Mining Interval Time Series

Roy Villafane[1], Kien A. Hua[1], Duc Tran[1], Basab Maulik[2]

[1]University of Central Florida, School of Computer Science
{villafan, kienhua, dtran}@cs.ucf.edu
[2]Oracle Corporation
bmaulik@us.oracle.com

Abstract. Data mining can be used to extensively automate the data analysis process. Techniques for mining interval time series, however, have not been considered. Such time series are common in many applications. In this paper, we investigate mining techniques for such time series. Specifically, we propose a technique to discover temporal containment relationships. 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 allows the user to gain insight on the 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.

1 Introduction

Numerous data mining techniques have been developed for conventional time series (e.g., [1], [13], [3], [10], [14].) In general, a time series is a sequence of values of a given variable 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 [5]. 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 nature of the lock. When there are several such events, an interval time series is formed. An example is given in Figure 1; interval event *B* begins and ends during the time that interval event *A* is occurring. 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).

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 [2] is perhaps the most researched problem of the three. Extensions to the problem include the inclusion of the effect of time on association rules [6][11] and the use of continuous numeric and categorical attributes [12]. Mining sequential patterns is explored in [4]. Therein, a pattern is a sequence of events attributed to an entity, such as items purchased by a customer. Like association rule mining, [4] 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 interval time series. In [9], 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 [4], the fact that sub-events of a given-event are frequent cannot be used for optimization purposes.

The name *interval event sequence* does not imply that the interval events happen sequentially, as we have seen that intervals may overlap. A partial order can be imposed on the events to transform the sequence into a graph. Let this relation be called the containment relation. Applying this relation to the above example yields the graph in Figure 2. This graph represents the containment relationship between the events. A directed edge from event *A* to event *B* denotes the fact that *A contains B*. We note that a longer event sequence would normally consist of several directed graphs as illustrated in Figure 2. Furthermore, events can repeat in a sequence. 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.

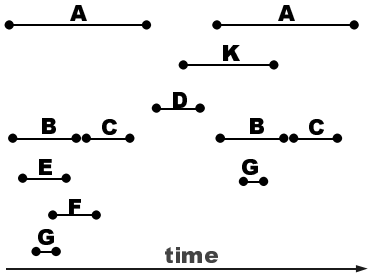

Fig. 1. Interval Events

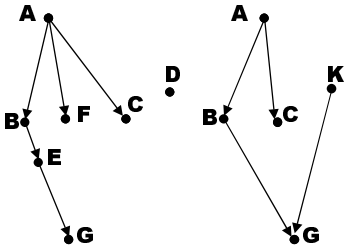

Fig. 2. Containment Graph

Data mining can be performed on the interval sequence by gathering information about how frequently such containments happen. Given two event types *S* and *D*, all edges *S->D* in the containment graph represent instances of the same containment relationship. Therefore, associated with each containment relationship is a count of its instances. For example, the count for *A contains B* is 2 in Figure 2. 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 interval time series. The proposed technique can have many applications. We will discuss some in section 2.

A related work was presented in [7]. 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 *A1* and *A2* and … and *Ah* occur within *V* units of time, then *B* occurs within time *T*.

This rule format is different from the containment relationship defined in the current paper. The mining strategies are also different. The technique in [7] 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. Algorithms, functions, measures and other items related to the mining process are discussed in section 3. Experimental studies are covered in section 4. Finally, we provide our concluding remarks in section 5.

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 Mining Interval Time Series

3.1 From Series of Interval Events to Containment Graph

Both of the containment graphs shown in the introduction are minimally connected for simplicity of illustration. However, the algorithms and measures described in this paper use a transitively closed version of a containment graph. A straightforward algorithm converts an interval event series into this kind of graph. It takes a list of event endpoints, sorted by time stamp, of the form

<time_stamp, event_id, end_point in {begin,end}, event_type}

where each interval event has two such tuples: one for the beginning time and one for the ending time. By having the input in this format, the entire graph can be loaded and build with one pass through the input data, and searching the graph for the location of a containment (as each new containment is added to it) becomes uncecessary. The output is a directed containment graph *G=(V,E)*, where each node in *V* corresponds to an individual interval event and is of the form

<event_id, event_type, begin_time_stamp, end_time_stamp>

and each directed edge in *E* from a node *Vi* to a node *Vk* exists because interval event *Vi* contains interval event *Vk*. The constructed graph is transitively closed in order to reduce the complexity of the mining algorithms.

3.2 Quantization

It might be desirable to apply interval event mining to a dataset that is not in interval event form. Continuously varying data is not fit for mining because of the potentially infinite number of different values that a parameter can assume. In such cases, there might not be any repetition of containments, rendering the mining algorithm useless. By setting thresholds and/or discretizing, quantitative performance data can be classified into bins, and these bins can be considered intervals (that is, an interval event occurs during the time that the given parameter is within the specified bin 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<=CPU.busy<30%*, *C2:30%<=CPU.busy<70%*, *C3:CPU.busy>=70%*, *D1:disk.busy<40%*, *D2:disk.busy>=40%*, *N1:network.busy<75%*, and *N2:network.busy>=75%*, continuously varying performance data can be transformed into these discrete bin values according to which predicate is satisfied by a measurement point. Furthermore, whenever two or more of these predicates occur contiguously, the time during which this happens can be interpreted as an interval 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%, and this behavior was observed with P% support" can be discovered.

Quantization can be done in several ways, and many methods have been researched in various areas both within and outside computer science. Some important considerations include determining how many discrete values the data should be pigeonholed into, the number of observations that should fall into each discrete 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. This, of course, assumes that such groups exist in the data. The output of the regression tree methods in [8] can be used to segment continuous values into meaningful subgroups. The numeric ranges chosen for attributes in output from using [12] 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 Oracle performance data. Another method is to select equally sized ranges, without guaranteeing that each range will have an equal number of or a significant number of observations. In contrast, the observations could be sorted and then divided up into bins of equal size, without regard to the significance of the numeric attribute. The choice of which quantization method to use is heavily dependent on the domain that the data is coming from.

3.3 Containment Frequency and Support

In the field of data mining, a recurrent theme is that of constraint measures that the user specifies, which any piece of knowledge extracted must satisfy. Support, confidence, and interestingness are some of the most common. In interval time series mining, several functions can be used for selecting useful knowledge. Each of these measures will be referred to as a counting predicate. The usefulness and interestingness of the mined containments depend on which counting predicate is chosen. A factor driving the selection is the domain of data being mined, and consequently the form that the interval event data takes.

Some of the most straightforward counting predicates involve measures of items in the containment graph. The most obvious counting predicate is the number of times that a given containment appears. For this measure, the containment frequency measures the number of times a containment appears in the graph, where each

containment counted does not share any nodes (interval events) with any other instance of that containment. Multipath containment frequency relaxes this requirement, and thus counts the number of times a containment exists, given all possible paths that exist in the graph. Node frequency is the number of distinct nodes which comprises the set of all combined nodes from all the paths for a given containment. Similarly, edge frequency is number of distinct edges (size-two containments) which comprises the set of all combined edges from all the paths for a given containment. Multipath node frequency and multipath edge frequency relax the distinctness requirement in a fashion similar to the difference between containment frequency and multipath containment frequency, so a node/edge can be counted multiple times. Examples of these counting predicates follow.

3.3.1 Counting Predicates and Containments Enumeration

Define a containment (or path) as a tuple of the form *CC = <n1, n2, …, nk>*, where each *n(i)* is an interval event, and is labeled by its interval event type ID. Each *n(i+1)* is contained by *n(i)* for all *1<=i<=k-1* and there exists a directed edge *E(n(i), n(i+1))* in the containment graph for all such *i*. As discussed, containment frequency can be measured in different ways. In addition, because the 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. 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 is called *multipath containment frequency* and the frequency of containment *<A, B, X, Y, Z>* is 2. If the nodes on a path cannot appear on more than one path, then the counting predicate is called *containment frequency* and the result is 1. The *edge frequency* in this example is 6 and *node frequency* is 7. 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 values corresponding to each predicate.

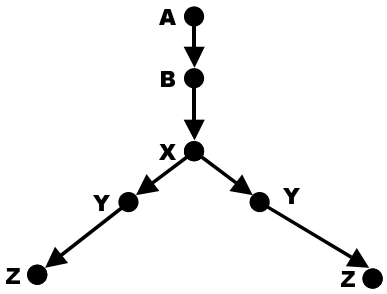

Fig. 3. Shared subcontainments

Table 1. Counting predicates for Figure 3

Counting Predicate	Value
containment frequency	1
multipath containment frequency	2
edge frequency	6
multipath edge frequency	8
node frequency	7
multipath node frequency	10

In determining the support of a containment, to maintain consistency with other data mining methods, support is defined as a percentage indicating how much the frequency measurement relates to the maximum frequency possible for the problem instance. Table 2 shows the percentage of what number, for each counting predicate, corresponds to the support percentage of that counting predicate for a given frequency of that counting predicate.

Table 2. Support measures

Counting Predicate	Support
Containment frequency	percentage of the maximum number of containments of that size that can exist
Multipath containment frequency	percentage of the maximum number of containments of that size that can exist, for all possible root-to-leaf paths in the graph
Edge frequency	percentage of the total number of edges in the graph
Multipath edge frequency	percentage of the total number of edges in all possible root-to-leaf paths of the graph
Node frequency	percentage of the total number of nodes in the graph
Multipath node frequency	percentage of the total number of nodes in all possible root-to-leaf paths of the graph

3.3.2 Multipath Counting Predicates

When would a multipath counting predicate be favored over its non-shared counterpart? A non-shared counting predicate only 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 (3*4*2).

Likewise, the node frequency is 9, and in contrast the multipath node frequency is 72 (3*24). In certain problem domains, the fact that there is overlap between several instances of the same containment is useful information. Suppose that interval event *B* is a disk failure, interval event *A* is a network adapter failure, and interval event *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.

3.4 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; this is the approach used in this paper. A path is described by the sequence of node types in the path. Because there is a one-to-many mapping from the node types to the transaction ID's, a path can exist multiple times in the entire graph. This graph can be traversed using lattice traversal algorithms, or it can be stored in relational database tables and mined using SQL statements.

3.4.1 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 the program is following. Path *TP* is *<tp1, tp2, tp3, …, tpn>*, where *tp1* is the topmost node in the path and *tpn* is the current node (can be internal or leaf node) being visited by the traversal algorithm. By definition, *tp1* has no parent and hence, there is no interval event which contains *tp1*. For each subpath (containment) of *TP* of the form *TPS* in *{<tp(n-1), tpn>, <tp(n-2), tp(n-1), tp>, …, <tp1, tp2, …, tp(n-1), tpn>}*, increment this subpath's counter in the path list *PL* which indicates the number of times that this path (containment) appears. When the entire lattice has been traversed, the paths in *PL* that satisfy the counting predicates (such as *containment frequency >= minimum mining containment frequency*) are presented to the user. This exhaustive counting method will find all possible containments. Herein lies the disadvantage: the number of frequent containments will typically be a small (or very small) subset of all the possible containments, so 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.

3.4.2 Growing Snake Traversal, Shared Node Multiple Containments

Unlike several other data mining methods, when finding frequent containments it is not always possible to prune the search space by using mining results of previous iterations. A corresponding statement, if it held, would be the fact that if a containment *CSUB* has frequency *CSUB.FREQ* for a given counting predicate, then any containments *CSUPER* of which *CSUB* is a subcontainment possess the following property: *CSUPER.FREQ* <= *CSUB.FREQ*. Unfortunately, this property can not be 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 the other 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 as in the naive algorithm, except that only paths of the form <*tp(n-1), tpn*> are enumerated. When all such containments have been found, only those that satisfy the selected counting predicates are retained. Multiple counting predicates can be mixed in a boolean expression, forming a *counting predicate function* to be satisfied by each mined containment. Allowing this freedom for the user broadens the applications of the mining method because the user can decide what counting predicates or counting predicate function(s) must be met by a mined containment in order for it to be considered useful knowledge. Next, containments of size 3 (having form <*tp(n-2), tp(n-1), tpn*>) are enumerated and the same counting predicate function is applied to select useful containments. This is repeated until the maximum containment size is reached. Algorithm 1 contains the details.

Algorithm 1.

```
Input: Containment graph CG, containment predicate
function CPF
Output: Set FINAL_CONT of mined containments
containment_bucket array CA[] (each element containing
CASIZE containments)
containment_bucket FINAL_CONT
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 CCL's in all CA buckets into the FINAL_CONT
  bucket, putting in only those that meet the
  criteria of sufficient frequency, sufficient node
  frequency, sufficient edge frequency, and/or other
  counting predicate(s) (an n-way merge is used to merge
```

```
  the buckets, or an iteration of 2-way merges could also
  be used)
- delete all containments in CA
- endfor
```

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 RAM (Random Access Memory) in today's computers. 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. Randomly accessing such a structure on a computer with sufficient disk space to store it but not enough RAM for it all to be on-line at once will cause thrashing, 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 shown, our implementation of the algorithm ensures that a containment is not counted twice by pruning paths which exist entirely within subsections of the graph which have already been visited. For edge frequency and node frequency counting predicates, the small number of duplicate edges and nodes that arise during the merge step (as a result of paths which are partially in an 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 will permit 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. Consequently, the section of the containment graph being mined is built dynamically as access to it is required. Our algorithm already resorts to merging for generating the mined containments, so a combination of these two techniques yields an algorithm that is limited only by available secondary storage. Furthermore, the data access and generation pattern (if using multiple 2-way merges) is sequential, so a group of devices that support sequential access, such as tape drives, could also be used by the algorithm.

4 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 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, so this quantization method provided good results in this case.

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 3. 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. Table 4 shows the CPU run times for mining the Oracle dataset. 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. 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.

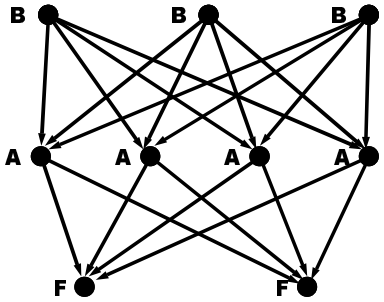

Fig. 4. Multiple shared subcontainments

Table 4. CPU time for execution of mining algorithm vs. number of containments mined for Oracle data

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

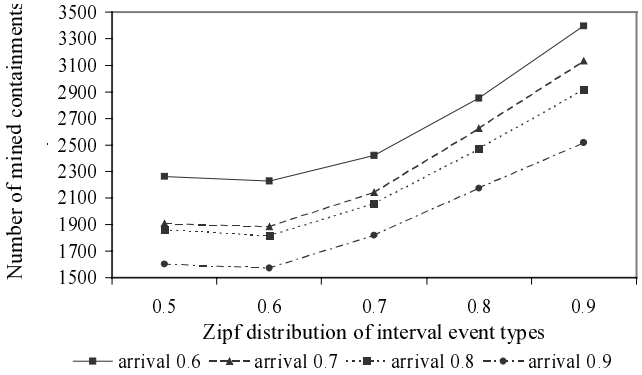

Fig. 5. Synthetic data results

5 Concluding Remarks

Numerous data mining techniques have been developed for conventional time series. In this paper, we investigated techniques for interval time series. We consider an event to be "active" for a period of time, and an interval time series is a sequence of such interval events. We pointed out that existing techniques for conventional time series and sequential patterns cannot be used. Basically, interval time series 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 interval time series.

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

1. Agrawal, R., Faloutsos, C. and Swami, A. Efficiency Similarity Search in Sequence Databases. Proceedings of the Conference of Foundations of Data Organization, 22. 1993.
2. Agrawal, R., Imielinski, T. and Swami, A. Mining Association Rules Between Sets of Items in Large Databases. ACM SIGMOD, 1993.
3. Agrawal, R., Psaila, G., Wimmers, E.L. and Zait, M. Querying Shapes of Histories. Proceedings of VLDB, 1995.
4. Agrawal, R. and Srikant, R. Mining Sequential Patterns. IEEE Data Engineering, 1995.
5. Bohlen, M.H., Busatto, R. and Jensen, C.S. Point- Versus Interval-based Temporal Data Models. IEEE Data Engineering, 1998.
6. Chakrabarti, S., Sarawagi, S. and Dom, B. Mining Surprising Patterns Using Temporal Description Length. Proceedings of the 24th VLDB Conference, 1998.
7. Das, G., Lin, K., Mannila, H., Renganathan, G. and Smyth, P. Rule Discovery From Time Series. The Fourth International Conference on Knowledge Discovery & Data Mining, 1998.
8. Morimoto, Y., Ishii, H. and Morishita, S. Efficient Construction of Regression Trees with Range and Region Splitting. Proceedings of the 23rd VLDB Conference, 1997.
9. Mannila, H., Toivonen, H. and Verkamo, A.I. Discovery of Frequent Episodes in Event Sequences. Data Mining and Knowledge Discovery 1, 259-289, 1997.
10. Rafiei, D. and Mendelzon, A. Similarity-Based Queries for Time Series Data. SIGMOD Record, 1997.
11. Ramaswamy, S., Mahajan, S. and Silberschatz, A. On the Discovery of Interesting Patterns in Association Rules. Proceedings of the 24th VLDB Conference, 1998.
12. Rastogi, R. and Shim, K. Mining Optimized Association Rules with Categorical and Numeric Attributes. IEEE Data Engineering, 1998.
13. Shatkay, H. and Zdonik, S.B. Approximate Queries and Representations for Large Data Sequences. Proceedings of the 12th International Conference on Data Engineering, 1996.
14. Yazdani, N. and Ozsoyoglu, Z.M. Sequence Matching of Images. Proceedings of the 8th International Conference on Scientific and Statistical Database Management, 1996.