Knowledge Discovery from Series of Interval Events *

Roy Villafane (villafan@cs.ucf.edu), Kien A. Hua (kienhua@cs.ucf.edu) and Duc Tran (dtran@cs.ucf.edu)
University of Central Florida

Basab Maulik (bmaulik@us.oracle.com)
Oracle Corporation

Abstract. Knowledge discovery from data sets can be extensively automated by using data mining software tools. Techniques for mining series of interval events, however, have not been considered. Such time series are common in many applications. In this paper, we propose mining techniques to discover temporal containment relationships in such series. Specifically, an item A is said to *contain* an item B if an event of type B occurs during the time span of an event of type A, and this is a frequent relationship in the data set. Mining such relationships provides insight about temporal relationships among various items. We implement the technique and analyze trace data collected from a real database application. Experimental results indicate that the proposed mining technique can discover interesting results. We also introduce a quantization technique as a preprocessing step to generalize the method to all time series.

Keywords: data mining, knowledge discovery, time series, event sequence, temporal

1. Introduction

Several data mining techniques have been developed for conventional time series; (Agrawal et al., 1993), (Shatkay, Zdonik, 1996), (Agrawal et al., 1995), (Rafiei, Mendelzon, 1997), (Yazdani, Ozsoyoglu, 1996). In general, a time series is a collection of values for a given set of parameters ordered by time. Existing mining techniques treat these values as discrete events. That is, events are considered to happen instantaneously at one point in time, e.g., the speed is 15 miles/hour at time t. In this paper, we consider an event as being "active" for a period of time. For many applications, events are better treated as intervals rather than time points (Bohlen et al., 1998). As an example, let us consider a database application, in which a data item is locked and then unlocked sometime later. Instead of treating the lock and unlock operations as two discrete events, it can be advantageous to interpret them together as a single interval event that better captures the significance of placing, holding and releasing the lock. When there are

* This research is partially funded by grants from Oracle Corporation and the State of Florida.

several such events, a series of interval events is formed. Note that the name *series of interval events* does not imply that the interval events happen uniquely one after another, since there might be some overlap over their occurrences. An example is given in Figure 1; interval event B begins and ends during the time that interval event A occurs. Furthermore, interval event E happens during the time that interval event B happens (is active). The relationship is described as A *contains* B and B *contains* E. Formally, let $BeginTime(X)$ and $EndTime(X)$ denote the start time and end time of an event X, respectively. Event X is said to contain event Y if $BeginTime(X) < BeginTime(Y)$ and $EndTime(X) > EndTime(Y)$. We note that the containment relationship is transitive. Thus, A also contains E in this example (but this and several edges are not shown to avoid clutter).

Data mining can be performed on a series of interval events by gathering information about how frequently such containments happen. Associated with each containment relationship is a count of its instances in the series. For example, the relationship A *contains* B is observed 2 times in Figure 1. Given a threshold, a mining algorithm will search for all containments, including the transitive ones, with a count that meets or exceeds that threshold. These mined containments can shed light on the behavior of the entity represented by the series of interval events. The proposed technique can have many applications. We will discuss some in section 2. (Villafane et al., 1999) is an initial work on the techniques in this paper.

The problems of data mining association rules, sequential patterns and time series have received much attention lately as Data Warehousing and OLAP (On-line Analytical Processing) techniques mature. Data mining techniques facilitate a more automated search of knowledge from large data stores which exist and are being built by many organizations. Association rule mining (Agrawal, Imielinski et al., 1993) is perhaps the most researched problem of the three. Extensions to the problem include the inclusion of the effect of time on association rules (Chakrabarti et al., 1998) (Ramaswamy et al., 1998) and the use of continuous numeric and categorical attributes (Rastogi, Shim, 1998). Mining sequential patterns is explored in (Agrawal, Srikant et al., 1995). Therein, a pattern is a sequence of events attributed to an entity, such as items purchased by a customer. Like association rule mining, (Agrawal, Srikant et al., 1995) reduces the search space by using knowledge from size k patterns when looking for size $k+1$ patterns. However, as will be explained later, this optimization cannot be used for mining series of interval events. In (Mannila et al., 1997), there is no subgrouping of items in a sequence; a sequence is simply a long list of events. To limit the size of mined events and the algorithm

runtime, a time window width is specified so that only events that occur within time w of each other are detected. Unlike (Agrawal, Srikant et al., 1995), the fact that sub-events of a given-event are frequent cannot be used for optimization purposes.

A related work was presented in (Das et al., 1998). Therein, a rule discovery technique for time series was introduced. This scheme finds rules relating patterns in a time series to other patterns in that same or another series. As an example, the algorithm can uncover a rule such as "a period of low telephone call activity is usually followed by a sharp rise in call volume." In general, the rule format is as follows:

> If A_1 and A_2 and ... and A_h occur within V units of time, then B occurs within time T.

This rule format is different from the containment relationship defined in this current paper. The mining strategies are also different. The technique in (Das et al., 1998) uses a sliding window to limit the comparisons to only the patterns within the window at any one time. This approach significantly reduces the complexity. However, choosing an appropriate size for the window can be a difficult task. As we will discuss later, our technique does not have this problem.

The remainder of this paper is organized as follows. Section 2 covers some applications where this technique is useful. Functions, measures and other items related to the mining process are discussed in sections 3, 4, and 5. Mining algorithms are treated in section 6. Experimental studies are covered in section 7. Finally, we provide our concluding remarks in section 8.

2. Applications

Several applications exist where mining containment relationships can provide insight about the operation of the system in question. A database log file can be used as input to the mining algorithm to discover what events happen within the duration of other events; resource, record, and other locking behavior can be mined from the log file. Some of this behavior is probably obvious since it can be deduced by looking at query and program source code. Other behavior may be unexpected and difficult to detect or find because it cannot be deduced easily, as is the case for large distributed and/or concurrent database systems.

Another application area is mining system performance data. For example, a file-open/file-close event can contain several operations performed during the time that the file is open. Some of these operations may affect the file, while other operations are not directly associated with the file but can be shown to occur only during those times which

the file is open. Other interesting facts relating performance of the CPU to disk performance, for example, can be studied. Although performance data is not usually in interval event format, it can be converted to that format by using quantization methods.

In the medical field, containment relationship data can be mined from medical records to study what symptoms surround the duration of a disease, what diseases surround the duration of other diseases, and what symptoms arise during the time of a disease. For example, one may find that during a FLU infection, a certain strain of bacteria is found on the patient, and that this relationship arises often. Another discovery might be that during the presence of those bacteria, the patient's fever briefly surpasses 107 degrees Fahrenheit.

Factory behavior can also be mined by looking at sensor and similar data. The time during which a sensor is active (or above a certain threshold) can be considered an interval event. Any other sensors active during/within that time window are then considered to have a containment relationship with the first sensor. For example, it is possible to detect that the time interval during which a pressure relief valve is activated always happens within the time interval in which a new part is being moved by a specific conveyor belt.

3. Interval Events, Interval Event Series and Containment Graphs

An interval event IE_k is formally defined as a contiguous period in time during which an entity is in some given state. This period is defined by the time interval $[BeginTime(IE_k), EndTime(IE_k)]$, where $BeginTime(IE_k) < EndTime(IE_k)$. An entity can bear the given state multiple times during its lifetime, and for different amounts of time, so there can be several instances of interval event IE_k $(IE_1, IE_2, IE_3, ...)$. IE is considered an interval event type. Given two interval events A_p and B_q, a *containment* relationship exists where A_p contains B_q if $BeginTime(A_p) < BeginTime(B_q)$ and $EndTime(A_p) > EndTime(B_q)$. A containment can be both instance-specific as was just demonstrated, or generic for an interval event type. We can then refer to all of the instances of A_p *contains* B_q for all appropriate p and q as A *contains* B (without the subscripts). Containments are not limited to two levels. Consequently, we generically define a containment as a tuple of the form $CC =< n_1, n_2, ..., n_j >$, where each $n(i)$ is an interval event type and every $n(i)$ contains every $n(i+1)$ for all $1 \leq i \leq j-1$. There can be several instances of containment CC, each composed of a different combination of interval events.

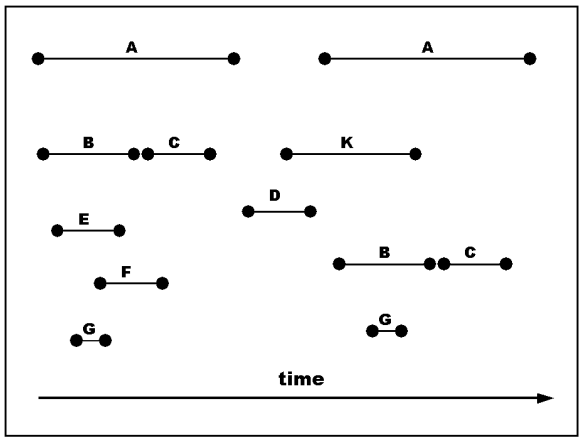

Figure 1. Interval Events

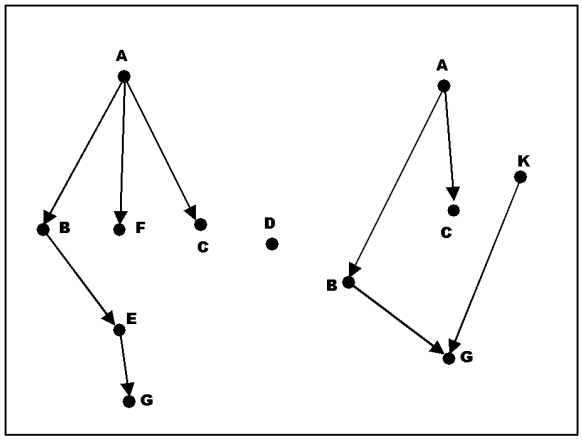

Figure 2. Containment Graph

A partial order can be imposed on the interval events to transform the series into a *containment graph*. Let this relation be called the containment relation. Applying this relation to the series in Figure 1 yields the graph in Figure 2. This graph represents the containment relationship between the events. A directed edge from event A to event B indicates that A *contains* B. We note that a longer series of interval events would normally consist of several directed graphs as illustrated in Figure 2. Furthermore, events can repeat in a series. For instance, events of type A occur twice in Figure 2. Each event is a unique instance, but the nodes are labeled according to the type of event. Given two event types S and D, all edges $S- > D$ in the containment graph represent instances of the same containment relationship.

Algorithm 1.

Input: List of event endpoints, sorted by time stamp, of the form
$< time\_stamp, event\_id, end\_point = beginorend, event\_type >$
Output: Containment graph $G = (V, E)$ of interval event nodes
$< event\_id, event\_type, begin\_time\_stamp, end\_time\_stamp >$
Variables: List open_events
Algorithm:

```
e = get_event()
while e <> NULL_EVENT
if e.end_point = begin
   /* there is a new event to be considered */
   open_events.add(e)
else if e.end_point == end
   /* this is the endpoint of event e.event_id */
   open_events.remove(e.event_id);
   for each event oe in open_events
      if oe.time_stamp < e.time_stamp
         /* create an edge if current start time is later */
         add graph edge E(oe,e) and appropriate nodes
      endif
   endfor
   for each directed edge ee_edge in E(e, *)
      ee_node = node pointed to by edge ee_edge
      if node ee_node.end_time_stamp >= e.time_stamp
         remove graph edge ee_edge
      endif
   endfor
endif
endwhile
```

The containment graphs shown are not fully connected for simplicity of illustration. However the algorithms and measures described in this paper use a transitively closed containment graph. This graph embodies all possible combinations of containment instances for all interval events. Using figure 2 as an example, the transitively closed containment graph would also include edges $< A, E >$, $< A, G >$ and

$< B, G >$ for the left subgraph and edge $< A, G >$ for the rightmost subgraph. A straightforward algorithm converts an interval event series into this kind of graph. It takes as input a list of tuples corresponding to interval event endpoints, sorted by time stamp, of the form

$< time\_stamp, interval\_event\_id,$
$end\_point\ in\ \{begin, end\}, interval\_event\_type >$

where each interval event has two such tuples: one for the beginning time and one for the ending time. By providing the input in this format, the entire graph can be loaded and built with one pass through the input data. Searching the graph for the location to insert new containments as they are added becomes unnecessary. Furthermore, it is not necessary to keep the entire graph on-line, so larger problem sizes can be solved (as will be described in the algorithms section).

The output is a directed containment graph $G = (V, E)$, where each node in V corresponds to an individual interval event and has a property tuple with attributes

$< interval\_event\_id, interval\_event\_type,$
$begin\_time\_stamp, end\_time\_stamp >$

Each directed edge in E from node V_i to V_k exists because interval event V_i contains interval event V_k. The constructed graph is transitively closed. Algorithm 1 outlines this process.

4. Quantization

It might be desirable to apply interval event mining to a dataset that is not in interval event form. Much conventional time series data is not fit for the data mining method presented in this paper. There is a large or potentially infinite number of different values that a parameter can assume, irrespective of whether the parameter is numerically discrete (integers) or continuous (real numbers). In such cases, there might be few or no repetition of containments, rendering the mining algorithm useless. Consequently, by setting thresholds and/or discretizing, quantitative performance data can be classified into "bins", and these bins can be considered intervals event types (that is, an interval event occurs during the contiguous time that the given parameter's value is within the specified bin's value range).

Suppose we have a day's worth of log data for CPU, disk and network interface usage. By carefully selecting predicates, such as

$C1 : 0 \leq CPU.busy < 30\%$
$C2 : 30\% \leq CPU.busy < 70\%$
$C3 : CPU.busy \geq 70\%$

$D1 : disk.busy < 40\%$
$D2 : disk.busy \geq 40\%$
$N1 : network.busy < 75\%$
$N2 : network.busy \geq 75\%$

parameter values in a conventional time series can be transformed into these discrete bin values according to which predicate is satisfied by a measurement point. Whenever two or more of these predicates occur contiguously, the time during which this happens can be interpreted as an interval event of type X, where X is in $\{C1, C2, C3, D1, D2, N1, N2\}$. Using these "bin-events", containments such as "when network usage is at or above 55%, disk usage is at or above 40%, and when such disk usage is observed, CPU usage during that time dips below 30%" can be discovered.

Quantization can be done in several ways, and many methods have been researched in various areas both within and outside of computer science. Important considerations include determining how many discrete values the data should be pigeonholed into (number of bins), the number of observations that should fall into each discrete value (number of instances of each bin value), and the range of continuous values that each discrete value should represent. To achieve some kind of grouping, clustering methods can be used along a parameter's range of observations, thereby coalescing similar values. The output of the regression tree methods in (Morimoto et al., 1997) can be used to partition continuous values into meaningful subgroups. This, of course, assumes that such groups exist in the data. The numeric ranges chosen for attributes in output from using (Rastogi, Shim, 1998) can also be utilized for segmentation. In the absence of such patterns, another method is to statistically separate the continuous data by using standard deviation and average metrics. This is the approach used in this paper for transforming the computer system performance data used in our experiments. Another method is to select equally sized ranges, without guaranteeing that each range will have an equal/significant number of observations. In contrast, the observations could be sorted by a parameter's values and then divided up into bins of equal size, without regard to the significance of the numeric attribute or the ranges formed thereof. The choice of which quantization method to use is heavily dependent on the domain that the data is coming from.

5. Counting Predicates and Support Measures

In the field of data mining, a key concept is that of constraint measures that the user specifies, which any piece of knowledge extracted must

satisfy. Support and confidence are among the most common. When mining series of interval events, several constraint measures or functions can be used for selecting useful knowledge. Each of these measures is a *counting predicate*. The usefulness and interestingness of the mined containments depend on which counting predicates are chosen. For traditional association rule mining, the two primary counting predicates are support and confidence. A factor driving the selection is the domain of data being mined, and consequently the form that the interval event data takes.

5.1. Counting Predicates

A *counting predicate* is a member function defined for a containment graph, which takes as a parameter a containment of the form $CC =< n_1, n_2, ..., n_j >$ (as previously defined). For each instance of containment CC in the graph, there exists a directed edge labeled $< n(i), n(i+1) >$ in the containment graph for each interval event instance of type $n(i)$ where $1 \leq i \leq j-1$. Applying a counting predicate to the set of all instances of a containment CC yields a value according to the properties of this set. Figure 3 and table 1 provide an example for containment $< A, B, X, Y, Z >$. Assume we are given a containment graph CG, a containment CC and the set/subgraph of all instances of containment CC in CG as $InstCC$. The notation $|CC|$ represents the size (number of interval event types) of the containment. Then counting predicates measure the following properties over $InstCC$:

- **Containment Frequency**: number of instances of CC in CG (same as number of instances of CC in $InstCC$)
- **Node Frequency**: number of unique nodes (interval events) in $InstCC$
- **Edge Frequency**: number of unique edges (containment relationships having 2 interval events) between nodes in $InstCC$
- **Edge/Node Coefficient**: given enr = number of unique edges in $InstCC$ divided by number of unique nodes in InstCC, and $enr\_min = (|CC| - 1)/|CC|$, the edge/node coefficient $enc = (enr - enr\_min)/(1 - enr\_min)$, where $enc >= 0$
- **Temporal Length Sum**: the sum of $EndTime(nt) - BeginTime(nt)$ for all unique interval events nt in $InstCC$, divided by $|CC|$

- **Maximum/Minimum Fan-in/Fan-out**: for each node nt in $InstCC$, the maximum/minimum number of incoming/outgoing edges

The simplest counting predicates involve measures of graph characteristics in the set of instances $InstCG$: *node frequency* and *edge frequency*. *Temporal length sum* is a measure accounting for the amounts of time that the containment relationship was observed. A relationship observed 1000 times over a few seconds might not be interesting, whereby another one observed 50 times spanning several hours would be; the opposite could also be true depending on the observer. Counting predicates *edge/node coefficient* and the four variations of *maximum/minimum fan-in/fan-out* (max/in, max/out, min/in, min/out) expose information about the structure of the mined containments. A *maximum fan-out* of 2, for example, limits the mined facts to cases whereby in a containment $< A, B >$, an instance of interval event A can have a containment relationship with at most 2 instances of interval event B. This serves in cases where it is desirable to avoid finding containments where a single, long interval event (such as 'computer system is powered on') contains numerous instances of short interval events (such as 'disable interrupts').

The former is not an exhaustive list of counting predicates. Additional predicates might be specified inspired by data mining techniques or graph theory. Furthermore, the definitions of the counting predicates defined in this paper could be altered slightly to alter the form of the mined containments. A variant of *Temporal Length Sum* could be defined as simply the sum of the lengths of all interval events, for example. Incidentally, counting predicates need not be applied in isolation for the mining operation. Multiple counting predicates can be combined to form a boolean expression called a *counting predicate function*, which each mined containment must satisfy. Allowing this freedom broadens the applications of the mining method because the user has greater control over what constitutes useful mined knowledge.

5.2. Multipath Counting Predicates

Because the containment graph is a lattice, an internal node can have several parent nodes. This property translates into entire subpaths that can be shared by several nodes. So when counting the frequency of a path, should nodes be allowed to appear in more than one path? For example, in the containment graph in Figure 3, how often does containment $< A, B, X, Y, Z >$ occur? If nodes can appear on more than one path, then the counting predicate becomes *multipath containment frequency* and the value for containment $< A, B, X, Y, Z >$ is 2. If the

nodes on a path are prohibited from appearing on more than one path, then the counting predicate is simply *containment frequency* and the result is 1. Examples of these counting predicates follow. The definitions for the *multipath counting predicates* are:

- **Multipath Containment Frequency**: number of distinct paths which are instances of CC in CG (same as number of paths which are instances of CC in $InstCC$)

- **Multipath Node Frequency**: number of nodes in all distinct instances of CC in CG, which is the product of $|CC|$ and the *multipath containment frequency* of CC

- **Multipath Edge Frequency**: number of edges in all distinct instances of CC in CG, which is the product of $(|CC|-1)$ and the *multipath containment frequency* of CC

Note that multipath equivalents of *edge/node coefficient* and *maximum/minimum fan-in/fan-out* are not necessary. The relaxation of the uniqueness of nodes and edges results in $InstCC$ consisting of a set of disconnected paths, each path being an instance of containment CC. Hence, the multipath edge/node coefficient is always 0. Similarly, the *multipath fan-in* of interval events contained by other interval events is always 1, and the *multipath fan-out* of interval events containing other interval events is also 1. *Multipath temporal length sum* is not considered since counting a time period multiple times can result in a sum which exceeds the time period of the entire dataset.

The relationships between containment frequency, edge frequency, node frequency, and the multipath variations of these counting predicates will vary according to the shape of the containment graph, which in turn is determined by how interval events contain each other. Table 1 shows the counting predicates and their values for containment $< A, B, X, Y, Z >$ from figure 3. The interval event lengths are $A = 10, B = 7, X = 6, Y = 2, Z = 1.5$, and $|CC| = 5$.

When is a multipath counting predicate favored over its non-shared counterpart? A non-shared counting predicate typically indicates what percentage of the containment graph supports a given containment. It does not readily differentiate where there is overlap among instances of a given containment and where there is not. For example, in Figure 4, the containment frequency for $< B, A, F >$ is 2 because there are at most 2 unique occurrences of this containment given the restrictions of that counting predicate. In contrast, the multipath containment frequency is 24 ($3x4x2$). Likewise, the node frequency is 9, and in contrast the multipath node frequency is 72 ($3x24$). In certain problem

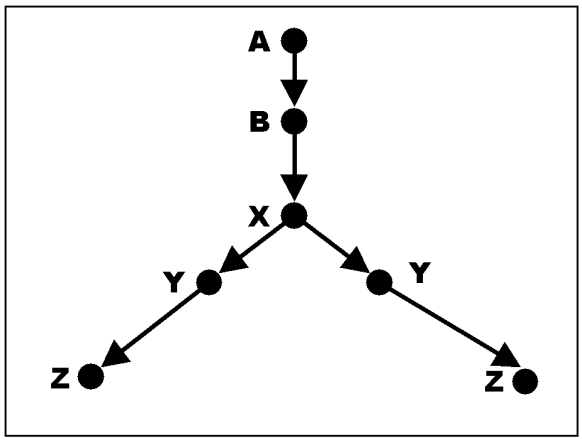

Figure 3. Shared subcontainments

Table I. Counting predicates for $< A, B, X, Y, Z >$

Counting Predicate	**Value**
containment frequency	1
multipath containment frequency	2
edge frequency	6
multipath edge frequency	8
node frequency	7
multipath node frequency	10
edge/node coefficient	0.2857
temporal length sum	6
multipath temporal length sum	10.6

domains, the fact that there is overlap between several instances of the same containment is useful information. Suppose that interval event type B is a disk failure, interval event type A is a network adapter failure, and interval event type F is a network failure. The fact that these events happen at approximately the same time, thus causing the amount of overlap seen in the example, has a different meaning than if these containments happened at different times. The events probably occur together because of a malicious program virus attack that is set off at a specific time of day, for example.

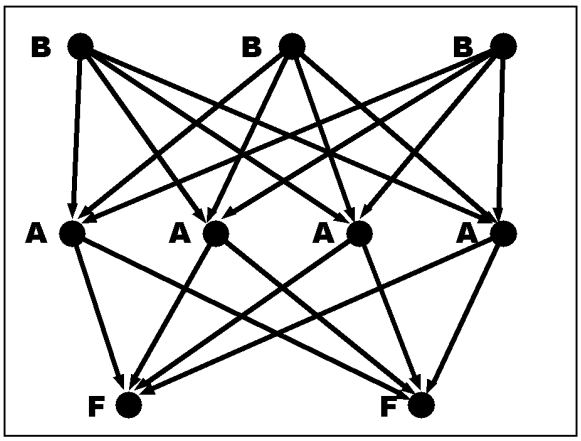

Figure 4. Multiple shared subcontainments

5.3. Containment Support and Confidence

In association rule mining and several other data mining methods, *support* and *confidence* are measures which specify some statistical strength or significance of a mined fact. Thresholds are specified a priori so mining algorithms only return facts which are frequently represented in a dataset. In consistency with other data mining methods, we also define support and confidence measures for mining series of interval events. These are based upon the counting predicates formerly defined.

The support of a mined containment can be loosely defined as the percentage of the counting graph which contributes to the value of a given counting predicate. Another definition is the value of a counting predicate for a containment divided by the value of the counting predicate for the entire containment graph, if the containment instances exclusively comprised the graph. The semantics of support vary among the counting predicates over which it is defined. For a given containment graph CG and containment CC, the support measures are as follows:

- **Containment Support**: *containment frequency* divided by the maximum containment frequency possible for that containment
- **Edge Support**: *edge frequency* divided by the number of edges in CG
- **Node Support**: *node frequency* divided by the number of nodes in CG
- **Temporal Length Sum Support**: *temporal length sum* divided by the time span $TimeCG$ which CG represents, where $TimeCG =$

$max\{EndTime(A_k)\} - min\{BeginTime(B_m)\}$ for all interval events A_k and B_m in CG

Node support and *edge support* each identify the percentage of the containment graph which contributes to the existence of the mined containment. To attain the optimal containment support of 1.0, the nodes in the containment graph would need to be restructured such that each and every node was part of an instance of containment CC. Obviously, many nodes will likely correspond to interval events whose type is not in CC, so this might often be impossible. Restructuring the graph also means that the input data would require alteration, another impossibility. In the absence of such manipulations, the containment support is the number of instances of CC mined from the real containment graph divided by the number of such containments that would be mined from the hypothetically restructured graph. We do not define support measures for *edge/node coefficient* nor *maximum/minimum fan-in/fan-out* because these measures are not affected by subgraphs outside the one representing $InstCC$. Consequently, specifying a counting predicate threshold is similar to specifying a support measure.

If we define a *maximal traversal path* as a path whose first node does not have a parent node and whose last node does not have a child node, then multipath support measures also exist as follows:

- **Multipath Containment Support**: *multipath containment frequency* divided by the maximum multipath containment frequency possible for $|CC|$
- **Multipath Edge Support**: *multipath edge frequency* divided by the sum of the number of edges in all unique *maximal traversal paths* of CG
- **Multipath Node Support**: *multipath node frequency* divided by the sum of the number of nodes in all unique *maximal traversal paths* of CG

Closely related to support, confidence can be loosely defined as the percentage of the number of subcomponents of a mined fact in the dataset that contribute to the existence of that fact. As before, given CG, CC and $InstCC$, confidence measures are as follows:

- **Node Local Confidence**: number of nodes in $InstCC$ divided by the number of nodes nt (interval events) in CG where the type of any such nt is any of the types found in CC
- **Edge Local Confidence**: number of edges in $InstCC$ divided by the number of edges ecg (2-way containment relationships) in CG

where ecg is an instance of containment $< SourceNodeType(ecg)$, $DestinationNodeType(ecg) >$ and this is a containment of the form $< n(i), n(i+1) >$ from CC

And as in support, corresponding *multipath confidence measures* are as follows:

- **Multipath Node Local Confidence**: product of $|CC|$ and *multipath containment frequency* of CC divided by the number of nodes nt (interval events) in all unique *maximal traversal paths* of CG, where the type of any such nt is any of the types found in CC

- **Multipath Edge Local Confidence**: product of $|CC|$ and *multipath containment frequency* divided by the number of edges ecg (2-way containment relationships) in all unique *maximal traversal paths* of CG, where ecg is an instance of containment $< SourceNodeType(ecg)$, $DestinationNodeType(ecg) >$ and this is a containment of the form $< n(i), n(i+1) >$ from CC

Note that *edge local confidence* is only interesting for containments of size 3 (having 3 interval events) or greater; for any containments of size 2 in CG, the *edge local confidence* will always be 1.0. By carefully selecting support and confidence thresholds, a user can mine for containments that happen frequently ("large support"), containments where the relationships between interval event types are limited to each other ("large edge local confidence"), both of the former and a large number of combinations. These combinations are formed by building a counting predicate function such as $node\_local\_confidence \geq 0.7$ *and* $node\_support \geq 0.22$ *and* $edge/node\_ratio \leq 0.4$. As a final note, computing the value of some counting predicates and support/confidence measures might not be trivial, such as containment frequency; the best algorithm to compute this might be of NP time complexity. Computing some multipath counting predicates might also at first seem to have NP time complexity with respect to the containment graph depth. However, using dynamic programming methods for the graph traversal helps reduce/eliminates this penalty.

6. Mining algorithms

There are several ways to mine the data to find the frequent containments. The naive approach is to traverse the lattice on a depth-first basis and at each point of the traversal enumerate and count all paths.

Another way is to search for containments incrementally by path size, which is the approach used in this paper. A path is described by the ordered sequence of node types in the path. Because there is a one-to-many mapping from node types to node instances, a path can exist multiple times in the entire containment graph. It can be traversed using lattice traversal algorithms, or it can be stored in relational database tables and mined using SQL statements.

6.1. Naive Algorithm for Mining Containments

Perform a depth-first traversal of the lattice whereby all the possible paths throught the lattice are explored. At each node visit of the traversal, there exists a traversal path TP by which this node was reached. This corresponds to the recursive calls that a program is following. Path TP is $< tp_1, tp_2, tp_3, ..., tp_n >$, where tp_1 is the topmost node in the path and tp_n is the current node (which can be an internal or leaf node) being visited by the traversal algorithm. By definition, tp_1 has no parent and hence, there is no other interval event which contains the one represented by tp_1.

Each subpath (containment) of TP can be further broken down into a set $\{< tp_{(n-1)}, tp_n >, < tp_{(n-2)}, tp_{(n-1)}, tp_n >, ..., < tp_1, tp_2, ..., tp_{(n-1)}, tp_n >\}$. Consider each element of this set, labelled by TPS (the containment of which it is an instance). Note that we will not miss any containment instances. It is not necessary to "skip" graph nodes (for example, $< tp_1, tp_3, tp_8 >$) when considering TPS elements because the containment graph is transitively closed. Consequently, an enumeration of all possible paths is equivalent to an enumeration of all possible containment instances. Define a *path dictionary* having entries of the form $< TPS, cp_1, cp_2, ... >$, where each cp_i corresponds to the value of a counting predicate for containment TPS. As the containment graph is traversed, the path dictionary is updated, adding or modifying information for each containment instance encountered.

When the entire lattice has been traversed, the containments in the path dictionary that satisfy the counting predicate(s) (such as $containment\ frequency \geq minimum\ mining\ containment\ frequency$) are presented to the user. This exhaustive counting method will enumerate *all* possible containments. Herein lies the disadvantage: the number of mined containments will typically be a very small subset of all the possible containments. This algorithm might not have a chance to run to completion because of the large amount of storage required to store all the paths. We discuss this algorithm because it helps to illustrate the mining technique.

6.2. Growing Snake Traversal

Algorithm 2.

```
Input: Containment graph CG, containment predicate function CPF
Output: Set FINAL_CONT of mined containments
Variables: containment_bucket array CA[]
   (each element containing CASIZE containments)
containment_bucket FINAL_CONT
Algorithm: int k = 0
for containment size CS = 2 to CG.max_containment_size
   for each containment CCL in CG of size CS
      put CCL in current bucket CA[k]
      if CA[k] is full
         sort CA[k]
         allocate a new bucket CA[k+1]
         k=k+1
      endif
   endfor
   Merge all CCLs in all CA buckets into the FINAL_CONT
   bucket, inserting only those that meet the specified
   counting predicate function, using a k-way merge to
   merge the buckets (alternatively use 2-way merges).
   Delete all containments in CA.
endfor
```

Unlike several other data mining methods (such as association rule mining), when finding frequent containments it is not always possible to prune the search space by using mining results of previous iterations. Use the *containment frequency* counting predicate as an example. A corresponding statement, if it held, for a containment $CSUPER$ with subcontainment $CSUB$ would be $containment\_frequency(CSUB) \geq containment\_frequency(CSUPER)$. Unfortunately, this property can not be consistently exploited by mining in stages for incrementally larger containments, because several of these larger containments can potentially share a smaller containment. Sharing leads to violation of this property. Containment $< A, B, X, Y, Z >$ shown in Figure 3 illustrates this: the containment frequency for $< A, B, X >$ is 1, but

the containment frequency for $< A, B, X, Y, Z >$ is 2, a higher value. Results are similar for a few counting predicates.

To reduce the amount of storage required for intermediate results, the *Growing Snake Traversal*, as the name implies, starts by mining all size 2 containments. A traversal is done over the transitively closed containment graph, as in the naive algorithm, except that only paths of the form $< tp_{(n-1)}, tp_n >$ are enumerated. When all such containments have been found, only those that satisfy the selected counting predicate function are retained. Next, containments of size 3 having form $< tp_{(n-2)}, tp_{(n-1)}, tp_n >$ are enumerated and the same counting predicate function is applied to select useful containments. This is repeated until the maximum containment size or maximum user specified containment size is reached. Algorithm 2 contains the details.

For each containment size CS, the step of containment enumeration is followed by a merge-count because the enumeration has to happen in stages in order to effectively use the limited amount of available RAM (Random Access Memory). For example, given about 7 hours worth of interval data from discretized performance data from a system running an Oracle database application, the memory usage for the algorithm can at times exceed 300MB. The algorithm accesses large structures non-sequentially. With sufficient disk space to store it but not enough RAM for it all to be on-line at once, excessive swapping will be encountered, rendering the algorithm ineffective. A merge-count allows the use of very large datasets. The $CASIZE$ parameter is chosen such that the size of each $CA[k]$ is small enough to fit in physical RAM. Although it is not explicitly shown, our implementation of the algorithm ensures that a containment is not counted twice. This is done by pruning paths which exist entirely within subsections of the graph which have already been visited. Remaining duplicate edges and nodes that arise during the merge step (as a result of paths which are partially in a formerly explored region of the graph) are eliminated during the merge phase of the algorithm.

In our experiments, the entire containment graph was kept on-line. The graph does not need to be stored completely on-line, however. A modification to the algorithm permits mining datasets where the containment graph is larger than available RAM space by only keeping events in memory that are "active" during the current timestamp. There is no required traversal order required for enumerating containments, as long as all possible containment instances are explored, as shown in Algorithm 2. Hence, we can explore the graph in a temporal order which corresponds to the order in which interval event endpoints are read from the input. As we build up the containment graph, the mining algorithm is run. On any given step, we can dispose of sections

of the graph which have been visited and will no longer be necessary. These sections can be readily identified. Recall that during the graph traversal, there is a path TP by which the current node was reached. The containment graph nodes GN to be discarded are those where $BeginTime(GN)$ is less than the minimum $BeginTime(TP_i)$ for all TP_i in TP. Thus, the section of the containment graph being mined is built dynamically as access to it is required.

A combination of merge-count enumeration and partial containment graph yields an algorithm that is limited only by available secondary storage. The data access patterns for graph traversal and enumeration (if using multiple 2-way merges) are sequential. In extremely large problem cases, a group of devices that support sequential access, such as tape drives, could also be used by the algorithm.

7. Experimental Results

Experiments were run on a Dell PowerEdge 6300 server with 1GB RAM and dual 400Mhz Pentium processors for the synthetic data, and on a Dell Pentium Pro 200Mhz workstation with 64MB RAM. The first experiment consisted of mining containment relations from an artificially generated event list. A Zipf distribution was used in selecting the event types and a Poisson arrival rate was used for the inter-event times. This smaller list is beneficial in testing the correctness of the programmed algorithm because the output can be readily checked for correctness.

In the second experiment, disk performance data from an Oracle client/server database application was converted from quantitative measurements to interval events by quantizing the continuous values into discrete values. The disk performance data consists of various parameters for several disks, measured at 5-minute time intervals. Discrete values were chosen based on an assumed normal distribution for each parameter and using that parameter's statistical z-score. *Low*, *average* and *high* were assigned to a value by assigning a z-score range to each discrete value. Values used were *low*, corresponding to $z-score < -0.5$, *average* corresponding to a $z - score$ in $[-0.5, 0.5]$, and *high* corresponding to a $z - score > 0.5$. The resulting quantized versions of the parameters were close to uniformly distributed in terms of the number of occurrences of each range.

Some containment results gathered from looking at the output of the sar utility of the Sun machine the database was running on are shown in Table 2. Additionally, several containments were the average service time parameter of disk id's 40, 18, 20 and 25 were near their mean value, contained several other quantized values of parameters of other

disks, revealing interesting interactions among several disk performance metrics which were obtained by running the mining algorithm. Figure 5 shows the relationship between varying Zipf, Poisson arrival times and number of mined interval events for the synthetic data set, which consists of 500 events and 8 event types.

Table 3 shows the CPU run times for mining the Oracle dataset on the Pentium Pro workstation. Because the algorithm processes an acyclic directed graph using a depth-first traversal, the complexity depends greatly on the shape of the graph. Consequently, it is difficult to provide good space and time complexity measures. Some guidelines can be provided if we traverse the graph in some form of temporal order as discussed in the algorithms section. For several sequential pattern methods discussed in the introduction, the user must specify a window size within which relationships will be discovered. Otherwise, the problem becomes intractable because all event combinations must be considered. Due to the nature of containment relationships, the window over which an exponential number of combinations must be considered is determined by the temporal length of the interval events themselves. So with respect to the *breadth* of the containment graph, greater concentrations of interval events and interval events having longer temporal length might lead to subgraph sections with greater depth and greater possibilities of intractability. Data mining methods in general are at the mercy of combinatorial explosion in the worst cases, and containment relationship mining is not exempt from this. Fortunately, this can be addressed to some extent by limiting the explored combinations (a method that can also be used for other data mining techniques). With respect to the *breadth* of the containment graph, the mining algorithm is linear. Given a containment graph with somewhat 'homogeneous' subsections given a temporal order traversal, the time to process each of these subsections is similar from one to another.

Finally, we collected system performance data during a disk defragmentation operation on the PowerEdge server for over 120 parameters. The counting predicate specified was $node\_support \geq 0.7\%$ *and* $node\_local\_confidence \geq 30\%$ *and* $edge/node\_coefficient \leq 0.40$. The given *node support* was chosen due to the high number of parameters. The mined containments in Table 4 show an interaction between the interrupt processing time, both processors, and the DPC (deferred procedure call) rate. The containments show that during times of average load processing interrupts, CPU 1 is assigned a lower number of deferred procedure calls than CPU 0, so the deferred procedure calls are not equally distributed among both processors. The reason could either be that such balancing is not implemented / not possible, or that CPU

Table II. Some Oracle dataset results

Param 1	Param 2	Description
Page faults 'high'	namei 'high'	During the time that the number of page faults is above average, the number of namei function requests is also high. This is probably an indication that files are being opened and accessed, thus increasing the RAM file cache size and reducing the amount of RAM available to execute code
'average' CPU usage by system	vflt 'low'	During average usage of the CPU by the system code, the number of address translation page faults was below average. This might be an indication that much system code is non-pageable, so very little page faults are generated
'average' CPU usage by system	slock 'average'	During average usage of the CPU by the system, there is an average number of lock requests requiring physical I/O

Table III. CPU time for execution of mining algorithm vs. number of containments mined for Oracle data

# of events	cpu time (sec)
178	40
286	104
335	367
387	543

1 is assigned DPCs only after the DPC queue of CPU 0 reaches a specific load (which is not necessarily a negative situation).

8. Concluding Remarks

Various data mining techniques have been developed for conventional time series. In this paper, we investigated techniques for series of interval events. We consider an event to be "active" for a period of time,

Table IV. Windows NT defragmentation operation

Param 1	Param 2	Description
Proc0 Interrupt Time Avg	Proc0 DPC Rate Avg	When time spent by CPU 0 handling interrupts is average, number of DPCs for CPU 0 is average
Proc0 Interrupt Time Avg	System Total DPC Rate Avg	When time spent by CPU 0 handling interrupts is average, number of DPCs for entire system is average
Proc1 Interrupt Time Avg	Proc1 DPC Rate Low	When time spent by CPU 1 handling interrupts is average, number of DPCs for CPU 1 is low
Proc1 Interrupt Time Avg	System Total DPC Rate Low	When time spent by CPU 1 handling interrupts is average, number of DPCs for entire system is low

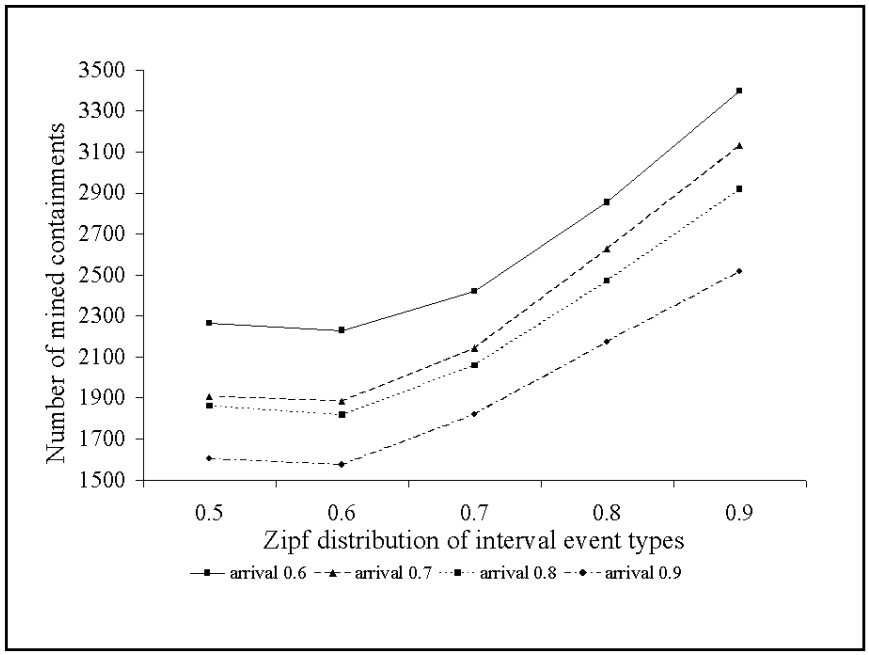

Figure 5. Synthetic data results

and a series of interval events as a collection of such interval events. We pointed out that existing techniques for conventional time series and sequential patterns cannot be used. Basically, series of interval events are mined differently than event series because an event has both a starting and ending point, and therefore the containment relationship has different semantics than simply happens-before or happens-after.

To address this difference, we proposed a new mining algorithm for series of interval events.

To assess the effectiveness of our technique, we ran the mining algorithm on system performance trace data acquired from an application running on an Oracle database. Traditionally, spreadsheet and OLAP (On-line analytical processing) tools have been used to visualize performance data. This approach requires the user to be an expert and have some knowledge of what to explore. Unsuspected interactions, behavior, and anomalies would run undetected. The data mining tools we implemented for this study address this problem. Our experimental study indicates that it can automatically uncover many interesting results.

To make the techniques more universal, we proposed a quantization technique which transforms conventional time series data into an interval event time series, which can then be mined using the proposed method. To illustrate this strategy, we discussed its use in a number of applications.

References

Agrawal, R., Faloutsos, C. and Swami, A. (1993). Efficiency Similarity Search in Sequence Databases. Proceedings of the Conference of Foundations of Data Organization, 22.

Agrawal, R., Imielinski, T. and Swami, A. (1993). Mining Association Rules Between Sets of Items in Large Databases. ACM SIGMOD.

Agrawal, R., Psaila, G., Wimmers, E.L. and Zait, M. (1995). Querying Shapes of Histories. Proceedings of VLDB.

Agrawal, R. and Srikant, R. (1995). Mining Sequential Patterns. IEEE Data Engineering.

Bohlen, M.H., Busatto, R. and Jensen, C.S. (1998). Point- Versus Interval-based Temporal Data Models. IEEE Data Engineering.

Chakrabarti, S., Sarawagi, S. and Dom, B. (1998). Mining Surprising Patterns Using Temporal Description Length. Proceedings of the 24th VLDB Conference.

Das, G., Lin, K., Mannila, H., Renganathan, G. and Smyth, P. (1998). Rule Discovery From Time Series. The Fourth International Conference on Knowledge Discovery & Data Mining.

Morimoto, Y., Ishii, H. and Morishita, S. (1997). Efficient Construction of Regression Trees with Range and Region Splitting. Proceedings of the 23rd VLDB Conference.

Mannila, H., Toivonen, H. and Verkamo, A.I. (1997). Discovery of Frequent Episodes in Event Sequences. Data Mining and Knowledge Discovery 1, 259-289.

Rafiei, D. and Mendelzon, A. (1997). Similarity-Based Queries for Time Series Data. SIGMOD Record.

Ramaswamy, S., Mahajan, S. and Silberschatz, A. (1998). On the Discovery of Interesting Patterns in Association Rules. Proceedings of the 24th VLDB Conference.

Rastogi, R. and Shim, K. (1998). Mining Optimized Association Rules with Categorical and Numeric Attributes. IEEE Data Engineering.

Shatkay, H. and Zdonik, S.B. (1996). Approximate Queries and Representations for Large Data Sequences. Proceedings of the 12th International Conference on Data Engineering.

Villafane, R., Hua, K.A., Tran, D. and Maulik, B. (1999). Mining Interval Time Series. Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery.

Yazdani, N. and Ozsoyoglu, Z.M. (1996). Sequence Matching of Images. Proceedings of the 8th International Conference on Scientific and Statistical Database Management.