattern sequences into output probabilistic events.

2. Confidence Computation, in which we compute probability values for output results based on their lineage from matched patterns.

We next show how decoupling pattern matching and confidence computation perform. Before explaining pattern matching processing, as a preparation to our proposition, we introduce how the probabilities of composite events can

be calculated with lineages of results.

4.1 Confidence Computation with Lineage

[6] shows that naive propagation of confidence assuming independence of tuples in intermediate results may lead to incorrect confidences in the result. This paper uses the approach of managing lineage for confidence computation [3] to avoid the problem and conserve the lineage of each result. A lineage representing sufficient dependency information of query results is important when query results are used for another query’s inputs.

In this paper, each alternative (possible event) e in probabilistic event streams contains lineages, which intuitively capture “where e came from.” Lineage is represented as a function λ that associates with each alternative identifier a boolean formula whose symbols are other alternatives in streams.

Lineage for Base Events: A base probabilistic event that contains only one alternative is not derived from other events and has no relation with any other events. We can therefore define $\lambda(e) = e$ for every event e . Assume a base probabilistic event has more than one alternative. Multiple alternatives of the same event are mutually exclusive, i.e., they cannot coexist in any possible world. Thus, alternatives of the same event cannot be treated as independent. In this paper, we use the approach in [10] to initially treat alternatives as if they are separate events. A coin tossing proof for the transformation is given in [10]. We set the lineage of each a_i in the same probabilistic event, $\lambda(a_1) = a_1, \lambda(a_i) = (\neg a_1 \wedge \dots \wedge \neg a_{i-1} \wedge a_i)$. Let the probability of an alternative a_i be p_i ; the probability of a_i should be replaced by

$$c(a_i) = \frac{p_i}{\sum_{j=i}^n p_j + \delta}, \text{ where } \delta = 1 - \sum_{k=1}^n p_k \quad (1)$$

Lineage for Event Sequences: Algorithms to derive lineage from relational operators are given by [10]. We assume an event sequence exists only in the possible worlds where all the events in the sequence exist. Thus if we have an event sequence $(e_1, \dots, e_{n-1}, e_n)$, we can give the lineage of an event sequence $\lambda(s)$ by logical conjunction as follows:

$$\lambda(s) = \lambda(e_1) \wedge \dots \wedge \lambda(e_{n-1}) \wedge \lambda(e_n) \quad (2)$$

Lineage for Composite Events: Because sequence patterns usually match more than one event sequence over probabilistic event streams. Duplicate elimination is always used for results that represent the same event. For a certain composite event e with multiple matched sequences $(s_1, \dots, s_{n-1}, s_n)$, the lineage is given by

$$\lambda(e) = \lambda(s_1) \vee \dots \vee \lambda(s_{n-1}) \vee \lambda(s_n) \quad (3)$$

Confidence Computation: [10] gives the basic and optimizing confidence computation algorithm based on lineage. For this paper, since lineage of the composite event is defined, we can simply borrow their approach to compute the probabilities.

Next is an example to show how probabilities are computed based on lineage information over probabilistic event streams.

```
PATTERN SEQ(Camera-in c1, Camera-out c2)
WHERE partition_contiguity(c1, c2) { [name] }
WITHIN 5 sec
RETURN count(name)
```

Figure 2 Query Example 2

Figure 2 shows the second query example, a query detecting the number of persons leaving the room in the most recent 5 seconds.

Suppose the input probabilistic event stream is as follows:

time	event type	(id, person)
100	Camera-in	(11, A):0.8 (12, B):0.2
101	Camera-out	(21, A):0.9 (22, B):0.1

From this event stream and sequence pattern, we can obtain two candidate event sequences: $s_1(11, 21), s_2(12, 22)$. Each sequence has only one person. Thus duplicated elimination strategy is used for the final output.

For base events in Camera-in and Camera-out, because of the alternatives in probabilistic events, we get the lineages and probabilities:

$$\begin{aligned} \lambda(11) &= 11, P(11) = 0.8 \\ \lambda(12) &= \neg 11 \wedge 12, P(12) = 1 \\ \lambda(21) &= 21, P(21) = 0.9 \\ \lambda(22) &= \neg 21 \wedge 22, P(22) = 1 \end{aligned}$$

The final lineage of the output leaving room event can be expanded as $\lambda(\text{one person}) = \lambda(s_1) \vee \lambda(s_2) = (\lambda(11) \wedge \lambda(21)) \vee (\lambda(12) \wedge \lambda(22)) = (11 \wedge 21) \vee ((\neg 11 \wedge 12) \wedge (\neg 21 \wedge 22))$. With this lineage, the probability of the final output composite event $P(\text{one person}) = P(11 \wedge 21) \vee ((\neg 11 \wedge 12) \wedge (\neg 21 \wedge 22))$ can be calculated. $P(\text{one person}) = 0.74$, which is the correct answer.

5. Pattern Matching

For multiple event stream inputs, existing stream processing systems only use relational joins to create a new relation with all the input information. This approach, however, is not efficient for pattern matching over temporal or

time-series data streams, because computational complexity grows exponentially with the number of joins. In this paper, we define a timestamp-based union of all tuples from different streams [7]. Input event streams are merged into one probabilistic event stream ordered by timestamps.

5.1 NFA

For pattern matching over streams, a useful method is to adopt Nondeterministic Finite Automata (NFA) to represent the structure of an event sequence [7], [13]. [13] use an NFA-based approach to implement a pattern matching strategy called sequence scan and construction (SSC). NFA’s structures are enforced by the translation from the sequence pattern specified in the language. Since our sequence pattern is defined by the language extended by SASE+, we can use the same approach and semantic model to build an NFA. A naive approach to execute pattern matching over uncertain probabilistic streams is to enumerate all possible worlds and scan the sequence based on NFA in each possible world. This approach is obviously not feasible because the number of possible worlds increases exponentially as the number of tuples grows in a sequence.

For matching patterns over probabilistic event streams, we adapt sequence scan and construction (SSC) [13] to generate the sequences matching the pattern. Our main contribution, however, lies in (1) extend the present event pattern matching techniques to efficiently support probabilistic event streams, and (2) a special data structure called Active Instance Graph (AIG) is managed to record the set of active states based on NFA over probabilistic event streams.

For a concrete example, consider the query pattern

$SEQ(A\ a, B^+\ b[],\ C\ c)$

where events a , b , c are constrained by the conditions A , B , C . In the following description, a lower-case (e.g., ‘ a ’) letter represents an event satisfying the condition denoted by its corresponding upper-case letter (e.g., ‘ A ’). Because different alternatives in one tuple may satisfy the same condition, the signal below each event is assigned a couple, including not only its timestamp but also its alternative id. For example, $a_{1,11}$ presents an event at timestamp 1 that satisfies the condition A , and its alternative id is 11.

At the first step, an NFA is created for the sequence pattern. Figure 3 shows the NFA created for the example pattern (A, B^+, C) , where state 0 is the starting state, state 1 is for the successful recognition of an A event, state 2 is for the recognition of a B event after that, and likewise state 3 is for the recognition of a D event after the B event. State 2 contains a self-loop with the condition of a B event representing the Kleene plus in the pattern. State 3 is denoted

using two concentric circles to represent an accepting state of the NFA.

5.2 Active Instance Graph

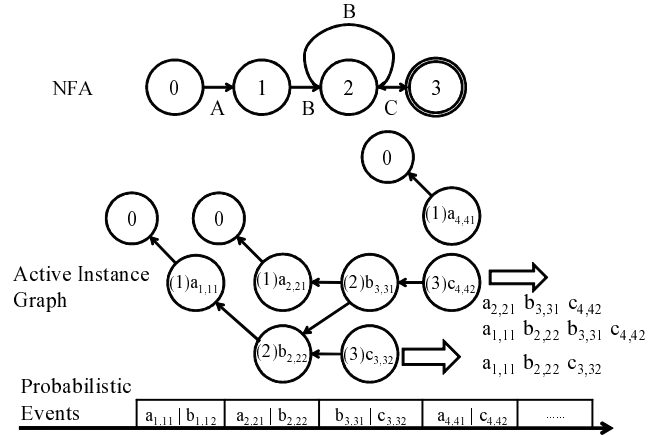


Figure 3 NFA to AIG

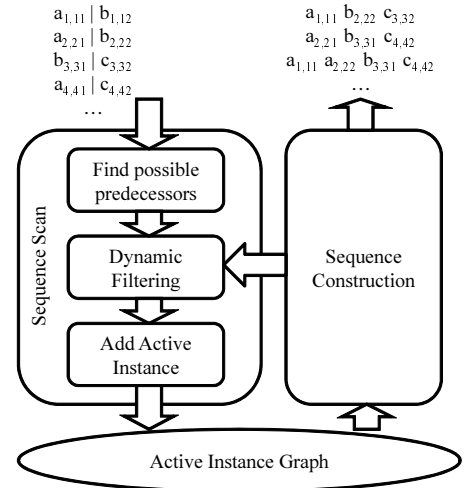


Figure 4 Execution Plan

To keep track of these simultaneous states, we employ an auxiliary data structure, Active Instance Graph (AIG). We propose to efficiently detect event sequences in possible worlds. The algorithm performs as follows:

Sequence Scan. Each node presents a probabilistic event at a certain NFA state. Such events are referred to as active instances of this state. A node is created at the event arrival if it satisfies the transition condition to this state. (Note that the transition condition may need the information of events in previous states. How to track these events is discussed later.) Figure 3 shows the evolution of an AIG after the event stream at the bottom of the figure is received. Each active state instance has a timestamp, alternative id, and an NFA state (shown in brackets). Each node in the graph has predecessor pointers specifying the active state instances

that it came from. A state 0 node is added at each point to initiate a new search for every arriving “a” event.

Sequence Construction. When previous active instances are required, sequence construction is invoked. For example, an accepting state is reached by NFA and we should immediately output the event sequence that the most recent event has completed. Or, in a situation for dynamic filtering, complete or part event sequences should be generated to restrict the possible sequences in sequence scan.

An approach to sequence construction is to traverse back along the predecessor pointers until an instance of state 0 is reached. Possible event sequences can be generated by enumerating all possible paths from the state 0 to the current instance. In Figure 3, at the instance when the event $c_{4,42}$ at accepting state is encountered, we can obtain two possible event sequences (shown at the right of the AIG in Figure 3) using this approach. An AIG has a temporal structure where the current instance has only the pointers of the previous instance. So an AIG is a kind of DAG (Directed Acyclic Graph). A simple algorithm for searching a DAG is a single depth first search with the complexity $O(E)$, where E is the number of edges in the DAG.

Dynamic Filtering. When additional conditions are added to the sequence pattern, we perform a dynamic filtering strategy to prune the possible sequences at an early stage. In practice, a pattern with simple structure may have a huge number of matched patterns, but people usually assign conditions to the pattern to find only the events they are interested in. In this case, dynamic filtering strategy is effective, because intermediate result sizes have been significantly reduced. Conditions for a pattern usually compare the current states with the previous states. For example, an equivalence test is always done for an event sequence. Therefore, sequence construction is called to get the previous information.

An execution plan for matching patterns over probabilistic event streams is shown in Figure 4. The sequence scan is called when new events arrive. First, we find the possible predecessors in the active instances. Second, we do a dynamic filter over predecessors and their sequence if necessary. Third, we maintain the DAG by adding active instances that satisfy the conditions. When an accepting state is achieved, sequence construction is called to generate the result sequences. Sequence construction is also used to dynamically filter possible sequences.

6. Experimental Evaluation

6.1 Experiment Setup

We implemented the performance evaluation presented in the previous section. All experiments were conducted on a

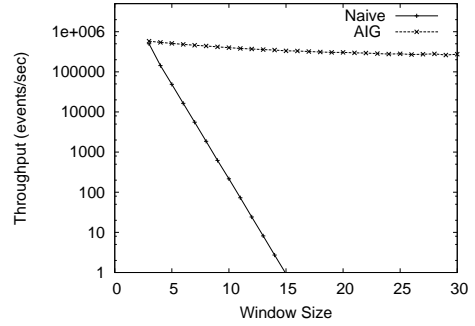


Figure 5 Throughput with Window Size

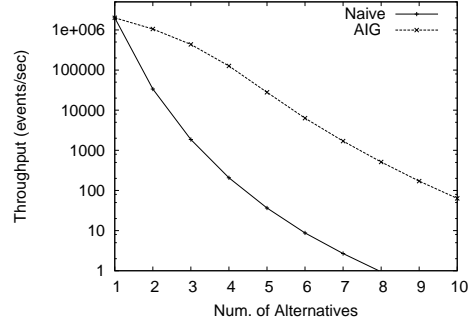


Figure 6 Throughput with Num. of Alternatives

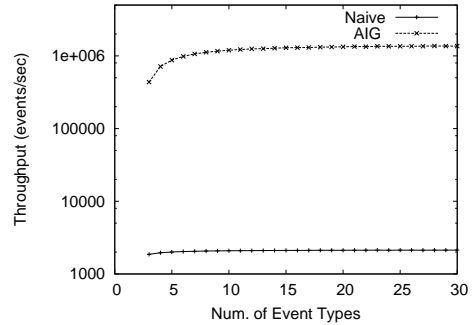


Figure 7 Throughput with Event Types

machine with an Intel Celeron 3.06GHz and 2GB RAM running CentOS Linux 2.6.18.

To examine pattern matching performance using AIG, we implemented an event simulator that creates a random event stream. The query pattern used for evaluation is (A, B+, C). In this experiment, we compare AIG with a naive algorithm that enumerates every possible world in a window. Performance in our experiments is measured as throughput, that is, the number of events processed per second. The window size used in experiments is attached to the whole sequence. The number of alternatives is fixed in every event tuple. The different event types are the possible events this tuple can take. Therefore, the number of event types can be regarded as the domain size of events.

6.2 Experiment Results

Varying window size: We fixed the number of event types as 3 and the number of alternatives as 3, varied the win-

dow size, and examined the performance of the algorithms as pattern matching over a probabilistic event stream. Figure 5 shows the throughput results of the two algorithms. Note that the Y-axis (throughput) is presented in a logarithmic scale. The X-axis is the window size varied from 3 to 15. Clearly, AIG outperforms the naive method. We see that performance of the naive method fell sharply. At the window size of 15, the throughput is about 1 event/sec, which is not practical.

Varying the number of alternatives: We fixed the number of window sizes as 8 and the number of event types as 3. We varied the number of alternatives from 3 to 10. The throughput results are shown in Figure 6. When the number of alternatives is 1, the input event stream becomes a certain stream. In this case, without considering possible worlds, the throughput of both algorithms is almost the same. And we see that the performance of both algorithms is reduced, because the number of possible worlds increases exponentially with the number of alternatives. But the degradation ratio of AIG is much lower than the naive algorithm benefit from the pruning phase.

Varying the number of event types: In this experiment, we investigate the algorithms' sensitivity to the number of event types. We considered the window size as 8 and the number of alternatives as 3, and varied the number of event types. In Figure 7, we find that the change of performance becomes smooth under a large domain size, because the number of matched patterns is reduced in this situation. Under this parameter, the throughput of AIG is about 1000 times higher than that for the naive method.

7. Conclusions

This paper proposed a query language and probabilistic query processing method combining the elements from ULDB and SASE, and extending them with (1) A probabilistic data model for event streams. (2) A decoupled framework of probabilistic event stream processing. (3) A lineage-based approach to confidence computation for event sequence patterns. (4) An extension of the present event pattern matching techniques to efficiently support probabilistic event streams using a special data structure called AIG.

For future work, we plan to develop an event stream processing system using the methods presented in this paper. Optimization is needed for real-time processing.

Acknowledgement

This research has been supported in part by the Grant-in-Aid for Scientific Research from JSPS(#18200005, #18700096).

文 献

- S. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *Proc. of VLDB*, pages 1151–1154, 2006.
- [2] Y. Bai, F. Wang, P. Liu, C. Zaniolo, and S. Liu. Rfid data processing with a data stream query language. In *Proc. of ICDE*, pages 1184–1193, 2007.
- [3] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. Uldb: Databases with uncertainty and lineage. In *Proc. of VLDB*, pages 953–964, 2006.
- [4] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Oshser, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *Proc. of SIGMOD (Demo)*, pages 1100–1102, 2007.
- [5] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proc. of VLDB*, pages 864–875, 2004.
- [6] N. Dalvi and D. Suciu. Management of probabilistic data foundations and challenges. In *PODS*, pages 1–12, 2007.
- [7] A. J. Demers, J. Gehrke, and e. a. M. Hong. Towards expressive publish/subscribe systems. In *Proc. of EDBT*, 2006.
- [8] Y. Diao, N. Immerman, and D. Gyllstrom. Sase+: An agile language for kleene closure over event streams. In *UMass Technical Report 07-03*, 2007.
- [9] S. Rizvi, S. R. Jeffery, S. Krishnamurthy, M. J. Franklin, N. Burkhart, A. Edakkunni, and L. Liang. Events on the edge. In *Proc. of SIGMOD (Demo)*, pages 885–887, 2005.
- [10] A. D. Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *Technical Report, Stanford InfoLab, 2007 and Revision will Appear in ICDE08*.
- [11] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.
- [12] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. of ACM SIGMOD*, pages 407–418, 2006.

[1] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth,