Sequence Pattern Matching over Event Data with Temporal Uncertainty

Yongluan Zhou [†1], Chunyang Ma [#], Qingsong Guo [†2], Lidan Shou [§1], Gang Chen [§2]

[†]University of Southern Denmark, Denmark
{[1]zhou,[2]qguo}@imada.sdu.dk

[#]IBM CRL, Beijing
machybj@cn.ibm.com

[§]Zhejiang University, China
{[1]should,[2]cg}@zju.edu.cn

ABSTRACT

In this paper, we consider complex pattern matching over event data generated from error-prone sources such as low-cost wireless motes, RFID. Such data are often imprecise in both their values and their timestamps. While there are existing works addressing the problem of spatial uncertainty (i.e. the uncertainty of the data values), relatively little attention has been paid to the problem of temporal uncertainty (i.e. the uncertainty of the event timestamps). As a step to fill this gap, we formulate the problem of matching complex sequence patterns over time-series data with temporal uncertainty and propose a new indexing structure to organize the information of the uncertain sequences and a set of efficient pattern query processing algorithms. We conduct an extensive experimental study on both synthetic and real datasets. The results indicate that the query processing algorithms based on our index structure can dramatically improve the query performance.

1. INTRODUCTION

Many emerging applications need to detect complex patterns over event data. Examples include supply chain management [27], tracking in libraries [23], environmental monitoring [8], business process management [16], suspicious behavior monitoring [20] and healthcare [15, 7]. In these applications, the input event data are often generated from various devices, such as wireless motes and RFID (Radio Frequency Identification) readers. Generally, an input event can be represented as a triplet: $(event\_id, time, attr\_values)$, where $time$ is the time when the event is detected and $attr\_values$ is a set of attribute values associated with the event. Consider an RFID deployment system inside an office building, the fundamental attribute of an event is the location of a person or an object over time. An event data output by the RFID system could contain several attributes and a timestamp as: *('Allen','Room 401',* 10:05*)*, which means that the person named Allen enters into Room 401 at 10:05. A typical pattern query would look for a certain interesting sequence of events. An example query could look for an event *('Allen', 'Room 401', . . .)* followed by another event *('Allen', 'Room 402', . . .)*.

A major challenge of this problem is that the data sources in the above applications are error-prone and hence the input data of the pattern queries are often imprecise. First, the locations of objects may be imprecise due to various reasons [19, 26], including conflicting readings and missed readings. For instances, a person is detected by two or more near-by sensors at the same time or RFID readers may not be able to detect all tags when there are too many tags in their vicinity. Second, events' occurrence time can be uncertain due to clock errors of RFID readers, granularity mismatch or clock synchronization problem among distributed sensors/systems. In such cases, the actual occurrence time of the events is unknown and can only be estimated as a time range with high probability [31].

Most existing work on probabilistic query processing over uncertain data mainly focused on data's spatial uncertainty [19, 26]. However, the temporal uncertainty in event data brings great challenges, especially for sequence pattern queries which assume a total ordering of the input event data. For example, suppose we have two events: e_1 with $e_1.TI = [1, 3]$ and e_2 with $e_2.TI = [2, 4]$, where TI denotes the range of possible occurrence time of an event. Here there are two possible occurrence orders of e_1 and e_2, which adds additional complexity for matching sequence patterns.

Temporal uncertainty of event data is studied in [31], where the occurrence time of events are assumed to be independent. However, in reality, this assumption often cannot hold. For example, in the above RFID application, consider two events corresponding to the same person and with different locations. As it is impossible for one person to be at two different places at the same time, the occurrence times of the two events are not independent. Especially when the time intervals of these events overlap, one has to carefully deduce their possible occurrence time.

This paper can be considered a complimentary work to the existing techniques for handling spatial uncertainty. In summary, our contributions include the following:

- We formulate the problem of sequence pattern matching over event data with spatial and temporal uncertainty. In particular, we formally define a data model of event sequences with both spatial and temporal uncertainty and the semantics of pattern matching queries. In our data model, events are partitioned into a number of dependency groups according to their temporal dependencies. Basically, events from the same group have temporal dependency, while events from different groups are temporally independent.
- To speed up the query processing, we propose an index structure to organize and index event data. In the index, the temporal dependency relationship of events are captured by a set of integer codes, based on which the ordering of events in the same dependency groups can be deduced efficiently. Furthermore, a multidimensional index is used to index all the

10.5441/002/edbt.2014.19

attributes and time intervals of the events.

- Based on the above index structure, we develop algorithms to efficiently extract patterns from the event data. The algorithm is able to optimize the evaluation order of the sequence pattern and utilize the index structure to minimize the number of I/O access.

- We perform an extensive experimental study with both synthetic and real datasets to evaluate our proposed approach. The results indicate that both the index construction algorithm and the query processing algorithms are efficient and scale well in terms of query response time and I/O cost.

The rest of the paper is organized as follows. Section 2 reviews the related work and Section 3 formulates the problem by formally defining the data and query models. Section 4 describes our indexing scheme. Section 5 presents the details of the algorithms for processing pattern matching queries. Section 6 reports the experiments study and the paper is concluded in Section 7.

2. RELATED WORK

2.1 Event processing over event streams

Complex event processing (CEP) over event streams has been studied in several recent works [4, 5, 11, 14, 28, 29, 31]. These works focused on extracting sequence patterns from event streams. However, most of these existing works considered only event sequence with precise attribute values. Some of them [5, 11, 28] use time interval to represent the duration time of each event, however the duration time is considered precise and events are still given a strict partial order.

There are also some recent efforts that addressed out-of-order streams, where events do not arrive in the order of their occurrence times [22, 24]. In these papers, the timestamps of events are precise, so the ordering of the events are still deterministic.

In [19], Christopher et al. process event queries on event streams with uncertain attribute values with precise timestamps. Zhang et al. [31] addressed complex event processing over event streams with imprecise timestamps. In [31], a time interval is used to bound the occurrence time of each event and the timestamps of different events are assumed to be independent of each other. In other words, any two events are allowed to happen at the same time instance. As discussed earlier in Section 1, this is not reasonable in many real applications. On the contrary, we consider the situations that some events have temporal dependencies. This subtle difference brings significant challenges to the searching of possible orders of events and the computation of the corresponding probabilities. In addition, the approach in [31] assumes attribute values are precise, while our work does not make such assumptions.

In addition, most of the above papers focused on real-time event streams, where event processing is often constraint within a relatively short time window. Our paper considers archived event data. In particular, we address the challenges of pre-processing and indexing event data in order to improve query performance.

2.2 Query processing on time series data

The problem of finding patterns in a time series database has been well studied [9, 13, 17, 18]. However these works were not proposed in the context of event processing and none of them considers temporal uncertainty of time series data.

More recently, more attention has been paid to uncertain time series, trajectories and data streams. Most of these works addressed either probabilistic range queries [25, 32] or probabilistic similarity queries [6, 21, 30]. In other words, they mostly focused on the uncertainty of attribute values of the data events.

2.3 Probabilistic temporal databases

Dyreson and Snodgrass are among the first to introduce the valid-time indeterminacy in temporal databases [12]. They proposed the concept of indeterminate instant, which is known to be located some time within a set of time points. Furthermore Dekhtyar et al. [10] proposed probabilistic temporal databases and capture uncertainties in both valid time and attribute values. However, both of the above works only considered a single "select-from-where" block, while pattern queries in our context are more complicated [31] and hence their techniques cannot be readily employed.

3. PRELIMINARIES

3.1 Data Model

Basically, an *uncertain event* comprises of following information:

- *ID* is a unique identifier of e.

- A unique *tag* indicates the object of the event, where *tag* belongs to a discrete categorical domain $D = \{TAG_1, TAG_2, \ldots\}$.

- A *time interval*, $TI = [TI^-, TI^+]$ ($TI \in T$), that bounds all the possible occurrence times of that event. As in most existing temporal data model, we assume the event sequence has a discrete and totally ordered time domain T, containing instants numbered as $1, 2, \ldots$.

- A set of *attributes* that describes the event. The values of these attributes could be imprecise. Without loss of generality, we represent each attribute of an event as a value range $R = [R^-, R^+]$, associated with a probability density function *pdf* of the possible values in R.

Take an RFID deployment in an office building for example, an event e_0 records the current location of a person TAG_0. The location can be represented by two attributes x and y. TI_0 stands for an uncertain time interval, in which TAG_0's location was recorded.

We assume an event's spatial and temporal uncertainty to be independent, as their causes are often different. In this way, the spatial and temporal uncertainty of a sequence of events can be analyzed independently and easily combined to produce the actual probability of the occurrence of a sequence pattern.

3.1.1 Semantics of Possible Worlds

Given a set of events $\mathbb{S} = \{e_1, \ldots, e_n\}$, each combination of the possible occurrence time gives rise to an instance, also called a possible world. Each instance assigns a specific timestamp ts to each event e_i, where $ts \in e_i.TI$. All these instances comprise the *possible worlds* of $\mathbb{S}$.

As we claimed in Section 1, events cannot be simply viewed as temporally independent. Take the RFID application as an example, we have the following two dependencies: (1) it is impossible that one person appeared at two different locations simultaneously; (2) a person cannot be instantly transported between two faraway locations. Apparently, both dependencies relate to the occurrence time of events and we call them *temporal dependencies*.

To model such dependencies, we propose a concept called *dependency group* and assume that two events belonging to different groups are temporally independent. Consequently, an event set $\mathbb{S}$ can be partitioned into different dependency subsets:

$$\mathbb{S} = \mathbb{S}_1 \oplus \mathbb{S}_2 \oplus \cdots \oplus \mathbb{S}_m \quad (1)$$

where $\mathbb{S}_k$ is a subset of $\mathbb{S}$ and events within $\mathbb{S}_k$ compose a dependency group G_k. $\oplus$ is an operator to combine two event subsets into a single event set.

In order to handle the dependencies, we enforce that each instance of $\mathbb{S}$ satisfies the following two conditions: for every pair of events from $\mathbb{S}_k$, denoted as $(e_1, tag_1, [t_1, t_2], x_1, y_1)$ and $(e_2, tag_2, [t_3, t_4], x_2, y_2)$ respectively,

Condition 1. $e_1.ts \neq e_2.ts$; and

Condition 2. $f(x_1, y_1, ts_1, x_2, y_2, ts_2) \propto \theta$.

Condition 1 prohibits events belonging to one dependency group occur at the same time. **Condition 2** is a generalization of dependency (2), where $f(x_1, y_1, ts_1, x_2, y_2, ts_2)$ is a user-defined function and θ is a user specified value. For instance, assuming that $tag_1 = tag_2$, (x, y) represents the location of a person, and θ represents the maximum moving speed of a person. Suppose

$$f(x_1, y_1, ts_1, x_2, y_2, ts_2) = \frac{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}{|e_1.ts - e_2.ts|},$$

where $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ is the Euclidean distance between (x_1, y_1) and (x_2, y_2). Then the second condition can guarantee that each instance of $\mathbb{S}$ obeys the second dependency.

3.1.2 *Order Instances*

According to our model, as shown in equation (1), an event set consists of several dependency groups. To simplify our discussion, we just describe the processing of one dependency group $\mathbb{S}_k$, and all results can be generalized to multiple dependency groups. Apparently, a possible world of $\mathbb{S}_k$ that satisfies **Condition 1** actually assigns a total ordering to the events in $\mathbb{S}_k$. We call such a possible world an *order instance*, denoted as OI. In other words, an order instance of $\mathbb{S}_k$ assigns a distinct timestamp to each event according to **Condition 1** and hence defines a precedence relation over all the events in $\mathbb{S}_k$.

Table 1 shows an example of the possible timestamps of events belonging to $\mathbb{S}_k$. There are 720 possible combinations of the timestamps in total, but only 10 are order instances, as listed in Table 2. Therefore, the order instances are a subset of the possible worlds. In other words, the actual possible timestamps of an event may be only a subset of *e.TI*. Take e_1 in Table 2 for instance, e_1 can only occur at timestamp 1 in $\mathbb{S}_k$'s order instances, even though 2 and 3 could be possible timestamps.

Table 1: An example event sequence $\mathbb{S}_k$

Time	1	2	3	4	5	6	7
e_1	⋆	⋆	⋆				
e_2		⋆	⋆				
e_3		⋆	⋆				
e_4			⋆	⋆	⋆	⋆	⋆
e_5				⋆	⋆	⋆	
e_6					⋆	⋆	
e_7						⋆	⋆

3.2 Pattern Queries

A pattern query consists of a pattern structure, a time window and a confidence constraint.

Pattern Item. A pattern structure is defined by a sequence of pattern items, where each pattern item is defined by several predicates over the specific event attributes. Let $\mathcal{E} = \{E_1, E_2, \dots\}$ be a set of pattern items appearing in the pattern structure. Given an event e, e matches a pattern item E_i iff the attributes of e satisfy the predicates of item E_i.

Table 2: All order instances of $\mathbb{S}_k$

Time	1	2	3	4	5	6	7
OI_1	e_1	e_2	e_3	e_4	e_5	e_6	e_7
OI_2	e_1	e_2	e_3	e_4	e_6	e_5	e_7
OI_3	e_1	e_2	e_3	e_5	e_4	e_6	e_7
OI_4	e_1	e_2	e_3	e_5	e_6	e_4	e_7
OI_5	e_1	e_2	e_3	e_5	e_6	e_7	e_4
OI_6	e_1	e_3	e_2	e_4	e_5	e_6	e_7
OI_7	e_1	e_3	e_2	e_4	e_6	e_5	e_7
OI_8	e_1	e_3	e_2	e_5	e_4	e_6	e_7
OI_9	e_1	e_3	e_2	e_5	e_6	e_4	e_7
OI_{10}	e_1	e_3	e_2	e_5	e_6	e_7	e_4

Each predicate can be modeled as a value range of a particular event attribute. Hence a pattern item E_i can be represented as an aggregated value range over all the event attributes, denoted by R_{attr}.

Furthermore, since each attribute associates with a probability distribution function (*PDF*), we can only assert that e matches E_i with a probability, denoted as $Pr(E_i|e)$.

Pattern Structure. A pattern structure defines a sequence of pattern items occurring in a sequential order. A primitive pattern $\mathbb{P}$ contains only one pattern item. A composite pattern is composed by one or more primitive patterns with the following operators:

- **Sequence operator:** seq. This is a binary operator. The concatenation of two sequences S_1 and S_2 matches $seq(\mathbb{P}_1, \mathbb{P}_2)$ iff S_1 and S_2 match $\mathbb{P}_1$ and $\mathbb{P}_2$, respectively, and all the events in S_2 occur after those in S_1.
- **Negation:** $\neg$. A possible sequence S matches $\neg\mathbb{P}$, iff there is no subsequence of S that matches $\mathbb{P}$.

Constraints. A pattern query associates with two constraints, i.e. a time constraint L and a confidence Θ. The time constraint requires that all matched events should occur within a time interval with length L. Furthermore, the confidence constraint Θ is a value within $(0, 1]$. The **confidence** of a match is the sum of the probabilities of all matched instances in the possible worlds. All matches with confidence less than Θ will be eliminated from the query results.

4. INDEX STRUCTURE

An straightforward way to process a pattern query over an event sequence is to enumerate the possible worlds and then find out the matched instances by exhaustive search. However, the cost of such a method is prohibitive since the cardinality of the possible worlds is usually huge.

To speed up the processing of pattern queries, we propose an index structure to enumerate all possible instances in this section. The index is essentially a prefix tree, where each node represents an event in $\mathbb{S}$ and each path from the root to a leaf node is an order instance of $\mathbb{S}$. One can perform pattern matching by searching the pattern within the index.

Unfortunately, the size of such an index is too large to perform an efficient pattern matching, which requires a traversal of potentially many paths. We proposed a strategy to reduce its redundancies. However, the optimized index structure could still be very large. Thus, we introduce an encoding scheme to improve the performance, where each tree node will be labeled with a unique code based on the topology structure of the index. Thereafter, pattern matching can be performed efficiently by using the codes rather than by traversing the tree structure.

4.1 OI-tree

As discussed earlier, one major challenge of the problem is the temporal dependency of the events in each dependency group. To facilitate the analysis of the possible worlds, we build an OI-tree (Order Instance Tree) for each dependency group to capture the possible orderings of the events in the group. Subsequently, we use the notion $\mathbb{T}$ to denote an OI-tree and $\mathbb{S}_k$ to denote the set of events belonging to the kth dependency group.

4.1.1 The Single-Tree Index

Our first method, called single-tree index, organizes all the order instances of an event group $\mathbb{S}_k$ into one single OI-tree. Each node in the tree corresponds to one or more possible instances of an event (e) within $\mathbb{S}_k$ and each path from the root node to a leaf node represents one order instance of $\mathbb{S}_k$. Note that a virtual node, i.e. a node without any corresponding event, is used here as the root node. This is to accommodate the cases where there are two or more order instances with different starting events. Figure 1 shows the OI-tree of our running example $\mathbb{S}_k$ (Table 1).

In the construction of the OI-tree, we filter out those impossible order instances by checking the two aforementioned dependency conditions. So all the instances compose a OT-tree satisfy dependency conditions. As we can see later, based on the OI-tree, events can be encoded into integers to enhance the efficiency of pattern matching.

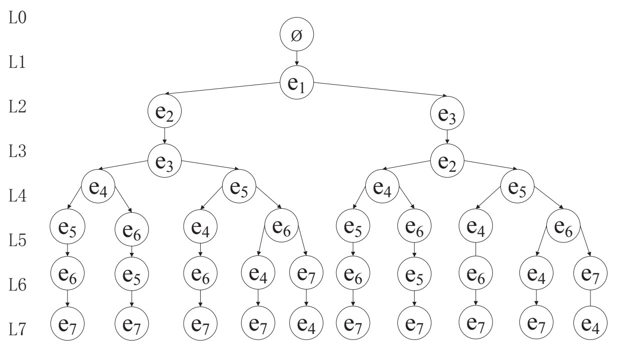


Figure 1: The Single-Tree Structure

Details about the tree construction is given in Algorithm 1. In this algorithm, each node has a list L_{nap} for keeping track of events that have not appeared on the path from the root node to the current node.

Initially, we generate a virtual root node N_{root} (lines 2–4). N_{root} is at level 0 and does not correspond to any event. Then we generate the OI-tree level by level. Based on a node N at level i, we generate its child nodes at level $i+1$ corresponding to each event in $N.L_{nap}$. In each step, we create a new node N_{new} for each possible time instant of e in $N.L_{nap}$ and update $N_{new}.L_{nap}$ subsequently. If any event e' in $N_{new}.L_{nap}$ has no possible time after the time instant of N_{new}, we discard N_{new}, because e' does not appear in the path from N_{root} to N_{new} and cannot appear in the subtree of N_{new} either. Otherwise, N_{new} becomes a child of N (lines 5–18).

Line 11–16 is used to eliminate paths (event sequences) that do not satisfy **Condition 2** (Section 3.1.1). Node N_{new} will be added into the path only if event e and its preceding nodes satisfy **Condition 2**, where e^j is the *j-th* preceding event of e. Note that as in each path all events have different time stamps, **Condition 1** is

Algorithm 1: Build OI-tree($\mathbb{S}_k$)

```
1  Create an empty tree node list $L_{node}$;
2  $N_{root}.L_{nap} \leftarrow$ all events in $\mathbb{S}_k$ ;
3  $N_{root}.level \leftarrow 0$;
4  $L_{node} \leftarrow L_{node} \cup N_{root}$;
5  for each level i from level 1 to level $|\mathbb{S}_k|$ do
6      for each node $N$ in $L_{node}$ with $N.level = i - 1$ do
7          for each event $e$ in $N.L_{nap}$ do
8              Create a new node $N_{new}$ corresponding to each
               possible time of $e$;
9              $N_{new}.L_{nap} \leftarrow N.L_{nap} - e$;
10             if no event $e'$ in $N_{new}.L_{nap}$ has
               $e'.TI^+ \leq N_{new}.time$ then
11                 $j \leftarrow 0$ ;
12                 do
13                     $e^j \leftarrow L_{node}[L_{node}.size - j]$ ;
14                     $j \leftarrow j + 1$ ;
15                 while $f(x_1, y_1, ts_1, x_2, y_2, ts_2) \propto \theta$ holds for
                   $e$ and $e^j$;
16                 if $j$=$i$ then
17                     $N_{new}$ becomes a child node of $N$;
18                     $L_{node} \leftarrow L_{node} \cup N_{new}$;
19 Clear $L_{node}$;
20 return $L_{node}$;
```

inherently satisfied.

Complexity analysis: Suppose n is the number of events in the dependency group which is actually the hight of the OI-tree. At each level i, we need to enumerate all events with an occurrence instant at i. Let λ_i be the number of events that are possible to occur at time instant i, then the cardinality of enumeration is $\prod_{i=1}^{n} \lambda_i$. Let κ be the maximum value in $\{\lambda_1, \ldots, \lambda_n\}$, which is a constant. Then $\prod_{i=1}^{n} \lambda_i \leq \kappa^n$. Thus the complexity of the single-tree construction is $O(\kappa^n)$. Our experimental results (as illustrated in Table 5 in Section 6) also verify that this algorithm has an exponential complexity.

4.1.2 The Multi-Tree Index

Note that each path in an OI-tree corresponds to an order instance. Since a subsequence could be shared by multiple order instances, it would repeatedly appear in the OI-tree multiple times. From Figure 1, we can see that starting from level 3, the left half and the right half of the tree are exactly the same. Such redundancy increases the complexity of not only the index construction but also the pattern matching algorithm.

Therefore, we introduce a **multi-tree index** to compress the OI-tree obtained by the single tree method. Basically, we divide $\mathbb{S}_k$ into isolated subsequences, and then construct a smaller OI-tree for each isolated subsequence. All these OI-trees together represent the possible orderings of $\mathbb{S}_k$. An isolated subsequence is defined as follows.

DEFINITION 1. *Let S.TI denotes the time interval of a subsequence (S) of an event set $\mathbb{S}_k$. Then S is an* isolated subsequence *of $\mathbb{S}_k$ iff in all order instances of $\mathbb{S}_k$, all events in S can never occur at an instant t such that $t \notin$ S.TI and all events not in S can never occur at an instant t' such that $t' \in$ S.TI.*

Going back to our running example (Table 1), we can see that the whole $\mathbb{S}_k$ is a sequence and also an isolated subsequence. Therefore, an OI-tree, denoted as T_0, has to be built on the whole sequence. According to the definition, $\mathbb{S}$ can be divided into the following isolated subsequences: $IS_1 = \{(e_1, [1, 3])\}$, $IS_2 = \{(e_2, [2, 3]), (e_3, [2, 3])\}$ and $IS_3 = \{(e_4, [3, 7]), (e_5, [4, 6]), (e_6, [5, 6]), (e_7, [6, 7])\}$. For each isolated subsequences, as shown in Figure 2, we construct one OI-tree, i.e. T_1, T_2 and T_3, for IS_1, IS_2 and IS_3 respectively. We call T_0 the **parent OI-tree** of T_1, T_2 and T_3, which are then called the **child OI-trees** of T_0.

The compression procedure continues by recursively detecting the isolated subsequences within a subtree of an OI-tree index. For instance, inside T_3 (IS_3), the subtree rooted at $e4$ corresponds to subsequence $S' = \{(e_5, [5, 6]), (e_6, [5, 6])(e_7, [6, 7])\}$. Obviously, S' has two isolated subsequences: $IS'_1 = \{(e_5, [5, 6]), (e_6, [5, 6])\}$ and $IS'_2 = \{(e_7, [7, 7])\}$. Correspondingly, two OI-trees T_4 and T_5 are constructed for IS'_1 and IS'_2 respectively. The final multi-tree index is constructed as shown in Figure 2. T_3 is called the **parent OI-tree** of T_4 and T_5, whereas T_4 and T_5 are called as **child OI-tree** of T_3. In an OI-tree, we call nodes correspond to child OI-trees as **virtual nodes** and the others as **real nodes**.

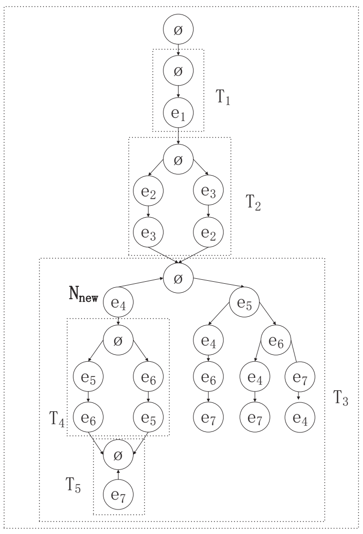


Figure 2: An example of order instance trees.

Finding isolated subsequences. Now we present the algorithm to find isolated subsequences. First, we sort the events in $\mathbb{S}_k$ in ascending order of the lower bounds of their time intervals. Then we decide if $\mathbb{S}_k$ can be divided into several isolated subsequences in the following two phases.

Step 1: Searching for *splitting instants* by scanning an input sequence S. The definition of a splitting instant is given as follows.

DEFINITION 2. *Given a sequence S, an instant $\hat{t}$ is a splitting instant iff, for any event e in S, we have $e.TI^+ \leq \hat{t}$ or $e.TI^- > \hat{t}$.*

When a splitting instant $\hat{t}$ is found, $\mathbb{S}_k$ will be divided into two isolated subsequences IS_1 and IS_2, where all events in $\mathbb{S}_k$ with $e.TI^+ \leq \hat{t}$ fall into IS_1 and other events belong to IS_2. Therefore, IS_1 and IS_2 have time intervals $[\mathbb{S}_k.TI^-, \hat{t}]$ and $[\hat{t} + 1, \mathbb{S}_k.TI^+]$ respectively.

Step 2: For each isolated subsequence IS acquired in the previous step, we further divide it based on the following lemma.

LEMMA 1. *Given a sequence S, let $[t^-, t^+]$ be the minimum time interval bounding the time intervals of n events in S. If $[t^-, t^+]$ contains exactly n timestamps, then these n events compose an isolated subsequence.*

In particular, we scan events of each IS one by one. For the *i-th* event e_i, we check if any *j-th* event e_j ($j < i$) has a time interval $e_j.TI$ so that $[e_j.TI^-, e_i.TI^+]$ contains exact $i - j + 1$ events. If so, event e_j to event e_i compose an isolated subsequence IS_1 and all other events falls into another isolated subsequence IS_2. After that we update the time intervals of events in IS_2, and then divide IS_2 further by repeating the process from Step 1. This process stops when no subsequences can be further divided in either step.

Constructing OI-tree for each isolated subsequence. The isolated subsequence based OI-tree construction is similar to Algorithm 1. The only difference is that every time we create a new node N_{new}, we get a sequence S, which contains all events in $N_{new}.L_{nap}$. Then we check if S can be divided into several isolated subsequences as described above. If yes, we create an OI-tree for each new isolated subsequence in exactly the same way and then we treat each acquired OI-tree as a virtual node in the current OI-tree.

Time complexity: The multi-tree index is based on following observations: (1) For each event, its possible instant interval *TI* is very limited; (2) Events within a dependency group can be further divided into isolated subsequences (IS), where event instances in an IS are correlated, but those belonging to different IS are independent. Here event instance refers to an event associated with one of its possible instant. In practice, we expect that event instances within $\mathbb{S}_k$ can be naturally divided into distinct isolated subsequences.

Suppose that $\mathbb{S}_k$ can be divided into m isolated subsequences and the total number of event instances are $f \cdot n$, where n is the number of events in $\mathbb{S}_k$. Consequently, the average number of event instances in each isolated subsequence is $\frac{f \cdot n}{m}$. Furthermore, we assume an event in each isolated subsequence has at most c instances and hence each isolated subsequence has at most $\frac{f \cdot n}{c \cdot m}$ events.

Note that for each isolated subsequence, we have to construct an OI-tree. So the time complexity of the construction of multi-tree index is $O(\kappa^{\frac{f \cdot n}{c \cdot m}} \cdot m)$. If each isolated subsequence contains at most a constant number of event instant, say k. In other words, we assume $\frac{f \cdot n}{c \cdot m} = k$. Then the time complexity of the multi-tree approach is $O(\kappa^k \cdot m)$, i.e. a linear complexity.

We expect that in practice most isolated subsequences should have a small number of events and hence can be bounded by a constant as analyzed above. We have experimented different distributions of possible instant intervals of events in Section 6 (Figure 5). The results indicate that the multi-tree algorithm has a linear complexity.

4.2 Encoding The OI-Tree

Each node in an OI-tree is assigned a unique code, which is the sum of two parts, i.e. a local relative value v_{rel} and a base value v_{base}.

The local relative value. The local relative value v_{rel} indicates a node's relative position inside its OI-tree and it can be generated as follows. In an OI-tree $\mathbb{T}$, each child OI-tree within it will collapse into a virtual node, such as T_4 and T_5 in Figure 3 are considered as a node in T_3. Assuming f is the maximum fanout of $\mathbb{T}$, we first extend $\mathbb{T}$ to a complete f-tree (denoted as $\mathbb{T}^f$) by inserting several empty nodes into $\mathbb{T}$, as shown in Figure 3.

Each node in $\mathbb{T}^f$ is given a local relative value v_{rel} so that the node with v_{rel} is the $(v_{rel} + 1)$th node visited when we traverse $\mathbb{T}^m$ using *Breath-First Search*(BFS).

For instance, the extended complete binary tree of $\mathbb{T}_3$ is shown in Figure 3. The blank nodes are the nodes inserted when extending $\mathbb{T}_3$ to a complete binary tree $\mathbb{T}_3^f$. The numbers beside the nodes are their local relative values.

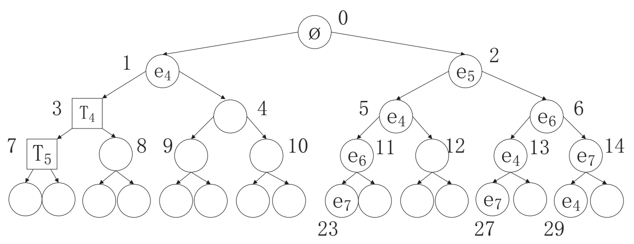


Figure 3: The extended complete binary tree of $\mathbb{T}_3$.

The base value. The base value of a node is generated based on the following rules.

1. The root node of OI-tree T_0 is assigned a base value of 0.
2. For a virtual node T_i, with a parent OI-tree as T_j, its base value equals to its local relative value in T_j, i.e. $T_i.v_{base} = T_i.v_{rel}$.
3. For a real node N of OI-tree T_j, let N' denotes the preceding node of N in BFS, then its base value is calculated according to the following two cases. (1) If N' is a virtual node (a child OI-tree of T_j), then $N.v_{base}$ equals to the largest code of the nodes inside the tree represented by N'. (2) Otherwise, $N.v_{base}$ equals to $N'.v_{base}$.

Example. *We further explain the generation of node codes using the running example. Inside the first OI-tree $\mathbb{T}_0$, initially the root node N_{root} has a base value 0. Since the fanout is 1, the four nodes N_{root}, $\mathbb{T}_1$, $\mathbb{T}_2$ and $\mathbb{T}_3$ have local relative values 0, 1, 2 and 3 respectively. Then because the node before the virtual node of $\mathbb{T}_1$ is a real node N_{root}, the base value of $\mathbb{T}_1$ is equal to the base value of N_{root}. So the final code of the virtual node of $\mathbb{T}_1$ is $0 + 1 = 1$.*

Then inside $\mathbb{T}_1$, the root node gets a base value that is equal to the final code of the virtual node in $\mathbb{T}_0$ that corresponds to $\mathbb{T}_1$, which is 1. Given the local relative value 0 and 1 of the two nodes in $\mathbb{T}_1$, their final codes are 1 and 2 respectively.

The third node in $\mathbb{T}_0$, i.e. the virtual node of corresponding to $\mathbb{T}_2$, gets a base value as the largest final code in $\mathbb{T}_1$, which is 2. So the final code of this virtual node is $2 + 2 = 4$, which then becomes the base value of the root node inside $\mathbb{T}_2$.

Codes of the other nodes are generated similarly. The final codes of the running example are shown in Figure 4, where each entry denotes a node in the OI-tree and the code of each node is given in the form of $v_{base} + v_{rel}$.

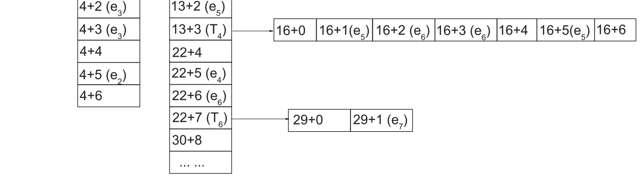


Figure 4: Codes of the running example.

In sequence pattern matching, the essential problem is to judge the temporal occurrence sequence of events. For example, a simple pattern query (E_1, E_2) require $e_1.ts < e_2.ts$ in each of the matching sequence. Such temporal relationship can be easily acquired by comparing the corresponding nodes' codes, as their codes uniquely determined their positions in the OI-tree. To judge the temporal relationship between a pair of nodes N_i and N_j corresponding to events e_i and e_j, we just need to judge whether $N_j.code$ falls into $[N_i.min, N_i.max]$, where $N_i.min$ and $N_i.max$ are the minimum and maximum codes in the tree rooted at N_i.

More specifically, we can derive the following two lemmas for our coding scheme.

LEMMA 2. *Given a complete f-tree $\mathbb{T}^f$ and a node N with local relative value v_{rel}, the descendant nodes of N must have local relative value within ranges $[f \cdot v_{rel} + 1, f \cdot v_{rel} + f + 1]$, $[f^2 \cdot v_{rel} + f + 1, f^2 \cdot v_{rel} + f^2 + 1]$, $[f^3 \cdot v_{rel} + f + 1, f^3 \cdot v_{rel} + f^3 + 1]$ and so on.*

LEMMA 3. *Given a complete f-tree $\mathbb{T}^f$ and a node N with local relative value v_{rel}, the ancestor nodes of N must have local relative value equalling $\lfloor \frac{v_{rel}-1}{f} \rfloor$, $\lfloor \frac{v_{rel}-1}{f^2} \rfloor$, etc.*

The proof of Lemma 2 and Lemma 3 is straightforward, so we omit it for brevity.

Note that the table in Figure 4 does not need to be stored in memory or on disk. Only the codes of nodes corresponding to some events are indexed and stored in a way described in the next section. In addition, we maintain the following information of OI-trees in memory to enhance the above operations: (1) $\mathbb{T}$.*TI*: the time interval of $\mathbb{T}$; (2) $\mathbb{T}.f$: the maximum fanout; and (3) $\mathbb{T}.CR$: the code range of $\mathbb{T}$. The information of the running example is shown in Table 3.

Table 3: The information of order instance trees of $\mathbb{S}_k$.

	TI	fanout	code range
$\mathbb{T}_0$	[1,7]	1	[0 , 60]
$\mathbb{T}_1$	[1,1]	1	[1 , 2]
$\mathbb{T}_2$	[2,3]	2	[4 , 10]
$\mathbb{T}_3$	[4,7]	2	[13, 60]
$\mathbb{T}_4$	[5,6]	2	[16, 22]
$\mathbb{T}_5$	[7,7]	1	[29, 30]

4.3 Indexing The Content of Events

Note that in the OI-tree of each group $\mathbb{S}_k$, an event would correspond to multiple nodes, which are kept in a node list L_{node} . Each node N in L_{node} is represented by $(N.time, N.code, N.poi)$. $N.poi$ is the probability for the occurrence of order instances in the possible worlds that involve N. We use $CR = [code_{min}, code_{max}]$ to denote the code range of all the nodes in L_{node}, where $code_{min}$ and $code_{max}$ stand for the minimum and maximum codes of the nodes within L_{node} respectively.

Combining with other information of an event defined in Section 3.1, such as its group id (k), time interval(*TI*), and attributes ($ATTR$), it can be represented by a $(d + 3)$-dimensional tuple $\{[k, k], TI, CR, R_1, \ldots, R_d\}$, where k, *TI*, CR, and $R_j (j = 1, \ldots, d)$ stand for the group ID, time interval, code range and attribute value ranges of an event respectively.

To support the efficient evaluation of pattern queries, we employ a multi-dimensional index structure to index all of the aforementioned $(d + 3)$-dimensional tuples. In principle, as our evaluation algorithms do not depend on a certain type of index, we can choose a multi-dimensional index according to the application scenarios, such as the data distribution, the value of d, etc. In our experiments, we use R*-tree as our index structure.

5. MATCHING EVENT PATTERNS

In this section we present the details of our pattern query processing algorithms. In particular, we first introduce a sequential evaluation approach in Section 5.1 and then an optimization is given in Section 5.2.

Algorithm 2: BasicQueryProcessing($\mathbb{P}$, L, Θ)

```
1  Create an empty list $L_{cand}$;
2  $S \leftarrow$ RangeQuery($[0,\infty]$, $[0,\infty]$, $[0,\infty]$, $E_1.R_{attr}$);
3  for each event e in $S$ do
4      for each node $N$ in $e.L_{node}$ do
5          $L_{cand} \leftarrow L_{cand} \bigcup (\{e\}, [N.time, N.time],$
           $\{N_k^{\vdash} = N\}, N.poi, Pr(e \in E_1))$;
6  while there are incomplete entries in $L_{cand}$ do
7      $EN \leftarrow$ one incomplete entry in $L_{cand}$;
8      $L_{cand} \leftarrow L_{cand} - EN$;
9      $i \leftarrow$ the number of events in $EN.CurMatch$;
10     Create an empty list $S$;
11     for each group of $\mathbb{S}$ do
12         $DesCR_k \leftarrow$ GetDescCR($N_k^{\vdash}.code$);
13         $S \leftarrow S \bigcup$ RangeQuery($[k,k]$,
           $[EN.t^{+}+1, EN.t^{-}+L]$, $DesCR_k$, $E_{i+1}.R_{attr}$);
14     for each event e in $S$ do
15         for each node $N$ in $e.L_{node}$ do
16             if $N.time \in [EN.t^{+}+1, EN.t^{-}+L]$ and
               $N.code \in DesCR_k]$ then
17                 Create new entry $EN_{new}$ for $N$;
18                 if $EN_{new}.PerOI \cdot EN_{new}.p \geq \Theta$ then
19                     $L_{cand} \leftarrow L_{cand} \bigcup EN_{new}$;
20 return $L_{cand}$;
```

5.1 The Sequential Evaluation

We first consider the pattern query with a pattern structure as a sequence of simple event types. The condition range of an event type is denoted as R_{con}. Given a pattern structure $\{E_1, E_2, \ldots, E_\kappa\}$, with a time constraint L and a probability constraint Θ, the query processing algorithms are given in Algorithm 2. The algorithm can be extended in a straightforward way to handle pattern structures involving negation operators.

Example. *We use our running example to illustrate the query processing algorithm. For the ease of explanation, we assume there is only one dependency group* $\mathbb{S}_k$. *The value ranges of event attributes in* $\mathbb{S}_k$ *are listed in Table 4. Consider an example pattern query with a time constraint* $L = 6$ *and the query structure* $\mathbb{P} = \{A, B, C\}$, *where the predicates of the items are represented as the following value ranges over the two attributes:* $A.R_{attr} = \{[0,2], [8,10]\}$, $B.R_{attr} = \{[8,10], [6,8]\}$ *and* $C.R_{attr} = \{[5,7], [5,7]\}$.

Table 4: The value ranges of event attributes in $\mathbb{S}_k$.

	e_1	e_2	e_3	e_4	e_5	e_6	e_7
d_1	$[0,2]$	$[2,4]$	$[0,1]$	$[8,9]$	$[4,6]$	$[0,2]$	$[4,6]$
d_2	$[5,8]$	$[3,8]$	$[9,10]$	$[7,8]$	$[5,7]$	$[0,1]$	$[4,6]$

In the algorithm, we maintain a candidate list L_{cand} in memory, where each entry in L_{cand} contains information for a partially matched sequence and is in the form of $EN = (CurMatch$, $[t^-, t^+]$, $LastNodes$, $PerOI$, $P)$. In particular, $CurMatch$ is a sequence of matching events found so far. $[t^-, t^+]$ is the time interval within which events in $CurMatch$ occur. $LastNodes = \{N_1^{\vdash}, N_2^{\vdash}, \ldots, N_{G_n}^{\vdash}\}$ is a list of nodes, where $N_i^{\vdash}$ records the node corresponding to the last event in the ith dependency groups of $\mathbb{S}$ contained by $CurMatch$. $PerOI$ is the percentage of total number of order instances containing the sequence $CurMatch$ and P is the product of the probability of each event in $CurMatch$ belonging to its corresponding pattern item. Specially, if the $CurMatch$ of an entry EN contains less than γ events, we call EN an *incomplete entry*. Otherwise, it is called a complete entry.

The whole query processing algorithm work in two steps.

Step 1. Searching for the first pattern item. Initially, we issue a range query to find all events that match E_1 via the multidimensional index (line 2). For each returned event e, we create a new entry for each of e's corresponding nodes (lines 3–5).

Coming back to our example. In step 1, we search for events matching item A by issuing a range query $([0,\infty], [0,\infty], [0,\infty], [0, 0.2], [0.8, 1])$. *An event* e_3 *is retrieved. There are two nodes in* $e_3.L_{node}$, *with timestamps 2 and 3, and codes 6 and 7, respectively. We create a new entry in* L_{cand} *based on each node. Specifically, for the node* N *corresponding to timestamp 2, we have an entry* $EN = \{\{e_3\}, [2,2], \{N_k^{\vdash}\}, PerOI = 0.5, P = 1\}$, *where* $N_k^{\vdash} = N$.

Step 2. Searching for the next pattern item. For each incomplete entry EN in L_{cand}, suppose E_i is its last matched item, i,e. EN has already i matched pattern items, then the next pattern item E_{i+1} can be recognized as follows. In particular, the next matching event e must satisfy the following requirements:

1. $Pr(E_{i+1}|e) > 0$, it means each attribute range of e must intersect with the condition range of E_{i+1}.
2. Event e is possible to appear after all the events in EN and fulfill the time constraint L, which means the occurrence times of e fall into $[EN.t^+, EN.t^- + L]$.
3. Suppose e belongs to the kth dependency group $\mathbb{S}_k$. If there are other events from the same group included in the already matched events, then there is at least one order instance of $\mathbb{S}_k$ such that e occurs after all those events. It means at least one node of e in the OI-tree appear in the subtree of $LastN_k$, the last matched events from $\mathbb{S}_k$.

For the last requirement, given the last node $N_k^{\vdash}$ corresponding to each group $\mathbb{S}_k$, we compute a range (denoted as $DesCR_k$) of codes of the nodes that are inside the subtree of $N_k^{\vdash}$ in the OI-tree corresponding to $\mathbb{S}_k$. Then if one node N is in the subtree of $N_k^{\vdash}$, $N.code$ must be inside $DesCR_k$. $DesCR_k$ is computed by a function $GetDescCR$ (called line 12), which, as described later, will take use of Lemma 2 for this purpose.

Then we issue range queries to the multidimensional index to search for qualified events and create new entries according to each returned event. If the new entries are corresponding to confidences larger than Θ, we insert them into L_{cand} (line 14–19).

The whole process is completed by iteratively repeating **Step 2** until all entries in L_{cand} are complete. Then based on each entry $EN = (CurMatch, [t^-, t^+], LastNodes, PerOI, P)$, we get a match result $CurMatch$, with time range $[t^-, t^+]$ and a confidence of $PerOI \cdot P$.

Let us look at our example again. Based on the code of $N_k^{\vdash}$ *in the previously mentioned entry* EN, *we compute the code range* $DesCR_k$ *as* $\{[9,10], [13,60]\}$. *Then we search for the next event matching item B by issuing two range queries* $([k,k], [3,7], [9,10], [0.8,1], [0.6,0.8])$ *and* $([k,k], [3,7], [13,60], [0.8,1], [0.6,0.8])$. *An event* e_4 *is returned, which has a corresponding node* N' *with* $N'.code = 14$ *and* $N'.time = 4$. *Then we have a new entry* $EN' = \{\{e_3, e_4\}, [2,4], \{N_k^{\vdash}\}, PerOI = 0.2, P = 1\}$, *where* $N_k^{\vdash} = N'$.

Based on EN' *we search for the next pattern item C in a similar way and get an complete entry* $\{\{e_3, e_4, e_7\}, [2,7], LastNodes,$

$PerOI = 0.2$, $P = 0.25\}$. The matched subsequence (e_3, e_4, e_7) *falls in* $[2, 7]$ *and has a confidence of* $0.2 * 0.25 = 0.05$.

Algorithm 3: GetDescCR($N_k^{\vdash}$)

```
1  $DesCR_k \leftarrow \emptyset$ ;
2  $S \leftarrow$ all of OI-trees with $CR$ covering $N_k^{\vdash}.code$ ;
3  for each OI-tree $\mathbb{T}$ in $S$ do
4      if $\mathbb{T}$ is the smallest OI-tree to contain $N$ then
5          $code \leftarrow N_k^{\vdash}.code$;
6      else
7          $\mathbb{T}' \leftarrow$ the largest tree containing $N$ and in $\mathbb{T}$;
8          $code \leftarrow \mathbb{T}'.CR^-$;
9      $S' \leftarrow$ all OI-trees with $CR$ in $[\mathbb{T}.CR^-, code]$;
10     if $S' = \emptyset$ then
11         $v_{base} \leftarrow \mathbb{T}.CR^-$;
12     else
13         $v_{base} \leftarrow$ the maximum upper bound of code ranges of
           OI-trees in $S'$;
14     $v_{rel} \leftarrow code - v_{base}$;
15     $R_{rel} \leftarrow$ ranges of relative values of descendant nodes of
       $N_k^{\vdash}$;
16     Add base values to the boundaries of ranges in $R_{rel}$;
17     $DesCR_k \leftarrow DesCR_k \bigcup R_{rel}$;
18 return $DesCR_k$;
```

Computing $\mathbf{DesCR_k}$. The code range of descendant nodes of a given node $N_k^{\vdash}$, it compute by $GetDescCR(\ )$, as shown in Algorithm 3. $DesCR_k = [0, \infty]$ if $N_k^{\vdash} = NULL$. Otherwise, we compute $DesCR_k$ in the following way.

Initially, we set $DesCR_k = \emptyset$. Then based on the information of all OI-trees corresponding to $\mathbb{S}_k$ (as shown in Table 3), we find all the OI-trees containing $N_k^{\vdash}$ and then compute a code range corresponding to each of these OI-trees. $DesCR_k$ is set as the combination of the acquired code ranges.

Particularly, for each OI-tree $\mathbb{T}$ containing $N_k^{\vdash}$, we can easily deduce the base value and local relative value of $N_k^{\vdash}$ (lines 4-14). Then we can deduce the ranges of local relative values of the descendant nodes of $N_k^{\vdash}$ based on Lemma 2.

Based on Lemma 2, given v_{rel}, we derive a set of code ranges (R_{rel}) bounding all the local relative values of $N_k^{\vdash}$'s descendants in $\mathbb{T}$. For each range in R_{rel}, we obtain the final code range by adding a base value on its boundaries. The base value can be computed in a similar way as generating the global base value described in Section 4.2.

5.2 Optimizations

The sequential evaluation of $\mathbb{P} = \{E_1, E_2, \ldots, E_n\}$ is carried out according to the occurrence order of of the pattern items, i.e. E_1 is matched first and then followed by E_2, and so on, until the last item has been matched. This can actually be further optimized by fine-tuning the matching order of items and hence eliminating the unqualified entries from the candidate list L_{cand} as early as possible. More specifically, we can choose the matching order according to the expected number of matching instances of each item.

Our optimized solution is carried out as follows. For matching a pattern structure $\mathbb{P} = \{E_1, E_2, \ldots, E_n\}$, we will estimate the expected number of order instances that match each pattern item over the whole time domain and choose the pattern item with the lowest number, say E_i. After matching E_i, we will generate a list of candidate sequences, one for each event instance that matches E_i. Then for each candidate, we re-estimate the expected number of matches of all the unmatched pattern items and again choose the one with the lowest number for the next matching attempt. Note that we have to re-estimate the expected number of matches because the matching conditions are changed due to the fact that some events have already been matched in the candidate sequence. As indicated by our analysis later in this section and also verified by our experimental study, this optimization can significantly cut down the number of candidates and hence reduce the query processing time considerably.

In the rest of this section, we provide more details of this algorithm and present an analysis on its performance.

Getting statistical information. We first extract some statistical information from the whole event set $\mathbb{S}$, which will be used for all future query processing. In particular, we uniformly partition the d-dimensional domain space of the event attributes into a number of cells. In addition, we equally divide the time domain T into a number of time intervals. Then for each cell c_j, we can collect the expected number of events with attributes falling in c_j during time interval I_k, denoted as $p_{j,k}$. All the numbers corresponding to all the cells and time intervals are stored in memory.

Given the above statistical information, we can estimate the number of matches for each pattern item E_i by finding out all the cells that overlap with matching conditions of E_i and then calculate an aggregate value over the numbers of all these cells.

Choosing the first pattern item. Suppose we are to process a pattern query with a pattern structure as $\mathbb{P} = \{E_1, E_2, \ldots, E_n\}$. Initially, for each pattern item E_i, we estimate the expected number of matching events over the whole time domain T and then we choose the one with the minimum number.

Suppose E_i is the chosen pattern item. Then we retrieve all events that match E_i by using the multi-dimensional index. Note that each such event e corresponds to multiple instances due to its temporal uncertainty. So for each node in node list e, i.e. each instance of e, we create a new entry in the candidate list L_{cand}.

In this algorithm, an entry in L_{cand} has to be extended with a list $FirstNodes$ (in addition to the original $LastNodes$ list). The $FirstNodes$ list, as indicated by its name, contains the first matched events of each dependency group in the partially matched sequence of this candidate. This is needed as we may have to search for matching events in both forward and backward directions as described later.

Choosing the next pattern items. Based on each entry EN in L_{cand}, we will choose an unmatched pattern item for the next matching attempt. To re-estimate the expected number of matched events for an unmatched pattern item, we should update its matching time interval based on the matched events in EN.

For an item E_i, we estimate the lower-bound (or upper-bound) of its matching time interval as the upper-bound (or lower-bound) of the occurrence time of the matched events before (or after) E_i. If there is no matched event before (or after) E_i, then the lower-bound (or upper-bound) is set to 0 (or inf). After updating the matching time interval of item E_i, we can estimate its expected number of matched events and choose the one with the minimum number for the next matching attempt.

Suppose E_i is the pattern item that is chosen. Then an event e may match E_i should satisfy the following conditions.

1. The attribute uncertain ranges of e intersect the condition range of E_i.
2. e has a possible occurrence time within the matching time interval of E_i.
3. Suppose e belongs to the kth dependency group of $\mathbb{S}$. Then

if there are other events in the same group in the matching subsequences before and after E_i, denoted as PS_{before} and PS_{after}, respectively, then there should be at least one order instance of $\mathbb{S}_k$ such that at least one node of e (out of the multiple node due to the temporal uncertainty of e) appear in the path from the last node $N_k^{\vdash}$ of PS_{before} to the first node $N_k^{\dashv}$ of PS_{after}.

For the last condition, we can compute a code range CR_k for each group that bounds the codes of nodes in this path. In particular, based on the code of $N_k^{\vdash}$ in PS_{before} and Lemma 2, we compute the code range of all descendants of $N_k^{\vdash}$, denoted as $DesCR_k$. Similarly, based on the code of $N_k^{\dashv}$ in PS_{after} and Lemma 3, we compute the code range of all ancestors of $N_k^{\dashv}$, denoted as $AnCR_k$. Finally, we can get $CR_k = DesCR_k \bigcap AnCR_k$.

With CR_k, we can issue a set of range queries to retrieve the matching events for E_i. Then the candidate entry will be updated. This process will be repeated until all pattern items are matched.

Cost analysis. Next we discuss the improvement that can be achieved by the optimization in comparing to the sequential evaluation in Section 4.1.1. Since the main cost of the query processing algorithms is to search for the next pattern item based on each entry in L_{cand}, the query performance is mainly determined by the number of entries in L_{cand}. So we measure the performance of each query processing algorithm by the total number of entries ever inserted into L_{cand}, denoted as ξ.

To account for the worst case, we consider a query pattern with time constraint equaling the length of the whole time domain and a confidence constraint as 0. Given the pattern structure $\mathbb{P} = \{E_1, \ldots, E_n\}$, every time we search for one pattern item E_i, we create new entries for each possible timestamp of each event e matching item E_i. Let p_i denotes the quantity of events that match E_i. In addition, we assume all events have f possible occurrence instants.

For the sequential evaluation approach, the total number of entries ever inserted into L_{cand}, ξ_{seq}, is computed as follows. As there are p_1 events that match E_1, we create f candidate entries for each of these p_1 events. Therefore, we create $p_1 \cdot f$ entries in the first step.

Thereafter, for the ith pattern item E_i, we create $p_i \cdot f$ entries for each entry already in L_{cand}. So after all n pattern items are found, we have

$$\begin{aligned} \xi_{seq} &= p_1 \cdot f + (p_1 \cdot f) \cdot (p_2 \cdot f) + \cdots + \Pi_{i=1}^{n}(p_i \cdot f) \\ &= \sum_{j=1} n(\Pi_{i=1}^{j}(p_i \cdot f)) \end{aligned} \tag{2}$$

With regard to the optimized approach, we can calculate ξ_{opt} in a similar way. Note that, for every step in this approach, we choose the pattern item that matches minimum number of events. So we can sort p_1 to p_n in ascending order and get a new sequence $\{\hat{p_1}, \hat{p_2}, \ldots, \hat{p_n}\}$. Then we have $\xi_{opt} = \sum_{j=1} n(\Pi_{i=1}^{j}(\hat{p_i} \cdot f))$.

In general, for any other searching order, let E_{r_i} denote the ith pattern items that are chosen, the cost of this order is equal to $\xi_{other} = \sum_{j=1} n(\Pi_{i=1}^{j}(p_{r_i} \cdot f))$. Based on these fomula, we can derive that ξ_{other} is no less than ξ_{opt}. Thus, the searching order introduced in our solution is optimal.

6. EXPERIMENTS

All our experiments are run on a desktop PC with a 2.66GHz CPU and 4GB RAM. The page size is set to 4096 bytes. We use both real and synthetic datasets. Details of the datasets and query generation are given in Section 6.1. We study the behavior of our index schema and the effects of various parameters in Section 6.2. Then we conduct comparative performance study of our query processing algorithms in Section 6.3.

6.1 Experiment Configurations

Real Data. We use three sequential datasets from the UCI machine learning repository. The Dodfers dataset collects the number of cars passing the 101 North freeway in Los Angeles [1]. It contains totally 50,400 tuples, each with one attribute value. The ICU dataset contains 7931 records provided by the Children's Hospital in Boston [2]. Each tuple has two attributes. The third dataset is the localized personal activity data (referred to as LPA for short), which contains totally 164,860 location records, each with 3 attributes [3].

Each tuple (viewed as an event) in these datasets has a timestamp and several attribute values. We generate uncertain event sequence by generalizing their timestamps and attribute values into uncertain ranges.

Synthetic Data. We first generate two certain event sequences. Particularly, we first generate N objects, each with d attribute values. The attributes of objects follow Gaussian distribution. Then we assign a timestamp, which is drawn from a time domain T containing $10N$ time instants, to each object. In one synthetic dataset, the timestamps of objects are randomly chosen from T, while in the other dataset the timestamps of objects are clustered into 10 groups. Within each group, timestamps follow Gaussian distribution around the group center.

After getting the certain event sequences, we convert these certain sequences into uncertain sequences in the same way as described above.

Pattern Queries. For every experiment, we generate 1000 pattern structures and the result presented is the average on 1000 queries, each with one pattern structure.

To generate a pattern structure, we first generate the condition range for each pattern item inside the pattern structure. Each condition range covers 20% of the spatial domain space, with a center randomly chosen from the domain space. Then we randomly assign the operator "$\neg$" and "[ ]" on 10% pattern items of the 1000 pattern structures.

Each pattern query is controlled by the number of pattern items n, the time constraint L, the confidence constraint Θ. We will vary these values to see their impacts on the query performance.

Baseline. Due to the lack of previous work on the problem, we use our less optimized approaches as the baseline methods. For example, we compare the single-tree method with the multi-tree method for the index construction algorithm. With regard to query evaluation algorithms, we compare a tree-traversal method without our tree encoding scheme, the sequential query processing algorithm and the optimized one proposed in this paper.

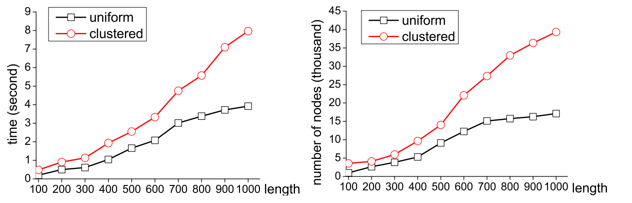
6.2 The Study of The Index

We first study the behavior of our index structure using the two synthetic sequences, with uniform distributed timestamps and clustered timestamps, respectively. We vary two parameters of an event sequence, the length and the number of possible occurrence times of each event, denoted as f. Note that since the construction of the OI-tree of each group are independent, so to study the behavior of the index structure, we assume all events inside the synthetic sequences belongs to the same group.

We measure the index construction time and the size of acquired OI-tree in our experiments. We compare the Multi-Tree approach with the Single-Tree approach in this set of experiments.

Effect of the sequence length. We vary the length of the sequence from 10 to 1000 to see its effect on the index. The number of possible occurrence times (f) of each event is randomly set to a value within $[2, 5]$. Figure 5 demonstrates the test results of the Multi-Tree approach. Both the construction time and number of nodes of OI-trees increase almost linearly with the increase of sequence length.

As for the Single-Tree approach, the construction time and number of nodes increase exponentially as the sequence length increases. For example, given a sequence where events only have two possible occurrence times, the construction time and number of nodes of the Single-Tree approach is shown in Table 5. As demonstrated in the table, it will take almost four days to construct the index for a sequence with a length of 20. We do not draw the results in Figure 5 as they are too large to be shown nicely.



(a) Time (b) Number of nodes

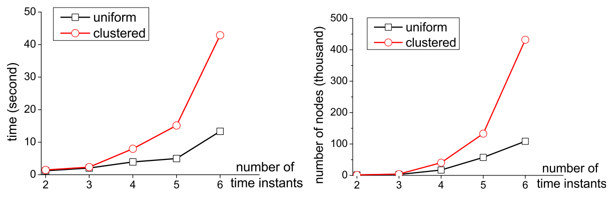
Figure 5: Effect of the length of sequence.

Table 5: The construction cost of the Single-Tree method

length	5	10	15	20
time (second)	0.015	1.944	472.4	3.44E+05 (est)
number of nodes	31	1023	32767	1.05E+06

Effect of the number of occurrence times. We vary the number of time instants of each event from 2 to 6 to see its effect on the index. The sequence length is 1000. Figure 5 demonstrates the test results of the Multi-Tree approach. The construction time and number of nodes increase evidently as f increases. This is because increasing f will increase the fanout of OI-trees, and therefore will increase the size and construction time of the OI-trees.

On the other hand, with a large f, the time intervals of events overlap with each other heavily. Therefore a long event sequence cannot be easily divided into many isolated subsequences, and hence the heights of OI-trees are very large. This is more obvious for the clustered dataset, where time intervals of many events are congregated together.



(a) Time (b) Number of nodes

Figure 6: Effect of the number of time instants.

6.3 Performance Comparison

We compare the query performance of the two methods proposed in Section 5.1 and in Section 5.2 with a *tree traversing* method. In the tree traversing method, we traverse every path of the OI-trees of a given sequence to find the matching subsequences. For the multidimensional index, we employ R*-tree as the underlying structure of our index.

We measure the total entries (in L_{cand}) created during the query processing, the total query processing time and the number of disk page accesses (I/O) in our experiments.

We conduct experiments with the three real datasets. Specially, we only extract the first 5000 events from each dataset. The timestamps in each dataset are normalized into a time domain contains 20000 time instants.

In this set of experiments, we randomly set the number of possible time instants f as a value within $[2, 6]$.

Number of dependency groups. We vary the number of dependency groups from 1 to 8 to see its impact on the query performance. In particular, we partition events in a given sequence into different dependency groups randomly.

The number of pattern items (n) of each pattern query is set to 5 in this set of experiments. The time constraint and confidence constraint are randomly chosen from $[2n, 5n]$ and $[0.6, 0.8]$, respectively.

The test results of the three real datasets are given in Figure 7. As shown in the figure, the number of entries of both methods slightly increases as the number of groups increases. With the optimized method, the number of entries increases much slower than the sequential evaluation.

Then query processing time of both our methods increase linearly as the number of groups increases, while the time cost of the tree traversing method increases exponentially. The speed up of the optimized method compared to the tree traversing is up to 20 times (8 groups, Dodfers dataset).

The number of I/O accesses of both our methods increases as the number of groups increases. We do not measure the I/O cost of the tree traversing method because all tree nodes are cached in memory during the experiments, which is actually in favor of the tree-traversing method.

As shown in the figure, the optimized method also outperforms the sequential one under all circumstances.

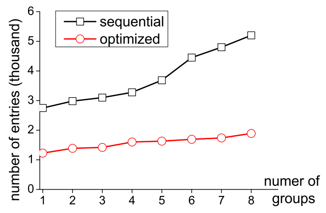
Number of pattern items n. We vary the average number of pattern items n from 5 to 30. The time constraint and confidence constraint of each query is randomly chosen from $[2n, 5n]$ and $[0.6, 0.8]$, respectively.

As shown by the results, the improvement achieved by our optimization method in comparing to the sequential approach is greater with a larger number of pattern items. This is because with long pattern structure, the order of searching for each pattern items becomes more important. In this set of experiments, the optimized method outperforms the sequential evaluation on all three real datasets. The test results of both our methods on the Dodfers dataset is given in Figure 8. Other results are omitted due the space limit.

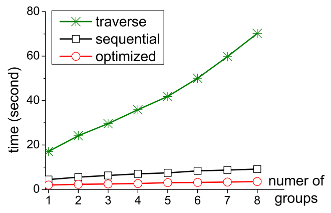
Time constraint L. We vary the time constraint from n to $5n$, where n is the number of pattern items of the particular query structure.

Shown by the test results, we find out that the performance of both query processing method decreases as L increases. This is because given a pattern structure, the loner of the time constraint, the more matches will be returned. Nevertheless, on all three real datasets, the optimized method's performance decreases much slower than the sequential one. The results of the Dodfer dataset is given in Figure 9 and the results of other two datasets are omitted again due the space limit.

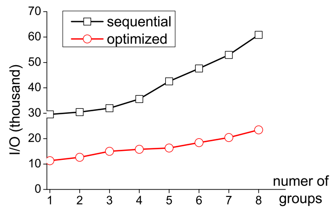
Confidence constraint. We vary the confidence constraint from 0.4 to 0.8 to examine its impact on the query performance. With larger confidence constraint, we expect more entries will be eliminated earlier during the query processing. This is validated by our experimental results. With both our query processing methods, the number of entries, query processing time and number of I/O de-



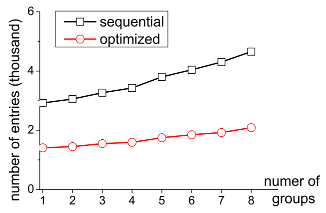
(a) Number of entries, Dodfers



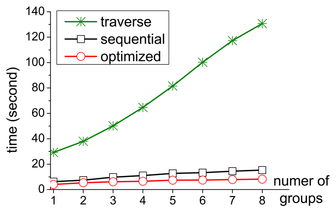
(b) Time, Dodfers



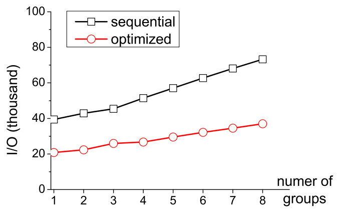
(c) I/O, Dodfers



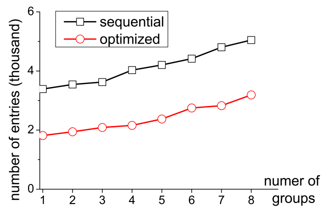
(d) Number of entries, ICU



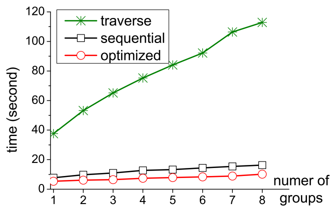
(e) Time, ICU



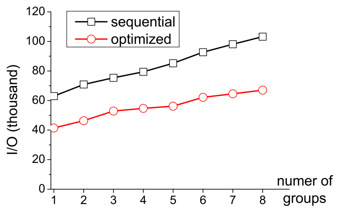
(f) I/O, ICU



(g) Number of entries, LPA

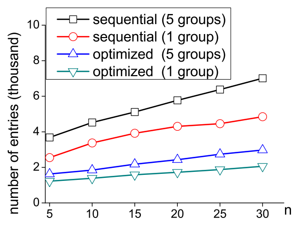


(h) Time, LPA

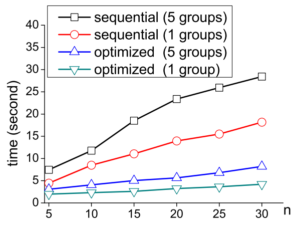


(i) I/O, LPA

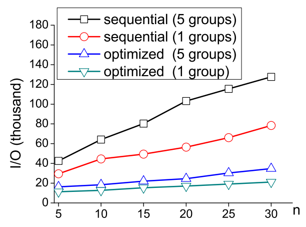
Figure 7: Effect of the number of dependency groups.



(a) Number of entries, Dodfers



(b) Time, Dodfers



(c) I/O, Dodfers

Figure 8: Effect of the number of pattern items.

crease decrease when the confidence constraint increases. The optimized method still outperforms its counterparts in all aspects on all datasets. Once again, we present the results on the Dodfers dataset in Figure 8 and omit the others as the trends shown on them are the same.

7. CONCLUSION AND FUTURE WORK

In this paper, we study the sequence pattern matching over archived event data with temporal uncertainties. To the best of our knowledge, this is the first work to address this problem. To enhance query efficiency, we propose an index structure to organize and index the archived event data, with which all possible orderings of events can be efficiently deduced. Based on such the index, we develop algorithms to efficiently extract patterns from the event data. Finally, we conduct an extensive experimental study on both real and synthetic datasets to prove the efficiency of our proposal. Experimental results show that our algorithms, both the index construction algorithm and query processing algorithms, are efficient and scale well with respect to query response time and I/O.

8. REFERENCES

[1] Dodfers dataset. `http://archive.ics.uci.edu/ml/datasets/Dodgers+Loop+Sensor`.

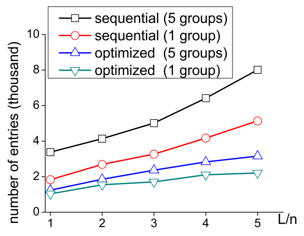
[2] Icu dataset. `http://archive.ics.uci.edu/ml/datasets/ICU`.

[3] Personal activity dataset. `http://archive.ics.uci.edu/ml/datasets/Localization+Data+for+Person+Activity`.

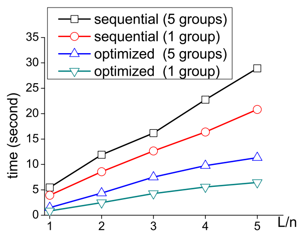
[4] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.

[5] M. Akdere, U. Cetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. In *PVLDB*, 2008.

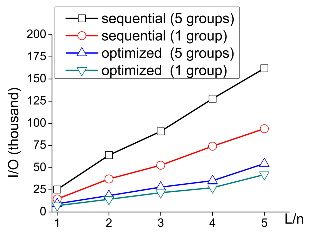
[6] J. Aßfalg, H.-P. Kriegel, P. Kröger, and M. Renz. Probabilistic similarity search for uncertain time series. In *SSDBM*, 2009.



(a) Number of entries, Dodfers

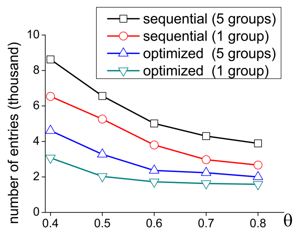


(b) Time, Dodfers

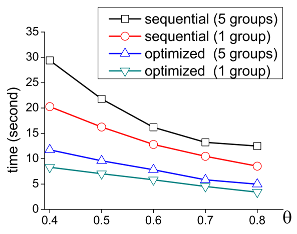


(c) I/O, Dodfers

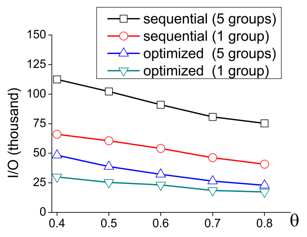
Figure 9: Effect of the time constraint



(a) Number of entries, Dodfers



(b) Time, Dodfers



(c) I/O, Dodfers

Figure 10: Effect of the confidence constraint

[7] Z. Cao, C. Suttony, Y. Diao, and P. Shenoy. Distributed inference and query processing for rfid tracking and monitoring. In *PVLDB*, 2011.

[8] K. M. Chandy, B. E. Aydemir, E. M. Karpilovsky, and D. M. Zimmerman. Event webs for crisis management. In *IASTED*, 2003.

[9] T. chung Fua, F. lai Chung, R. Luk, and C. man Ng. Stock time series pattern matching: Template-based vs. rule-based approaches. In *EAAI*, 2006.

[10] A. Dekhtyar, R. Ross, and V. S. Subrahmanian. Probabilistic temporal databases, i: algebra. In *TODS*, 2011.

[11] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *CIDR*, 2007.

[12] C. E. Dyreson and R. T. Snodgrass. Supporting valid-time indeterminacy. In *TODS*, 1998.

[13] X. Ge and P. Smyth. Deformable markov model templates for time series pattern matching. In *KDD*, 2000.

[14] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immelmann. On supporting kleene closure over event streams. In *ICDE*, 2008.

[15] L. Harada and Y. Hotta. Order checking in a cpoe using event analyzer. In *CIKM*, 2005.

[16] D. Jobst and G. Preissler. Mapping clouds of soa- and business-related events for an enterprise cockpit in a java-based environment. In *PPPJ*, 2006.

[17] E. Keogh. A fast and robust method for pattern matching in time series databases. In *TAI*, 1997.

[18] E. Keogh and P. Smyth. A probabilistic approach to fast pattern matching in time series databases. In *AAAI*, 1998.

[19] C. R. J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *SIGMOD*, 2008.

[20] S. Li, Y. Lin, S. H. Son, J. A. Stankovic, and Y. Wei. Event detection services using data service middleware in distributed sensor networks. In *IPSN*, 2003.

[21] X. Lian, L. C. 0002, and J. X. Yu. Pattern matching over cloaked time series. In *ICDE*, 2008.

[22] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. Claypool. Sequence pattern query processing over out-of-order event streams. In *ICDE*, 2009.

[23] S. Rizvi, S. R. Jeffery, S. Krishnamurthy, M. J. Franklin, and et al. Events on the edge. In *SIGMOD*, 2005.

[24] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, 2004.

[25] G. Trajcevski, A. Choudhary, O. Wolfson, L. Ye, and G. Li. Uncertain range queries for necklaces. In *MDM*, 2010.

[26] T. T. L. Tran, C. Sutton, R. Cocci, Y. Nie, Y. Diao, and P. J. Shenoy. Probabilistic inference over rfid streams in mobile environments. In *ICDE*, pages 1096–1107, 2009.

[27] F. Wang and P. Liu. Temporal management of rfid data. In *VLDB*, 2005.

[28] W. White, M. Riedewald, J. Gehrke, and A. Demers. What is “next” in event processing? In *PODS*, 2007.

[29] E. Wu, Y. Diao, and S. Riving. High-performance complex event processing over streams. In *SIGMOD*, 2006.

[30] M.-Y. Yeh, K.-L. Wu, P. S. Yu, and M.-S. Chen. Proud: a probabilistic approach to processing similarity queries over uncertain data streams. In *EDBT*, 2009.

[31] H. Zhang, Y. Diao, and N. Immerman. Recognizing patterns in streams with imprecise timestamps. In *VLDB*, 2010.

[32] K. Zheng, G. Trajcevski, X. Zhou, and P. Scheuermann. Probabilistic range queries for uncertain trajectories on road networks. In *EDBT*, 2011.