Chapter 1

Introduction

A huge amount of data is collected everyday and a real universal challenge is to find actionable knowledge from such large amount of data. Data mining is an emerging research direction to meet this challenge. Data mining techniques can be deployed to search large databases to discover useful information that might otherwise remain unknown.

Many data mining problems involve temporal aspects. Examples range from transaction databases in health care and insurance, stock exchange and customer goods in market sectors, sensor data collected from sensor networks, to scientific databases in geophysics and astronomy. Mining this temporal data poses interesting challenges than mining static data. While the analysis of static data sets often comes down to the question of relating data items, with temporal data there are many additional possible relations.

This thesis focuses on the discovery of temporal rules from sequential data, that is, ordered lists of events (nominal symbols from a particular alphabet) where each event has an associated time of occurrence. In sequential data, events not only can occur at an instant (time point), but also can occur over a time interval. In this thesis, an ordered list of events occurring at an instant is called an *event sequence* and an ordered list of events occurring over a time interval is called an *interval sequence*.

A large number of studies have been concentrated on analysing event sequence data, while the analysis of interval sequence data has received relatively little attention. As a result, there are still many issues that need to be addressed in the area of analysing interval sequence data. Three main issues that will be addressed in this thesis are as follows.

1. The first important issue is that of what constitutes an interesting pattern in data. As an example, the notions of sequential patterns or frequent episodes represent the currently popular structures for patterns in event sequence data. Therefore, the problem of defining structure for interesting patterns in interval sequence data would be a problem that deserves attention.

2. In all data mining applications, the primary constraint is the large volume of data. Hence there is always a need for efficient algorithms. Therefore, designing efficient algorithms for the discovery of patterns in interval sequence data is a problem that would continue to attract attention.

3. Another important issue is that of the analysis of discovered temporal patterns so that one can find interesting patterns from a large number of patterns generated by data mining algorithms. Making sense of such a large number of patterns presents a significant challenge. Therefore, there is a need for tools to assist the user finding interesting patterns from a set of discovered patterns.

The next section introduces the research area of temporal data mining where this thesis belongs. This is followed by describing the objectives, and the structure of the thesis.

1.1 Temporal Data Mining

Data mining is actually an integral part of *Knowledge Discovery in Database* (KDD) process, which is the overall process of converting raw data into useful information (Fayyad, Piatetsky-Shapiro & Smyth 1996). Figure 1.1 shows

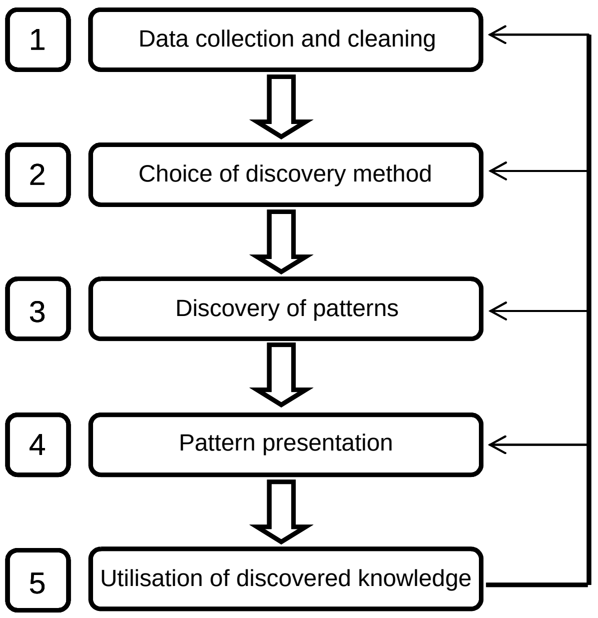

1
Data collection and cleaning
2
Choice of discovery method
3
Discovery of patterns
4
Pattern presentation
5
Utilisation of discovered knowledge

Figure 1.1: Knowledge discovery process

a typical KDD process which consists of five steps (Klemettinen, Mannila & Toivonen 1996):

1. **Data collection and cleaning**: selecting attributes, dealing with errors, identification of the necessary background knowledge, etc.
2. **Choice of pattern discovery method**: deciding on the types of knowledge to be discovered, parameter selection, etc.
3. **Discovery of patterns** (data mining): running algorithms for discovering different types of patterns
4. **Pattern Presentation**: selecting interesting patterns, visualisation of results, etc.
5. **Putting knowledge into use**.

Data mining refers to the third step in this process and is defined as the application of specific algorithms for extracting patterns from data. The KDD process is interactive and iterative. Depending upon the extent to which the results satisfy the user's goals, iteration can be performed between any two steps.

Temporal data mining deals with the problem of mining patterns from temporal data, which can be either *symbolic sequences* or *numerical time series*. It has the capability to look for interesting correlations or rules in large sets of temporal data, which might be overlooked when the temporal component is ignored or treated as a simple numeric attribute (Roddick & Spiliopoulou 2002). The following is the definition of temporal data mining presented in Lin *et al.* (2002).

> *Temporal Data Mining is a single step in the process of Knowledge Discovery in Temporal Databases that enumerates structures (temporal patterns or models) over the temporal data, and any algorithm that enumerates temporal patterns from, or fits models to, temporal data is a Temporal Data Mining Algorithm.*

Currently, temporal data mining is a fast expanding field with many research results reported and many new temporal data mining analysis methods or prototypes developed recently. There are two factors that contribute to the popularity of temporal data mining. The first factor is an increase in the volume of temporal data stored, as many real-world applications deal with huge amount of temporal data. The second factor is the mounting recognition in the value of temporal data. In many application domains, temporal data are now being viewed as invaluable assets from which hidden knowledge can be derived, so as to help understand the past and/or plan for the future (Chen & Petrounias 1998).

Temporal data mining covers a wide spectrum of paradigms for knowledge modeling and discovery. Since temporal data mining is relatively a new field of research, there is no widely accepted taxonomy yet. Several approaches have been used to classify data mining problems and algorithms. Roddick & Spiliopoulou (2002) have presented a comprehensive overview of techniques for the mining of temporal data using three dimensions: data type, mining operations and type of timing information (ordering). On the other hand, Antunes & Oliviera (2001) base their classification on representation, similarity and operations.

Based on its main tasks, temporal data mining can be grouped into five broad categories: prediction, classification, clustering, search and retrieval, and pattern discovery (Laxman & Sastry 2006). This categorization follows the categorization of data mining tasks presented by Han and Kamber (2001), extended to temporal data mining.

Prediction

Prediction is the task of explicitly modeling variable dependencies to predict a subset of the variables from others. The task of time series prediction is to forecast future values of the time series based on its past samples. In order to perform the prediction, one needs to build a predictive model from the data. Koskela *et al.* (1996) have studied neural networks for nonlinear modeling of time series data. The prediction problem for symbolic sequences has been addressed in AI research by Dietterich and Michalski (1985).

Classification

Classification is the task of assigning class labels to the data according to a model learned from the training data where the classes are known. Classification is one of the most common tasks in supervised learning, but it has not received much attention in temporal data mining (Antunes & Oliveira 2001). In sequence classification, each sequence presented to the system is assumed to belong to one of predefined classes and the goal is to automatically determine the corresponding category for a given input sequence. Examples of sequence classification applications include signature verification (Nalwa 1997), gesture recognition (Yamato, Ohya & Ishii 1992), and hand-written word recognition (Kundu, He & Bahl 1988).

Clustering

Clustering[1] is the process of finding intrinsic groups, called clusters, in the data. Clustering of time series (or sequences) is concerned with grouping a collection of time series (or sequences) based on their similarity. Time series clustering has been shown effective in providing useful information in various domains (Liao 2005). For example, in financial data, clustering can be used to group stocks that exhibit similar trends in price movements. Another example could be clustering of fMRI time series for identifying regions with similar patterns of activation (Goutte, Toft & Rostrup 1999). Clustering of sequences is relatively less explored but is becoming increasingly important in data mining applications such as web usage mining and bioinformatics (Laxman & Sastry 2006). A survey on clustering time series has been presented by Liao (2005).

Searching and Retrieval

Searching and retrieval are concerned with efficiently locating subsequences or sub-series (often referred to as queries) in large databases of sequences or time series. In data mining, query based searches are more concerned with the problem of efficiently locating approximate matching than exact matching, known as content-based retrieval. An example of a time series retrieval application is to find out all the days of the year in which a particular stock had similar movements to those of today. Another example is finding products with similar demand cycles. An example of a sequence retrieval is finding gene expression patterns that are similar to the expression pattern of a given gene. In order to address the time series retrieval problem, different notions of similarity between time series and indexing techniques have been proposed. There is considerably less work in the area of sequence retrieval, and the problem is more general and difficult. For more detail about time series and sequence retrieval can be found in Das and Gunopulos (2003).

[1]Clustering is sometimes called unsupervised classification.

Pattern Discovery

Unlike in search and retrieval applications, in the pattern discovery there is no specific query in hand with which to search the database. The objective is simply to discover all patterns of interest. While the other tasks described earlier have their origins in other disciplines like statistics, machine learning or pattern recognition, the pattern discovery task has its origin in data mining itself.

A pattern is a local structure in the data. There are many ways of defining what constitutes a pattern. There is no universal notion for interestingness of a pattern either. However, one concept that is normally used in data mining is that of frequent patterns, that is, patterns that occurs many times in the data. Much of data mining literature is concerned with formulating useful pattern structures and developing efficient algorithms for discovering frequent patterns.

Methods for finding frequent patterns are important because they can be used for discovering useful rules, which in turn can be used to infer some interesting regularities in the data. A rule usually consists of a pair of a left-hand side proposition (the antecedent) and a right-hand side proposition (the consequent). The rule states that when the antecedent is true, then the consequent will be true as well.

It was mentioned above that temporal data can be symbolic sequences or time series. The pattern discovery task typically assumes an underlying symbolic representation. Therefore, to apply the pattern discovery methods on time series data, the time series should be first converted into a discrete representation, for example by first forming subsequences (using a sliding window) and then clustering these subsequences using a suitable measure of pattern similarity (Das, Lin, Mannila, Renganathan & Smyth 1998). Another method can be used by quantizing the time series into levels and representing each level (e.g., high, medium, etc.) by a symbol (Aref, Elfeky & Elmagarmid 2004). A survey on time series abstraction methods can be found in Höppner (2002).

1.2 Research Objectives

1.2.1 Discovery of Point Temporal Rules

A large volume of research has been focused on discovering temporal patterns from the data that contain sequences of events, where events are stamped with, and interpreted as, time points. For example, several models of temporal association rules have been proposed such as interval association rules (Ale & Rossi 2000), calendric association rules (Li, Ning, Wang & Jajodia 2001), or cyclic association rules (Ozden, Ramaswamy & Silberschatz 1998). The data used for the discovery of these temporal association rules contain a set of transactions, each of which has a time associated with it. Furthermore, Agrawal and Srikant (1995) introduced the problem of discovering frequent sequences (sequential patterns) from a set of event sequences, such as customer purchases, web log accesses, DNA sequences, and so on. A number of algorithms have been proposed to discovers sequential patterns, but in general can be categorized into two different approaches, i.e., Apriori-based (Agrawal & Srikant 1994) and pattern-growth (Han, Pei & Yin 2000) approaches. Another formulation of the problem of discovering frequently occurring patterns in sequential data was introduced by Mannila *et al.* (1995). The discovered patterns called episodes are mined over a single sequence of events, not a set of event sequences.

In this thesis, previous studies on the discovery of temporal patterns from event sequences serve as a foundation for conducting further research on the pattern discovery from interval sequence data. Many methods and techniques that have been proposed in this research area provide ideas that can be applied or extended to interval sequence data. Furthermore, as a result of research conducted in this area, this thesis proposes a new temporal association rule model called *inter-transaction relative temporal association rules*, representing the associative relationships among items from different transactions (Winarko & Roddick 2003). The problem of discovering relative temporal association rules has not been proposed before.

1.2.2 Discovery of Interval Temporal Rules

The importance of analysing interval data is highlighted by Böhlen *et al.* (1998), who argue that for some applications events are better treated as intervals rather than time points. For example, consider a medical database in which a patient's treatment is regarded each time as an event time-stamped with a time point, indicating the time of the treatment. While it is useful, it could be advantageous to interpret the treatment as an interval, representing the period between the first and last occurrences of the treatment.

Several studies have been published about analysing interval sequence data. Most studies concentrate on extracting temporal patterns from a symbolic interval sequence originated from multivariate time series (Guimarães & Ultsch 1999, Villafane, Hua, Tran & Maulik 2000, Höppner 2001, Mörchen, Ultsch & Hoos 2004). The time series can be transformed to labeled intervals using segmentation and feature extraction, for example, via neural networks (Guimarães & Ultsch 1999). A set of interval patterns can also be extracted from a database of short interval sequences (Kam & Fu 2000, Papapetrou, Kollios, Sclaroff & Gunopulos 2005).

The analysis of interval sequence data is new research area and has not been fully explored. As mentioned above, the first important issue in analysing interval sequence data is that of what constitutes an interesting pattern in data. To address this problem, this thesis proposes the discovery of *richer temporal association rules* from interval sequence databases containing a set of interval sequences. Furthermore, to address the second issue regarding the algorithms for the discovery of patterns, this thesis proposes a new algorithm, ARMADA, for discovering richer temporal association rules. ARMADA is a novel algorithm which requires at most two database scans and does not require candidate generation or database projection (Winarko & Roddick 2005, Winarko & Roddick 2007).

1.2.3 Retrieval of Discovered Temporal Rules

Like other techniques for the rule discovery, ARMADA could generate a large number of richer temporal association rules. Finding interesting rules is a difficult task when the number of rules is large. Providing the user with a long list of rules hardly gives a useful overview of them. This problem is made worse as richer temporal association rules have more complex structures than, for example, association rules.

Several techniques commonly called the *post-processing techniques* have been proposed to assist the user in the process of finding interesting association rules. One approach is to use some interestingness measure to prune the discovered association rules and/or to use the user's domain knowledge to help the user to identify interesting association rules (Silberschatz & Tuzhilin 1995, Padmanabhan & Tuzhilin 1998, Liu, Hsu & Ma 1999). Another approach is to use templates (Klemettinen, Mannila, Ronkainen, Toivonen & Verkamo 1994), or data mining queries (Tuzhilin & Liu 2002), which allows the user to focus only on some subset of the rules that are of interest to the user by asking appropriate queries.

This thesis addresses the post-processing problem by proposing a retrieval system to facilitate the selection of interesting rules during the post-processing of discovered richer temporal association rules. A query language TAR-QL is proposed for specifying the criteria of rules to be retrieved. Furthermore, methods for evaluating queries are developed, especially for dealing with the queries involving the format of richer temporal association rules. The proposed methods employ signature file based indexes to speed up the query evaluation.

1.3 Organization of the Thesis

Apart from this introduction and the conclusion, the thesis contains six main chapters, which is divided into three main parts, each of which contains a review of the pertinent area and a contribution.

The first part discusses the discovery of temporal rules from event sequence and contains two chapters. Chapter 2 presents a comprehensive review of current work in the discovery of temporal association rules, sequential patterns, episodes, and periodic patterns. Chapter 3 describes the discovery of relative temporal association rules.

The second part discusses the discovery of temporal rules from interval sequences and consists of two chapters. Chapter 4 presents a survey of previous work on on the discovery of temporal patterns from interval sequences. Chapter 5 describes the discovery of richer temporal association rules from databases containing a set of interval sequences.

The third part presents the retrieval of discovered rules. Chapter 6 reviews the index structures for improving DBMS performance for set-oriented and sequence-oriented queries. This chapter first reviews the use of inverted files and signature files for set retrieval. Then, it describes how the use of signature files for set retrieval is extended for sequential pattern retrieval. Chapter 7 describes the retrieval system to facilitate the selection of interesting rules in the post-processing of discovered richer temporal association rules.

Finally Chapter 8 concludes the thesis, consisting of summary, contributions, and a discussion of areas of further research.

Chapter 2

Review of Mining Time Point Patterns

A large number of previous studies on discovering temporal patterns from sequential data have focused on data that are stamped with, and interpreted as, time points. Several variations of the classical association rule mining (Ceglar & Roddick 2006) have been proposed under the name of temporal association rules mining, whose task is to discover temporal association rules from a set of timestamped transactions. Abstractly, such data can be viewed as a sequence of event sets; and the temporal association rule represents a temporal relationship between a set of events.

Agrawal and Srikant (1995) introduced the discovery of sequential patterns from a set of data sequences. While in temporal association rule mining each transaction is normally treated individually with no record of the associated object, in sequential pattern mining each transaction is associated with a particular object at a given time. Each data sequence consists of an ordered list of transactions belonging to a particular object (e.g. customer or client who purchased the transactions). Even though the framework is described using an example of mining a database of customer transaction sequences, the concept of sequential patterns is quite general and can be used in many other areas, such as analysing web access sequences and protein sequences. In general, a sequential pattern represents a correlation among the events in the sequence.

Another framework for discovering temporal patterns in sequential data is the mining of frequent episodes (Mannila, Toivonen & Verkamo 1995). In this framework, the data are given in a single long sequence of events, and the task is to discover frequent episodes in the sequence, which are later used to generate episode rules. A frequent episode is essentially a frequent sequence, but instead of being frequent across many data sequences, an episode is frequent within one sequence. Later, Han *et al.* (1998) introduced the discovery of partially periodic patterns from a long sequence of events.

This chapter reviews previous studies on the discovery of temporal patterns from point-based sequential data. In particular, the review focuses on the discovery of temporal association rules, sequential patterns, episodes and periodic patterns. The chapter is structured as follows. Section 2.1 provides a survey of different types of temporal association rule and methods to discover the rules. Section 2.2 reviews algorithms for solving the basic sequential pattern mining problem. Various extensions to the sequential pattern mining problem are also discussed. Section 2.3 describes two basic algorithms for mining frequent episodes, highlights several extensions to episode mining model, and briefly discusses periodic pattern mining.

2.1 Mining Temporal Association Rules

Several types of temporal association rules have been proposed as extensions of the classical association rules (Agrawal, Imielinski & Swami 1993). Ozden *et al.* (1998) introduced cyclic association rules and presented two algorithms to discover such rules. A cyclic association rule is an association rule that occurs periodically over time. However, periodicity has limited power in describing real-life variations. For instance, periodicity cannot describe real-life concepts such as *the first business day of every month* in which the distances between two such consecutive business days are not always the same. Ramaswamy *et al.* (1998), therefore, consider the discovery of association rules that hold during the time intervals described by a calendar algebraic expression. The calendar algebra adopted is considered more powerful than periodicity in describing time intervals of interest.

Although the work of Ramaswamy *et al.* (1998) is more flexible than that of Ozden *et al.* (1998), it requires the user to define the calendar algebraic expression. To provide more flexibility to the user, Li *et al.* (2001) propose a calendar schema as a framework for discovering temporal association rules.

Ale and Rossi (2000) consider the discovery of association rules hold during the lifetime (lifespan) of items involved in the rules. An item's lifespan is a period between the first and the last time the item appears in transactions in the dataset. The lifespan is intrinsic to the data so that the users do not need to define it. Similarly, Lee *et al.* (2001) consider the problem of mining general temporal association rules in publication databases. A publication database is a set of transactions where each transaction contains an individual exhibition period. Zimbrão *et al.* (2002) combine the work of Li *et al.* (2001) and Ale and Rossi (2000) and propose a new approach to discover calendar-based association rules with an item's lifespan restriction.

Chen and Petrounias (2000) present the discovery of the longest interval and the longest periodicity of association rules. Rainsford and Roddick (1999) propose to add temporal features to association rules by associating a conjunction of binary temporal predicates that specify the relationships between the timestamps of transactions. Subsequently, Rainsford and Roddick (2000) provide visualisation methods for viewing these temporal predicate association rules.

This section reviews temporal association rule mining. It first describes the classification of temporal association rules. Based on this classification, four types of temporal association rules are discussed by describing their mining problems and the algorithms to discover the rules.

2.1.1 Taxonomy of Temporal Association Rules

Previous studies on temporal association rules mining have focused on developing temporal association rule models. The proposed models can be classified based on three different aspects: measures of interestingness used, temporal feature associated with the rules, and algorithms used to discover the rules. The result of this classification is summarized in Tables 2.1 and 2.2.

Measures of Interestingness

In classical association rule mining, two commonly used measures of interestingness are *support* and *confidence* (Agrawal et al. 1993). Both reflect the usefulness and certainty of discovered rules (Han & Kamber 2001). However, for some of the temporal association rule models discussed here, these two measures are considered insufficient and additional measures are proposed, for example, *temporal support*, *frequency*, and *temporal confidence* (see Table 2.1).

Temporal support is used to filter the items with high support but short life. The combination of support and temporal support is used to determine if an itemset is frequent (Ale & Rossi 2000). Frequency measures the proportion of the intervals during which the rules hold with respect to a set of given time intervals (Chen & Petrounias 2000). Temporal confidence determines the strength of the temporal relationships between temporal items in the rule (Rainsford & Roddick 1999).

Temporal Features of the Rules

According to Chen and Petrounias (2000), a temporal association rule can be represented as a pair $< AR, TF >$, where AR is an association rule and TF is a temporal feature belonging to AR. Depending on the interpretation of the

Table 2.1: Temporal association rule classification

Author	Measures			Temporal association rule class			
	Supp.	Conf.	Extra measure	Interval	Cyclic	Calendric	Binary pred.
Ale & Rossi (2000)	✓	✓	Temporal support	✓			
Ozden *et al.* (1998)	✓	✓			✓		
Chen & Petrounias (2000)	✓	✓	Frequency	✓	✓		
Ramaswamy *et al.* (1998)	✓	✓				✓	
Li *et al.* (2001)	✓	✓				✓	
Rainsford & Roddick (1999)	✓	✓	Temporal confidence				✓
Lee *et al.* (2001)	✓	✓		✓			
Zimbrao *et al.* (2002)	✓	✓		✓		✓	

temporal feature TF, a temporal association rule $< AR, TF >$ can be classified as:

- a *universal* association rule if $\phi(TF) = \{T\}$, where T represents the time domain;
- an *interval* association rule if $\phi(TF) = \{itvl\}$, where $itvl \subset T$ is a specific time interval;
- a *periodic* association rule if $\phi(TF) = \{p_1, p_2, \ldots, p_n\}$, where $p_i \subset T$ is a periodic interval in cycles;
- a *calendric* association rule if $\phi(TF) = \{cal_1, cal_2, \ldots, cal_m\}$, where $cal_j \subset T$ is a calendric interval in a specific calendar.

This classification method is used to classify the temporal association rule models discussed in this chapter. The universal association rule class represents a class of classical (non-temporal) association rules (Agrawal et al. 1993), so it will not be discussed further. Moreover, there is a type of temporal association rule that is not covered by this classification and needs to be added, that is, *binary predicate* association rules (Rainsford & Roddick 1999). As a result, Table 2.1 shows four classes of temporal association rules, namely *interval*, *cyclic*, *calendric*, and *binary predicate* association rules. As can be seen in the table, a model can be included into more than one class, depending on the temporal feature associated with the association rules.

In the interval association rule class, a time interval has several interpretations. It can be interpreted as a lifespan of items in the rules (Ale & Rossi 2000, Zimbrão, de Souza, de Almeida & da Silva 2002), the maximum common exhibition period of items in the rules (Lee, Lin & Chen 2001), or any specific time interval (Chen & Petrounias 2000). Section 2.1.2 describes in more detail the interval association rule model proposed by Ale and Rossi (2000).

The work of Ramaswamy *et al.* (1998) and Li *et al.* (2001) is classified as calendric association rules since both use the notion of calendars to describe phenomena of interest in association rules. Ramaswamy *et al.* (1998) introduced

Table 2.2: Temporal association rule algorithms and models

Author	Algorithms		Temporal association rule model
	Apriori-based	Other	
Ale & Rossi (2000)	✓		$X \rightarrow Y, [t_1, t_2]$ $[t_1, t_2]$ is a lifespan of $X \cup Y$
Ozden *et al.* (1998)	✓		$X \rightarrow Y, c = (l, o)$ c is a cycle with length l and offset o
Chen & Petrounias (2000)	✓		$X \rightarrow Y, TF$ $\phi(TF) = \{p_1, p_2, \ldots, p_n\}$, p_i is a periodic interval
Ramaswamy *et al.* (1998)	✓		$X \rightarrow Y, C$ C is a calendar expression
Li *et al.* (2001)	✓		$X \rightarrow Y, e$ e is a calendar pattern
Rainsford & Roddick (1999)	✓		$X \rightarrow Y \wedge P_1 \wedge P_2 \ldots P_n$ P_i is a binary temporal predicate
Lee *et al.* (2001)	✓	PPM	$X \rightarrow Y, (t, n)$ (t, n) is the max. common exhibition period
Zimbrao *et al.* (2002)	✓		$X \rightarrow Y, e, [t_1, t_2]$ e is a calendar pattern $[t_1, t_2]$ is a lifespan of $X \cup Y$

a calendar algebra concept to define a set of time intervals that the algorithm considers in discovering the association rules. The calendar algebra is based on the work reported by Allen (1983) and Leban *et al.* (1986), and the implementation reported by Chandra *et al.* (1994). On the other hand, Li *et al.* (2001) use calendar schemas, instead of calendar algebra expressions. They consider all possible temporal association rules valid during time intervals specified by a calendar schema, as opposed to simply complying with user-given temporal expressions. This model will be described further in Section 2.1.4.

Mining Algorithms

Most of the algorithms for discovering temporal association rules are *Apriori*-based algorithms[1]. The only algorithm that is not based on the Apriori is the Progressive Partition Miner (PPM) algorithm (Lee et al. 2001). The classification of the algorithms is shown in Table 2.2. The table also shows the format of each temporal association rule model.

The next four subsections discuss the Apriori-based algorithms to discover interval association rules (Section 2.1.2), cyclic association rules (Section 2.1.3), calendric association rules (Section 2.1.4), and temporal predicate association rules (Section 2.1.5). To facilitate the discussion, some common notations are defined below.

Notations

Definition 2.1 (Database) Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of literals, called *items*. Let D be a set of database transactions, where each transaction s consists of a set of items such that $s \subseteq I$. Each transaction s is associated with an identifier (TID) and a timestamp which represents valid time of a transaction s.

[1]Note that the Apriori algorithm refers to the algorithm to discover association rules (Agrawal & Srikant 1994)

A non empty set of items is called an *itemset*. An itemset i is denoted by $(i_1 i_2 \cdots i_m)$, where i_j is an item. An itemset that contains k items called an k-itemset. C_k denotes a set of candidate k-itemsets, while F_k denotes a set of frequent k-itemsets.

2.1.2 Interval Association Rules

This subsection discusses the interval association rule model proposed by Ale and Rossi (2000), which discovers association rules during the lifetime of items involved in the rules. The model is motivated by the observation that it is possible to have association rules with a high confidence but with little support. Normally, such rules may not be discovered as their support is less than the minimum support threshold. To overcome this, the model suggests that the denominator in the support calculation is not based on the total number of transactions in the database, but on the total number of transactions belonging only to the lifetime of items in the rules. As a result, each generated rule has an associated time frame, corresponding to the lifetime of the items participating in the rule.

TID	Timestamp	Items
s_1	1	a, c, f, h
s_2	2	a, b, c, g
s_3	3	b, c, d, g
s_4	4	a, c
s_5	5	c, d, e, h
s_6	6	a, d, f, g

Figure 2.1: Example database

Let T = $\{\ldots, t_o, t_1, t_2, \ldots\}$ be a set of time instants, countably infinite, over which a linear order $<_T$ is defined. For $t_i, t_j \in T$, $t_i <_T t_j$ means that t_i occurs before t_j. Let D be a transaction database (defined in Definition 2.1). Every item in D has a period of life (lifespan) in the database, which represents the time in which the item is relevant to the user. The *lifespan* of an item x is represented by a closed interval $[t_i, t_j]$, where $t_i <_T t_j$, and is denoted by l_x. As an example, from the transaction dataset in Figure 2.1, the lifespans of items a, b and c are l_a = [1, 6], l_b = [2, 3] and l_c = [1, 5], respectively. The lifespan of an itemset X is defined as an intersection of the items' lifespan in the itemset, that is, the lifespan of an itemset $X = \{i_1 i_2 \dots i_m\}$ is $l_X = l_{i_1} \cap l_{i_2} \cap \dots \cap l_{i_m}$. In the case $X = (ac)$, its lifespan is $l_X = l_a \cap l_c = [1,6] \cap [1,5] = [1,5]$.

In association rule mining, the support of an itemset is defined as the fraction of all transactions in the database in which the itemset is contained. In this framework, the definition of the support takes into consideration the itemset's lifespan. Thus, the *support* of an itemset X in the database D (over its lifespan l_X), denoted by $sup(X, l_X, D)$, is defined as the number of transactions that contain X divided by the number of transactions whose timestamp $t_i \in l_X$. An itemset $X = (ac)$ has the support of 0.6, since it is contained in 3 of the 5 transactions in its lifespan (Figure 2.1).

It is possible for an itemset to have high support but short life. From the example dataset, an item e has support of 100% but its lifespan is short, that is, $|l_e| = 1$. The concept *temporal support* is introduced and used to filter such items (or itemsets). The temporal support of an itemset X is defined as the amplitude of the lifespan of X, i.e., $|l_X|$. The combination of support and temporal support is used to determine if an itemset is frequent or not. Therefore, given the minimum support *minsup* and the minimum temporal support *min_tsup*, X is a frequent itemset in its lifespan l_X if its support and temporal support are greater than or equal to *minsup* and *min_tsup*, respectively. Let the *minsup* = 0.50 and *min_tsup* = 3, the itemset $X = (ac)$ is frequent because its support $sup(X, l_X, D) = 0.6$ is greater than *minsup* and its temporal support $|l_X| = 5$ is greater than *min_tsup*.

An interval association rule is a rule of the form $X \rightarrow Y$ $[t_1, t_2]$, where $[t_1, t_2]$ is a time frame corresponding to the lifespan of $X \cup Y$. The support of the rule is the support of $X \cup Y$ (over $l_{X \cup Y}$) and the temporal support of the rule is $|l_{X \cup Y}|$.

The confidence of the rule in D (over $l_{X \cup Y}$) is defined as

$$conf(X \rightarrow Y, l_{X \cup Y}, D) = \frac{sup(X \cup Y, l_{X \cup Y}, D)}{sup(X, l_{X \cup Y}, D)}$$

Given the transaction dataset D (as defined in Definition 2.1), the minimum support *minsup*, the minimum temporal support *min_tsup*, and the minimum confidence *minconf*, the problem of mining interval association rules is to find all association rules that have at least the given support, temporal support and confidence.

Ale and Rossi (2000) assert that any existing algorithms for association rule discovery, for example Apriori (Agrawal & Srikant 1994) and FP-growth (Han, Pei & Yin 2000), can be modified to discover interval association rules. Algorithm 2.1 shows pseudo code of an Apriori based algorithm for generating frequent itemsets (in their lifespan), which consists of several passes. The first pass is used to obtain a set F_1 of frequent 1-itemsets. In subsequent passes, say a pass $k > 1$, a set C_k of candidate k-itemsets is generated from a set F_{k-1} of frequent $(k-1)$-itemsets, using Apriori's method for candidate generation. If a candidate k-itemset U is obtained by joining $(k-1)$-itemsets V and W, the lifespan of U is the intersection of the lifespans of V and W. Then, the database is scanned to determine a set F_k of frequent k-itemsets. When scanning the database, the algorithm not only counts the occurrences of itemsets and records their lifespans but also counts the number of transactions in their lifespans, which is required for calculating the support of itemsets.

Once a set F of all frequent itemsets has been found, association rules can be discovered from F, as follows. For every frequent itemset $Z \in F$, find the rules $X \rightarrow (Z - X)[t_1, t_2]$[2] such that the rule confidence, $\frac{sup(Z, l_Z, D)}{sup(X, l_Z, D)}$, exceeds the *minconf*, for each $X \subset Z$. In computing the rule confidence, the value of $sup(X, l_Z, D)$ is estimated by using the value of $sup(X, l_X, D)$. The reason for doing this is to avoid recalculating the support for $(2^k - 2)$ itemsets $X \subset Z$ in l_Z.

[2] $[t_1, t_2]$ is the lifespan of Z.

```
Input: A database $D$, $minsup$, and $min\_tsup$
Output: A set of frequent itemsets $F$
1: $F_1 = \{\text{frequent 1-itemsets}\}$
2: for $(k = 2; F_{k-1} \neq \emptyset; k{+}{+})$ do
3:    Create $C_k$ from $F_{k-1}$
4:    for all transaction $s \in D$ do
5:       Count the support of all candidates $X \in C_k$
6:    end for
7:    $F_k = \{X \in C_k | sup(X, l_X, D) \geq minsup$ and $|l_X| \geq min\_tsup\}$
8: end for
9: $F = \bigcup_k F_k$
```

Algorithm 2.1: Pseudo code for generating frequent itemsets (in their lifespan)

2.1.3 Cyclic Association Rules

This subsection describes the concept of cyclic association rules and the discovering of such rules as described by Ozden *et al.* (1998). Unlike classical association rules, cyclic association rules are not required to hold for the entire transactional database, but rather only for transactional data in a particular periodic time interval.

The model assumes that the unit of time (hour, day, week, month, etc.) is given (e.g., by the user). Suppose u is a given unit of time, the i^{th} time unit is denoted by u_i, $i \geq 0$, and corresponds to the time interval $[i \cdot u, (i+1) \cdot u]$. Given a transactional database D, each time interval u_i corresponds to the disjoint segment of the database D_i, that is, a set of transactions in D whose timestamps are within time unit (interval) u_i. A *cycle* c is a tuple (l, o), where l (multiples of the time unit) is the *length* of the cycle, o is an offset (the first time unit in which the cycle occurs), and $0 \leq o < l$. As an example, if the unit of time is an *hour*, every fourth hour starting from the third hour (i.e, 3^{rd}, 7^{th},...) is part of a cycle $c = (4, 3)$.

A cyclic association rule is a rule of the form $X \rightarrow Y$ $c = (l, o)$, where $c = (l, o)$ is the cycle in which the rule holds. The rule holds in a cycle $c = (l, o)$ if it holds in every l^{th} time unit starting with time unit u_o[3]. For example, if the rule holds

[3]An association rule holds in time unit u_i means that it holds in D_i.

in time units $3^{rd}, 7^{th}, 11^{th}, \cdots$, then the rule holds at a cycle $c = (4, 3)$. It is possible for an association rule to have more than one cycle. As an example, if the unit of time is an hour and the rule holds during the interval *8am-9am* and *4pm-5pm* every day (i.e., every 24 hours), then it has two cycles $c_1 = (24, 8)$ and $c_2 = (24, 16)$. Given a set of cycles, a *large* cycle is a cycle that is not multiple of any other cycles in the set. A cycle (l_i, o_i) is a *multiple* of another cycle (l_j, o_j) if l_j divides l_i and $o_j = o_i \bmod l_j$. As an example, a cycle $(24, 15)$ is a multiple of a cycle $(12, 3)$.

The problem of mining cyclic association rules is to discover the rules with large cycles. Two algorithms have been proposed to discover cyclic association rules. The first algorithm, called the *sequential algorithm*, performs association rule mining and cycle detection independently. The second algorithm, called the *interleaved algorithm*, employs optimisation techniques for discovering cyclic association rules. The sequential algorithm is described in more detail below.

Generating cyclic association rules using the sequential algorithm consists of two phases. In the first phase, association rules are generated for each time unit using one of the existing methods. Once the rules of all the time units have been discovered, the second phase of the algorithm is applied to detect (large) cycles of each association rule, using the cycle detection procedure. In this procedure, each association rule is represented as a binary sequence in which '1's correspond to the time units in which the rule holds and '0's correspond to the time units in which the rule does not hold. For example, the binary sequence 001100010101 represents the association rule that holds in D_2, D_3, D_7, D_9, and D_{11}. Given a binary sequence and the maximum cycle length of interest m, a procedure of detecting cycles consists of two steps.

In the first step, the sequence is scanned. Each time '0' is encountered at a position i, candidate cycles $c = (j, i \bmod j)$, $1 \leq j \leq m$ are eliminated from the set of candidate cycles. If the maximum cycle length of interest is m, the set of candidate cycles contains all possible cycles $c(l, o)$, where $1 \leq l \leq m$, $1 \leq o < l$. Let

$m = 4$, then the set of candidate cycles is $\{c(1,0), c(2,0), c(2,1), \cdots, c(4,0), \cdots, c(4,3)\}$. Given a sequence 001100010101, a digit '0' is found at position 0, thus cycles $c(1,0)$, $c(2,0)$, $c(3,0)$ and $c(4,0)$ are eliminated from the set of candidate cycles. This step is completed whenever the last bit of the sequence is scanned or the set of candidate cycles becomes empty, whichever is first. From the example, after the eleventh bit is scanned, the only remaining cycle is $c(4,3)$, representing a rule that holds in every fourth time unit starting from the time unit u_3, followed by u_7 and u_{11}.

In the second step, large cycles are detected from the set of remaining cycles by eliminating cycles that are not large, that is, starting from the shortest cycle, for each cycle $c_i = (l_i, o_i)$ in the set, eliminate cycle $c_j = (l_j, o_j)$ that is multiple of $c_i = (l_i, o_i)$ from the set. Since there is only one cycle $c(4,3)$ remains, this cycle is obviously large.

2.1.4 Calendric Association Rules

This subsection discusses the calendric association rule model as described by Li *et al.* (2001). This model is more general than the one proposed by Ramaswamy *et al.* since it provides a framework for discovering association rules over a calendar schema, instead of a calendar algebra expression.

A *calendar schema* is a relational schema $R = (f_n\colon \Delta_n, f_{n-1}\colon \Delta_{n-1}, \ldots, f_1\colon \Delta_1)$, where an attribute f_i is a calendar unit name such as year, month and week, and a domain Δ_i is a finite subset of the positive integers. Given a calendar schema R, a *calendar pattern* on the calendar schema R is a tuple on R of the form $\langle d_n, d_{n-1}, \ldots, d_1\rangle$, where $d_i \in \Delta_i$ or the wild-card symbol '$*$' (representing the word *every*). Each calendar pattern represents a set of time intervals.

As an example, given a calendar schema (week, day, hour), a calendar pattern $\langle 1, *, 10\rangle$ represents the 10^{th} *hour of every day of week 1*, and similarly $\langle 1, 1, 10\rangle$ represents the 10^{th} *hour of day 1 of week 1*. A calendar pattern with k wild-card symbols is called a *k-star calendar pattern*, denoted e_k, and a calendar pattern with at least one wild-card symbol is called a *star calendar pattern*. A calendar pattern with no wild-card symbol is called a *basic time interval*, denoted e_0. Given calendar patterns e and e' (in the same calendar schema), e *covers* e' if the set of time intervals of e' is a subset of that of e. From the example, a calendar pattern $\langle 1, *, 10\rangle$ covers $\langle 1, 1, 10\rangle$.

A *calendric association rule* over calendar schema R is a pair (r, e), where r is an association rule and e is a calendar pattern on R. There are two classes of calendric association rules: *precise-match* association rules and *fuzzy-match* association rules. Precise-match association rules require the rules to hold during every interval in e, while fuzzy-match association rules require the rules to hold during a significant fraction of these intervals. More formal definition of precise-match and fuzzy-match association rules can be stated as follows.

Definition 2.2 (Precise-match association rule) Given a calendar schema R, a calendar pattern e on R, and a transaction database D, a **precise-match association rule** (r, e) holds in D if and only if the association rule r holds in $D[e_0]$ for each basic time interval e_0 covered by e. $D[e_0]$ denotes a set of transactions in D whose timestamps are covered by e_0.

Definition 2.3 (Fuzzy-match association rule) Given a calendar schema R, a calendar pattern e on R, and a transaction database D, and a real number m (*match ratio*), where $0 < m < 1$, a **fuzzy-match association rule** (r, e) holds in D if and only if the association rule r holds in $D[e_0]$ for at least $(100 \cdot m)\%$ of the basic time intervals e_0 covered by e.

Given a calendar schema R, the problem of mining calendric association rules is to discover precise-match or fuzzy-match association rules for all possible star calendar patterns in R. Similar to mining association rules, mining calendric association rules can also be divided into two subproblems: first, finding all frequent itemsets for all calendar patterns; and second, generating calendric association rules using the frequent itemsets and their calendar patterns. Two algorithms are proposed for finding calendric frequent itemsets; both are based on the Apriori algorithm. The first algorithm, *direct-Apriori*, treats each basic time interval individually, while the second one, *temporal-Apriori*, is optimised by exploiting the relationship among calendar patterns. The outline the direct-Apriori algorithm to generate precise-match frequent itemsets for all possible star calendar patterns on R is shown in Algorithm 2.2.

```
Input: A database $D$, $minsup$, a calendar schema $R$
Output: $F(e)$, a set of frequent itemsets, for all star calendar patterns $e$ on $R$
 1: for all basic time interval $e_o$ do
 2:   $F_1(e_o) = \{$frequent 1-itemsets in $D[e_o]\}$
 3:   for all star calendar patterns $e$ that covers $e_o$ do
 4:     Update $F_1(e)$ using $F_1(e_o)$;
 5:   end for
 6: end for
 7: for ($k = 2$; $\exists$ a star calendar pattern $e$ such that $F_{k-1}(e) \neq \emptyset$;$k++$) do
 8:   for all basic time interval $e_o$ do
 9:     Generate $C_k(e_o)$ from $F_{k-1}(e_o)$
10:     for all transactions $s \in D[e_o]$ do
11:       Count the support of all candidate $X \in C_k(e_o)$
12:     end for
13:     $F_k(e_o) = \{X \in C_k(e_o) | sup(X) \geq minsup\}$
14:     for all star patterns e that covers $e_o$ do
15:       Update $F_k(e)$ using $F_k(e_o)$;
16:     end for
17:   end for
18: end for
19: $F(e) = \bigcup_k F_k(e)$
```

Algorithm 2.2: Pseudo code for generating precise-match frequent itemsets

The notations used in the algorithm are defined as follows. $C_k(e_o)$ denotes a set of candidate k-itemsets for a basic time interval e_0. $F_k(e_0)$ and $F_k(e)$ denote the set of frequent k-itemsets for a basic time interval e_0 and a star calendar pattern e, respectively.

The algorithm works in several passes. In each pass, the basic time intervals in the calendar schema are processed one by one. For instance, if the calendar schema $R = (week\colon \{1, 2, 3\}, day\colon \{1, 2, \cdots, 7\})$, then there are totally 21 basic time intervals in R to process, from $\langle 1, 1\rangle$ to $\langle 3, 7\rangle$. Given a basic time interval

e_0, the first pass of the algorithm computes $F_1(e_0)$. Each $F_1(e_0)$ is used to update $F_1(e)$, where e is a calendar pattern that covers e_0. In the subsequent k-th pass, for $k > 1$, the algorithm first generates $C_k(e_o)$ from $F_{k-1}(e_o))$, based on Apriori's method for candidate generation. Then the algorithm scans the transactions whose timestamps are covered by e_0, and generates $F_k(e_0)$. The $F_k(e_o)$ is used to update $F_k(e)$, where e is a calendar pattern that covers e_0.

For precise-match, the update is performed by intersecting $F_k(e_o)$ with $F_k(e)$, that is, $F_k(e) = F_k(e) \cap F_k(e_o)$. As an example, using a calendar schema $R = (week\colon \{1,2,3\}, day\colon \{1,2,\cdots,7\})$, let $e_0 = \langle 2,3 \rangle$. After this basic interval is processed in a pass k, the algorithm updates $F_k(e)$ of calendar patterns $\langle 2,* \rangle$ and $\langle *,3 \rangle$, because both calendar patterns cover $e_0 = \langle 2,3 \rangle$. Note that when $F_k(e)$ is first updated, $F_k(e) = F_k(e_o)$. Using this update method, a k-itemset is frequent in $F_k(e)$ only if it is frequent in all basic intervals covered by e.

2.1.5 Temporal Predicate Association Rules

The temporal predicate association rule was first proposed by Rainsford and Roddick (1999). The model extends the association rule model by adding to the rules a conjunction of binary temporal predicates that specify the relationships between the timestamps of transactions. The binary temporal predicates are defined using thirteen interval based relations proposed by Allen (1983) and the neighbourhood relationships defined by Freksa (1992) .

Let D be a database of transactions and $X \rightarrow Y$ be an association rule holds in D. Let $P_1 \wedge P_2 \ldots \wedge P_n$ be a conjunction of binary temporal predicates defined on items contained in either X or Y, where $n \geq 0$. A temporal predicate association rule is a rule of the form $X \rightarrow Y \wedge P_1 \wedge P_2 \ldots \wedge P_n$. In addition to the rule's confidence, a *temporal confidence* is used to determine the strength of the temporal relationships between temporal items in the rule. The rule $X \rightarrow Y \wedge P_1 \wedge P_2 \ldots \wedge P_n$ holds in a dataset D with the confidence c if and only if at least $c\%$ of transactions in D that contain X also contain Y. Similarly, each predicate P_i holds with a temporal confidence tc_{P_i} if and only if at least $tc_{P_i}\%$ of transactions in D that contain X and Y also satisfy P_i.

An example of temporal predicate association rule is as follows (Rainsford & Roddick 1999):

$$\begin{gathered}\text{policy\_c} \rightarrow \text{invest\_a, product\_b} \mid 0.87 \wedge \\ (\textit{during}(\text{invest\_a,policy\_c}) \mid 0.79) \wedge \\ (\textit{before}(\text{product\_b,invest\_a})|0.91)\end{gathered}$$

An interpretation for this rule is:

> The purchase of investment *a* and product *b* are associated with insurance policy *c* with a confidence 0.87. The investment *a* occurs during the period of policy *c* with a temporal confidence 0.79 and the purchase of product *b* occurs before investment *a* with a temporal confidence of 0.91.

The algorithm to discover temporal predicate association rules consists of four phases. The first phase of the algorithm can be performed using any algorithm for mining association rules. During this phase the temporal attributes associated with the items are not considered. In the second phase, all of possible pairings of temporal items in each generated rule are generated. For example, if the association rule is $(ab) \rightarrow (c)$ then there are three possible pairings, i.e., *ab*, *ac*, and *bc*. In the third phase, the database is scanned to determine the temporal relationships between the candidate item pairings. Finally, the aggregation of temporal relationships found in the third phase is then concatenated with the original rule to generate temporal predicate association rules.

2.2 Mining Sequential Patterns

Sequential pattern mining, which discovers frequent sequences in a sequence database, is an important data mining problem with broad applications, such as the analysis of customer purchase patterns, web access patterns, scientific experiments, disease treatments, natural disasters, and DNA sequences (Pei, Han, Mortazavi-Asl, Pinto, Chen, Dayal & Hsu 2001). Since it was first introduced by Agrawal and Srikant (1995), it is one of the active research areas in temporal data mining. As a result, a number of algorithms and techniques have been proposed for mining sequential patterns. Earlier algorithms for sequential pattern mining are Apriori-like algorithms, based on the Apriori property proposed for mining frequent itemsets (Agrawal & Srikant 1994). A series of Apriori-like algorithms have been proposed, among others, AprioriAll (Agrawal & Srikant 1995), GSP (Srikant & Agrawal 1996), and SPADE (Zaki 1998). As alternatives to the Apriori-based approach, other methods have also been proposed, such as FreeSpan (Han, Pei, Mortazavi-Asl, Chen, Dayal & Hsu 2000) and PrefixSpan (Pei et al. 2001). These new methods use a pattern-growth approach, based on the FP-growth algorithm for mining frequent itemsets (Han, Pei & Yin 2000). Various extensions of sequential pattern mining have been developed, such as mining constraint-based sequential patterns, mining closed sequential patterns, incremental and interactive mining of sequential patterns. Most extensions are solved by extending the basic algorithms for mining sequential patterns.

This section provides a review of previous studies on sequential pattern mining. Section 2.2.1 defines the problem of mining sequential patterns. Section 2.2.2 describes different approaches for mining sequential patterns and various extensions of sequential pattern mining. Sections 2.2.3 and 2.2.4 discuss the discovery of sequential patterns using Apriori-based approach with horizontal and vertical data format, respectively. Section 2.2.5 describes the pattern-growth approach with database projection, while Section 2.2.6 describes the pattern-growth approach with database indexing. Each approach is described in some detail, outlining how the basic algorithm works and identifying other contributions related to the variations and extensions of the basic algorithms.

2.2.1 Problem Definition

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of of all items. An *itemset* is a non-empty unordered collection of items. Without loss of generality, it is assumed that items in an itemset are sorted in lexicographic order. An itemset i is denoted as $(i_1 i_2 \ldots i_m)$, where $i_j \in I$ for $1 \leq j \leq m$. An itemset that contains k items is called a k-itemset.

A *sequence* $s = \langle s_1 s_2 \ldots s_m \rangle$ is an ordered list of itemsets, where each itemset is called an *element* of the sequence. An item can appear only once in a given element, but it can appear multiple times in different elements of a sequence. The number of instances of items $k = \sum_{i=1}^{m} |s_i|$ is called the *length* of the sequence. A sequence with length k is called the k-sequence. For example, $\langle (a)(b)(a) \rangle$, $\langle (ab)(a) \rangle$, and $\langle (b)(ac) \rangle$ are all 3-sequences.

A sequence $t = \langle t_1 t_2 \ldots t_m \rangle$ is a *subsequence* of $s = \langle s_1 s_2 \ldots s_n \rangle$, denoted as $t \sqsubseteq s$, if there exist integers $1 \leq j_1 < j_2 < \ldots < j_m \leq n$ such that $t_1 \subseteq s_{j_1}, t_2 \subseteq s_{j_2}, \ldots, t_m \subseteq s_{j_m}$. If t is a subsequence of s, alternatively, s is said to *contain* t. As an example, a sequence $\langle (a)(bc)(d) \rangle$ is a subsequence of $\langle (a)(abc)(ac)(d) \rangle$. A sequence is said to be *maximal* in a set of sequences, if it is not contained in other sequences in the set.

A database D is a set of transactions (*CID, TID, ITEMS*), where *CID* is a customer-id, *TID* is a transaction-id (based on the transaction time), and *ITEMS* is a list of items in the transaction. For a given customer-id, there are no transactions with the same transaction time, which means that there are no transactions with the same transaction-id. All the transactions with the same CID can be regarded as a sequence of itemsets ordered by increasing TID. Each such sequence is called a *data sequence*. Therefore, an analogous representation of the database D is a set of data sequences, one sequence per customer, each representing an order of transactions a customer has conducted.

Example 2.1 Let D be a database given in Figure 2.2, which contains eight items (a to h), four customers, and ten transactions (adopted from Zaki (2001)).

CID	TID	ITEMS
1	10	c, d
1	15	a, b, c
1	20	a, b, f
1	25	a, c, d, f
2	15	a, b, f
2	20	e
3	10	a, b, f
4	10	d, g, h
4	20	b, f
4	25	a, g, h

(a)

CID	Data sequence
1	⟨(cd) (abc) (abf) (acdf)⟩
2	⟨(abf) (e)⟩
3	⟨(abf)⟩
4	⟨(dgh) (bf) (agh)⟩

(b)

Figure 2.2: A database shown as a set of transactions and a set of sequences

Figure 2.2(a) shows the database as a set of transactions, sorted on TID within each CID. Figure 2.2(b) shows the database represented as a set of data sequences. This database is used as a running example throughout the discussion of mining sequential patterns.

The *support* of a sequence s in the database D, denoted by *sup(s)*, is defined as the fraction of data sequences in the database that contain s. Given a minimum support threshold *minsup*, a sequence is called *frequent* in D if its support is greater than or equal to *minsup*. The task of mining sequential patterns is to find all frequent sequences (sequential patterns) in the database D.

As an example, given the database D in Example 2.1 and a minimum support threshold $minsup = 0.4$, a sequence $\langle(bf)(a)\rangle$ has the support of 0.5, since it is contained in 2 of 4 data sequences (of customer 1 and 4). Therefore, the sequence is frequent in D. A complete list of frequent sequences is shown in Figure 2.3.

Length	Sequential patterns (*minsup* = 0.4)
1	⟨(a)⟩, ⟨(b)⟩, ⟨(d)⟩, ⟨(f)⟩
2	⟨(ab)⟩, ⟨(af)⟩, ⟨(bf)⟩, ⟨(b)(a)⟩, ⟨(d)(a)⟩, ⟨(d)(b)⟩, ⟨(d)(f)⟩, ⟨(f)(a)⟩
3	⟨(abf)⟩, ⟨(bf)(a)⟩, ⟨(d)(b)(a)⟩, ⟨(d)(bf)⟩, ⟨(d)(f)(a)⟩
4	⟨(d)(bf)(a)⟩

Figure 2.3: A set of sequential patterns

2.2.2 Sequential Pattern Mining Algorithms and Extensions

Sequential Pattern Mining Algorithms

Discovering frequent sequences is much more complex than discovering frequent itemsets. If m is the total number of distinct items in the input data, the maximum number of sequences having k items is $O(m^k 2^{k-1})$. In contrast, there are only $\binom{m}{k}$ possible itemsets of size k in association rule mining (Joshi, Karypis & Kumar 2001). As a result, a large number of studies in sequential pattern mining have focused upon improving the efficiency of the algorithms.

In the data mining literature, algorithms for mining sequential patterns are classified into two broad categories: *Apriori-based* and *pattern-growth* approaches. The Apriori-based approach uses a candidate generation-and-test paradigm initially proposed for mining frequent itemsets (Agrawal & Srikant 1995). In order to reduce the search space, the algorithms in this category generally employ an *Apriori principle*, which states that all subsequences of a frequent sequence must be frequent. Based on the format of their input databases, the Apriori-based approach can be further separated into two categories: Apriori-based with *horizontal data format* and Apriori-based with *vertical data format*. The difference between these two data formats is described in Section 2.2.4.

Meanwhile, the pattern-growth approach adopts a pattern-growth principle of mining frequent itemsets developed in the FP-growth algorithm (Han, Pei & Yin 2000). This approach does not require candidate generation. The database is recursively projected into a set of smaller projected databases, and sequential patterns are grown in each projected database by exploring only local frequent fragments (Pei et al. 2001). Another approach for growing the patterns is by indexing the database in memory (Lin & Lee 2002). This approach does not require candidate generation and database projection. It discovers sequential patterns by a recursive *find-then-index* technique. This technique recursively finds the items which constitute a frequent sequence and constructs an index set which indicates the set of data sequences for further exploration.

Extensions of Sequential Pattern Mining

The sequential pattern mining problem has been extended in various ways, including mining constraint-based sequential patterns, mining closed sequential patterns, mining top-k closed sequential patterns, incremental and interactive sequential pattern mining. Each extension is briefly described below.

In many applications, sequential pattern mining can produce a large number of sequential patterns. One way to address this issue is by allowing the user to interactively add certain syntactic constraints on the mined sequences. For example, the user may be interested in only those sequences that occur close together, those that occur far apart, those that occur within a specified time frame, those that contain specific items or those that predict a given attribute (Zaki 2000). The mining process which incorporates user-specified constraints to reduce search space and derive only the user-interested pattern is called *constraint-based mining*. Srikant and Agrawal (1996) were the first to consider mining sequential patterns with time constraints (i.e., minimum and maximum gap between successive sequence elements), sliding time windows, and user-defined taxonomy. Garofalakis *et al.* (1999) proposed regular expressions as constraints for sequential pattern mining and developed a family of SPIRIT algorithms.

Another method is to search not just for frequent sequences but for sequences that are *closed* as well. Pasquier *et al.* (1999) have introduced the idea of mining

Category		Algorithm	Author	Year	Type of sequential pattern mining	
					Basic	Extension
Apriori-based	Horizontal data format	AprioriAll	Agrawal & Srikant	1995	√	
		GSP	Srikant & Agrawal	1996		Constraint-based mining
		PSP	Masseglia *et al.*	1998	√	
		SPIRIT	Garofalakis *et al.*	1999		Constraint-based mining
		MFS	Zhang *et al.*	2001	√	
		ISE	Masseglia *et al.*	2003		Incremental mining
		KISP	Lin & Lee	2003		Interactive mining
		SPaRSe	Antunes & Oliveira	2004		Constraint-based mining
		MFS+	Kao *et al.*	2005		Incremental mining
	Vertical format with id-list	SPADE	Zaki	2001	√	
		ISM	Parthasarathy *et al.*	1999		Incremental mining
		cSPADE	Zaki	2000		Constraint-based mining
		GOSPADE	Leleu *et al.*	2003	√	
		GOSpec	Leleu *et al.*	2003		Constraint-based mining
	Vertical format in binary	SPAM	Ayres *et al.*	2002	√	
		LAPIN-SPAM	Yang & Kitsuregawa	2005	√	
Pattern growth with database projection		FreeSpan	Han *et al.*	2000	√	
		PrefixSpan	Pei *et al.*	2001	√	
		DELISP	Lin *et al.*	2002		Constraint-based mining
		GenPrefixSpan	Antunes & Oliveira	2003		Constraint-based mining
		CloSpan	Yan *et al.*	2003		Closed sequential pattern mining
		TSP	Tzvetkov *et at.*	2003		Top-k closed sequential pattern mining
		BIDE	Wang & Han	2004		Closed sequential pattern mining
Pattern growth with database indexing		MEMISP	Lin & Lee	2002	√	

Figure 2.4: Classification of sequential pattern mining algorithms

frequent closed itemsets. Recently, methods for mining closed sequential patterns have been proposed (Yan, Han & Afshar 2003, Wang & Han 2004). A sequential pattern is said to be closed if it is not properly contained in any other sequence which has the same support. Formally, given a set F of all sequential patterns, a set CS of closed sequential patterns is defined as

$$CS = \{\alpha|\ \alpha \in F \text{ and } \ \nexists\beta \in F \text{ such that } \alpha \sqsubseteq \beta \text{ and } sup(\alpha) = sup(\beta)\}$$

Even though mining closed sequential patterns may reduce the number of generated sequential patterns, setting a minimum support threshold is not trivial. A too small value may result in the generation of a large number of patterns, while a too big one may lead to no patterns being found. One solution is to mine *top-k frequent closed* patterns (Han, Wang, Lu & Tzvetkov 2002). Tzvetkov *et al.* (2003) have developed the TSP algorithm to discover top-k closed sequential patterns. The task of mining top-k closed sequential patterns is to find top-k closed sequential patterns of minimum length min_ℓ. Here, k is the number of closed sequential patterns to be mined, top-k refers to the k most frequent sequences, and min_ℓ is the minimum length of the closed sequential patterns.

In other situations the database may not be static, but it changes over time as new data sequences are added to or deleted from the database. As an example, every visit to a web site will add a new log to the site's log database. Also, some out-of-date logs (say those that are more than five-years old) may need to be deleted. Since an operational database changes continuously, the set of frequent sequences has also to be updated to stay valid. One simple strategy is to re-run the mining algorithm on the updated database. However, this strategy fails to take advantage of the valuable information obtained from a previous mining exercise, which is particularly useful if the updated database and the old one share a significant portion of common sequences. Therefore, there is a need for methods that are able to efficiently maintain valid mined information due to database updates. This problem is known as the *incremental sequential pattern* *mining* problem (Parthasarathy, Zaki, Ogihara & Dwarkadas 1999, Masseglia, Poncelet & Teisseire 2003, Kao, Zhang, Yip, Cheung & Fayyad 2005).

The list of algorithms discussed in this section is summarized in Figure 2.4. The figure shows the classes of algorithms for solving a basic sequential pattern mining problem and its extensions. Note that this classification is neither unique nor exhaustive, our objective is to facilitate an easy discussion of the numerous techniques in the field.

2.2.3 Apriori-Based using Horizontal Data Format

When introducing the problem of mining sequential patterns in 1995, Agrawal and Srikant presented three algorithms, namely, *AprioriAll*, *AprioriSome*, and *DynamicSome* (Agrawal & Srikant 1995). AprioriAll finds all sequential patterns, while AprioriSome and DynamicSome find only maximal sequential patterns. However, many applications require all sequential patterns, and finding maximal sequential patterns from a set of all sequential patterns is straightforward. Therefore, AprioriAll usually becomes the preferable algorithm.

AprioriAll is based on the Apriori algorithm for mining frequent itemsets (Agrawal & Srikant 1994). This algorithm divides the problem of finding sequential patterns into three phases. It first finds all frequent itemsets in the database, then transforms the database with each transaction replaced by the set of all frequent itemsets contained in the transaction, and finally finds all sequential patterns.

The problem with AprioriAll is the cost of transforming the database. It is computationally expensive to do the data transformation on-the-fly during each pass while finding sequential patterns. Alternatively, to transform the database once and store the transformed database will be impracticable for large databases. Therefore, the same authors propose the GSP (Generalized Sequential Patterns) algorithm that is faster than AprioriAll (Srikant & Agrawal 1996) and does not require database transformation. Moreover, GSP not only solves the basic sequential mining problem (Section 2.2.1), but it also generalizes the problem by considering time constraints, sliding windows, and taxonomies on the items.

Algorithm 2.3 outlines the discovery of sequential patterns using GSP, which consists of multiple passes. In the first pass, it scans the database to find all of the frequent items, which form the set F_1 of frequent 1-sequences. In the k-th pass, $k > 1$, the algorithm uses a set F_{k-1} of frequent $(k-1)$-sequences to generate a set C_k of candidate k-sequences. Then, the database is scanned to count the support for each candidate sequence. All of the candidates whose support in the database exceeds a given minimum support threshold form the set F_k of frequent k-sequences. This set is then used to generate a set of candidate sequence in the next pass. The algorithm terminates when no new sequential pattern is found in the pass, or no candidate sequence can be generated. The method for candidate generation is described in more detail in the following.

```
1: $F_1 = \{\text{frequent 1-sequences}\}$
2: **for** $(k = 2; F_{k-1} \neq \emptyset; k = k + 1)$ **do**
3:    Generate $C_k$ from $F_{k-1}$;
4:    **for** all customer-sequences $s$ in the database **do**
5:       Increment support count of all $c \in C_k$ contained in $s$
6:    **end for**
7:    $F_k = \{c \in C_k | sup(c) \geq minsup\}$;
8: **end for**
9: Return $F = \bigcup_k F_k$;
```
Algorithm 2.3: Pseudo code of the GSP Algorithm

GSP generates the set C_k from F_{k-1} in two steps. In the first step, *join step*, GSP joins F_{k-1} with F_{k-1}. For any pair of sequences α and β in F_{k-1}, α joins with β if discarding the first item of α and the last item of β results in identical sequences. The candidate sequence generated by joining α and β is the sequence α extended with the last item of β. The new item is added as a new element of α if it is a separate element in β, otherwise it is added to the last element of α. For example, consider three frequent 2-sequences in Figure 2.3: $\langle(ab)\rangle$, $\langle(bf)\rangle$, and $\langle(f)(a)\rangle$. The first and the second sequences can be joined to create a candidate 3-sequence $\langle(abf)\rangle$, because dropping items a and f from the first and the second sequences, respectively, results in an identical subsequence $\langle(b)\rangle$. Similarly, the join of the second and the third sequences results in a candidate sequence $\langle(bf)(a)\rangle$. Next, in the *prune step*, the algorithm deletes all candidate sequences $c \in C_k$ such that some $(k-1)$-subsequence of c is not in F_{k-1}. As an example, the join of $\langle(ab)\rangle$ and $\langle(b)(a)\rangle$ results in a candidate sequence $\langle(ab)(a)\rangle$. The prune step deletes $\langle(ab)(a)\rangle$ because it has a subsequence $\langle(a)(a)\rangle$ which is not in F_2 (Figure 2.3). According to the Apriori principle (mentioned in Section 2.2.2), the sequence $\langle(ab)(a)\rangle$ will not be frequent because it has a subsequence that is not frequent.

Even though the use of the Apriori pruning in GSP can reduce the search space, it is only effective when $k > 2$. For $k = 2$, for example, if there are 1000 frequent 1-sequences, GSP will generate $1000 \times 1000 + \frac{1000 \times 999}{2} = 1,499,500$ candidate sequences. This requires a significant increase in resources. Besides that, each iteration of GSP only discovers frequent sequences of the same length; that is, a set of frequent k-sequences is discovered at the k-th iteration of the algorithm. Consequently, the number of iterations (hence database scans) is dependent on the length of the maximum length of frequent sequences. Multiple scans of databases could be costly if the databases contain long sequential patterns. As a result, other algorithms later developed to improve the performance of GSP are aimed to solve these problems, such as by reducing the number of database scans, or by getting rid of the need to generate candidate patterns.

The use of a more efficient data structure during candidate generation and support counting could also improve the performance of GSP. This approach is taken by the PSP algorithm (Masseglia, Cathala & Poncelet 1998). PSP still follows the approach of candidate generation followed by support counting. However, PSP uses *prefix-trees* as its internal data structures, instead of *hash-trees* used in GSP. The benefits of using the prefix-tree are that it requires less storage space and improves efficiency during candidate generation and support counting processes.

MFS (Mining Frequent Sequences) is a two-stage GSP-based algorithm for mining sequential patterns (Zhang, Kao, Yip & Cheung 2001). In the first stage, a sample of the database is mined to get the rough estimation of frequent sequences. Then, based on this estimation the database is scanned to check and refine candidate sequences until no more frequent sequences can be found. The difference between MFS and GSP is the function of candidate generation. While GSP generates C_k from F_{k-1}, MFS takes a set of frequent sequences of various length to generate a set of candidate sequences of various length. The paper claims that MFS discovers the same set of frequent sequences as does GSP, but no proof of this is given in the paper. The MFS+ algorithm is an extension of MFS for incremental mining of sequential patterns (Kao et al. 2005).

A universal formulation of sequential patterns was introduced by Joshi *et al.* (2001). Different types of constraints such as structure, time, and item to be integrated into the universal system are discussed; also corresponding counting methods of sequential patterns and how to set thresholds are explained. A GSP based algorithm is introduced with different count methods to explain how to use the universal formulation of sequential patterns.

ISE (Incremental Sequence Extraction) is a GSP based algorithm for incremental mining of sequential patterns (Masseglia et al. 2003). In order to reduce the cost of finding new sequential patterns in the updated databases, the algorithm utilises information collected during an earlier mining process. For example, during candidate generation, the algorithm avoids generating candidate sequences that have already been found to be frequent.

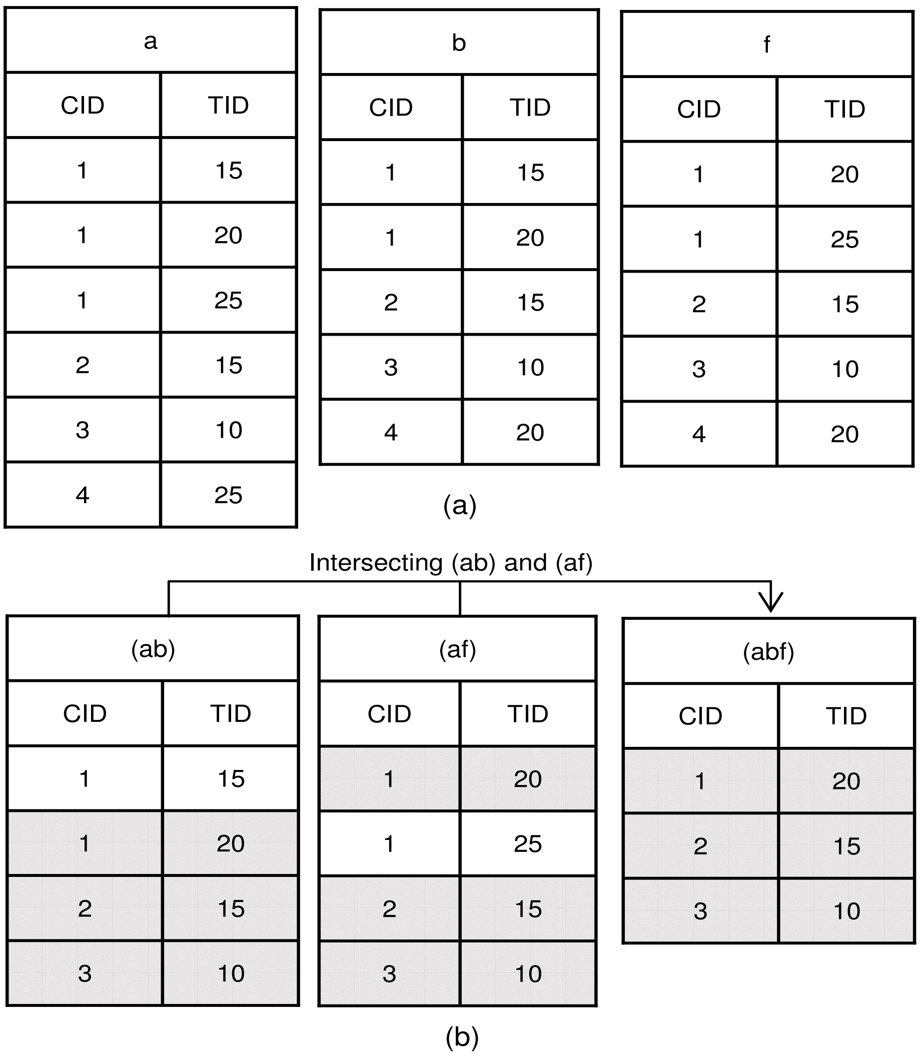
Lin and Lee (2003) introduced the KISP (Knowledge Base Assisted Incremental Sequential Pattern) algorithm for interactive discovery of sequential patterns. KISP extends GSP with a knowledge base (KB), so that all queries about sequential patterns of various minimum support thresholds can be obtained from the KB. When the desired patterns cannot be obtained from the KB, KISP mines the database for new patterns by employing two optimization methods: *direct* *new-candidate generation* and *concurrent support counting.*

2.2.4 Apriori-Based using Vertical Data Format

All of the previously discussed algorithms work on a horizontal data format illustrated in Figure 2.2. In the horizontal data format, the database contains a list of customers (CID), each with its own list of transaction-ids (TIDs) and sets of items. The SPADE algorithm (Zaki 1998), on the other hand, uses a *vertical* format consisting of items' *id-list.* The id-list of an item is a list of (CID, TID) pairs indicating the transaction-id (i.e., transaction-time) of the item in the data sequence. Figure 2.5(a) shows the vertical data format of part of an example database. As shown in the figure, the id-list for the item a consists of tuples $\{(1, 15), (1, 20), (1, 25), (2, 15), (3, 10), (4, 25)\}$. This is because item a appears in customer-id 1, transaction-ids 15, 20, 25, and so on (Figure 2.2)(a)).

As was mentioned in the previous subsection, one of the shortcomings of GSP is that it requires multiple database scans which depend on the length of the longest frequent sequence in the databases. As a result, if the database is large and contains long frequent sequences, GSP incurs a high I/O cost. Using the vertical data format, SPADE can improve the performance of GSP by reducing the number of database scans. Given the per item id-lists, SPADE iteratively determines the support of any k-sequence by intersecting the id-lists of its two generating $(k-1)$-sequences. By checking on the number of distinct customer-ids of the resulting id-lists, SPADE can determine whether the new sequence is frequent or not. As a result, SPADE can complete the mining process in three passes over the database, as outlined below.

SPADE first generates a set F_1 of frequent 1-sequences. Given, the vertical database format this can be done in a single database scan. For each item, the algorithm reads the item's id-list from the database. While scanning the id-list, it increments the item's support count for each new customer-id encountered. SPADE then generates a set F_2 of frequent 2-sequences. Computing F_2 using



a

CID	TID
1	15
1	20
1	25
2	15
3	10
4	25

b

CID	TID
1	15
1	20
2	15
3	10
4	20

f

CID	TID
1	20
1	25
2	15
3	10
4	20

(a)

Intersecting (ab) and (af)

(ab)

CID	TID
1	15
1	20
2	15
3	10

(af)

CID	TID
1	20
1	25
2	15
3	10

(abf)

CID	TID
1	20
2	15
3	10

(b)

Figure 2.5: Vertical data format used in SPADE

the vertical database format is expensive, because the algorithm has to read the id-list of each frequent item in F_1 and intersect it with the id-list of other items in F_1. If there are m frequent items in F_1, this approach requires m database scans, whereas in the horizontal format this can be done in a single scan. To overcome this problem, SPADE optimises the calculation of F_2 by inverting the vertical format to the horizontal format, and then using the new format to compute F_2.

The next step is to find a set F_k of frequent k-sequences, for $k \geq 3$. In order to generate a candidate k-sequence α, SPADE uses two frequent $(k-1)$-sequences α_1 and α_2 that share a common prefix. There are several different operations for generating candidates, depending on the form of α_1 and α_2. Let p be a common prefix of α_1 and α_2, and x and y are items. The first operation, if $\alpha_1 = \langle(px)\rangle$ and $\alpha_2 = \langle(py)\rangle$, then $\alpha = \langle(pxy)\rangle$. The second, if $\alpha_1 = \langle(px)\rangle$ and $\alpha_2 = \langle(p)(y)\rangle$, then $\alpha = \langle(px)(y)\rangle$. The third, if $\alpha_1 = \langle(p)(x)\rangle$ and $\alpha_2 = \langle(p)(y)\rangle$, then there are three possible candidates $\alpha = \langle(p)(x)(y)\rangle$, $\alpha = \langle(p)(y)(x)\rangle$, and $\alpha = \langle(p)(xy)\rangle$. As an example, two sequences $\langle(ab)\rangle$ and $\langle(af)\rangle$ have a common prefix $\langle(a)\rangle$ and their join results in a candidate $\langle(abf)\rangle$.

The support of the candidate sequence α is not counted by scanning the database. Instead, SPADE intersects (joins) the id-lists of α_1 and α_2 and counts the support of the resulting id-list. Figure 2.5(b) shows the intersection of the id-lists of $\langle(ab)\rangle$ and $\langle(af)\rangle$. As shown in the figure, the resulting sequence $\langle(abf)\rangle$ has the support of $\frac{3}{4}$, where 3 is the number of distinct customer-ids in its id-list. All of the candidates whose support exceeds a given minimum support threshold become the set of the newly found sequential patterns.

In order to reduce the memory consumption and increase its efficiency, SPADE employs various optimizations, in particular, a notion of equivalence class of sequential patterns, dedicated breadth-first and depth-first search strategies. For more detail, refer to Zaki (1998, 2000).

Based on the SPADE algorithm, Parthasarathy *et al.* (1999) propose the ISM (Incremental Sequence Mining) algorithm for incremental and interactive mining of sequential patterns. All queries are performed on a pre-processed in-memory data structure called ISL (Increment Sequence Lattice). The ISL includes all the frequent sequences and all the sequence in the negative border. The negative border is the collection of all sequences that are not frequent but both of their generating subsequences are frequent.

The cSPADE (Zaki 2000) algorithm extends SPADE by modifying the id-list in SPADE to incorporate minimum gap, maximum gap, time window, and some other constraints. The time window refers to the window of occurrence of the whole sequence, not the sliding window used in GSP (Srikant & Agrawal 1996).

GOSPADE extends SPADE to incorporate the *generalized occurrences* (Leleu, Rigotti, Boulicaut & Euvrard 2003*b*), which can be used to reduce the size of the occurrence lists (id-lists) by representing several occurrences with a single more general one. As described above, an id-list in SPADE stores one line per occurrence, that is, three lines for occurrences of item a in customer-id 1, one line for customer-id 2, and so on (Figure 2.5(a)). Using the concept of generalized occurrence, the three consecutive occurrences of item a in customer-id 1 can be represented by only one generalized occurrence of the form $(1, [15, 20, 25])$. This method is especially useful when the databases contain consecutive repetitions of items. Later, the same authors develop the GOSpec algorithm by extending GOSPADE for mining sequential patterns involving the maximum and minimum gap time constraints, and the time window constraint (Leleu, Rigotti, Boulicaut & Euvrard 2003*a*).

Instead of using the id-list, the SPAM (Sequential PAttern Mining) algorithm uses a vertical bitmap representation of the database (Ayres, Flannick, Gehrke & Yiu 2002). A vertical bitmap is constructed for each item in the database, and each bitmap has a bit corresponding to each transaction in the database. If an item appears in a transaction, the bit corresponding to the transaction of the bitmap for the item is set to '1'; otherwise, the bit is set to '0'. Consider the database in Example 2.1. The vertical bitmap representation of this database is presented in Figure 2.6. Each vertical bitmap for the item is partitioned into four *sections*, and each corresponds to a data sequence. The first transaction of the first customer contains items c and d, so the bit that corresponds to this transaction in each of the bitmaps c and d is set to '1', and other bits are set to '0'.

As an Apriori-based algorithm, SPAM applies a candidate generation-and-test principle. To generate the candidate sequences, it uses two operations: *S-step* and *I-step*. Given a sequence s and an item x, the S-step generates the candidate sequence by appending a new element (x) to the end of the sequence s. On the other hand, the I-step generates the candidate sequence by adding the item x

CID	TID	a	b	c	d	e	f	g	h
1	10	0	0	1	1	0	0	0	0
1	15	1	1	1	0	0	0	0	0
1	20	1	1	0	0	0	1	0	0
1	25	1	0	1	1	0	1	0	0
2	15	1	1	0	0	0	1	0	0
2	20	0	0	0	0	1	0	0	0
-	-	0	0	0	0	0	0	0	0
-	-	0	0	0	0	0	0	0	0
3	10	1	1	0	0	0	1	0	0
-	-	0	0	0	0	0	0	0	0
-	-	0	0	0	0	0	0	0	0
-	-	0	0	0	0	0	0	0	0
4	10	0	0	0	1	0	0	1	1
4	20	0	1	0	0	0	1	0	0
4	25	1	0	0	0	0	0	1	1
-	-	0	0	0	0	0	0	0	0

Figure 2.6: Vertical bitmap representation of the example database

to the last element of s. For example, given a sequence $\langle(af)\rangle$ and an item b, the S-step generates a candidate $\langle(af)(b)\rangle$, while the I-step generates $\langle(abf)\rangle$. SPAM counts the support of the candidate sequence by looking at the bitmap of the candidate sequence. The bitmap of the candidate sequence is the result of AND-ing the bitmaps of a sequence s and an item x.

LAPIN-SPAM (LAst Position INduction Sequential PAttern Mining) improves the performance of SPAM by using a table called ITEM_IS_EXIST_TABLE (Yang & Kitsuregawa 2005). The table contains bit vectors representing the existence of candidate sequences in each data sequence. Using information in the table, the algorithm can avoid performing AND operations required by SPAM.

2.2.5 Pattern-Growth using Database Projection

The first proposed pattern-growth algorithm for sequential pattern mining is the FreeSpan (Frequent Pattern-Projected Sequential Pattern Pattern Mining) algorithm (Han, Pei, Mortazavi-Asl, Chen, Dayal & Hsu 2000). Its basic idea is to use frequent items to recursively project sequence databases into a set of smaller projected databases to confine the search and growth of subsequences. Later, Pei *et al.* (2001) propose the PrefixSpan (Prefix-projected Sequential Pattern Mining) algorithm, which utilises prefix-projection to mine the complete set of sequential patterns. Both methods create projected databases but they differ in the criteria for database projection. FreeSpan creates projected databases based on the current set of frequent sequences, while PrefixSpan does so based on frequent prefixes only. The PrefixSpan algorithm is described in more detail below by first defining notations used in the discussion.

Given two sequences $\alpha = \langle e_1 e_2 \dots e_n \rangle$, where each e_i corresponds to a frequent element in D and $\beta = \langle e'_1 e'_2 \dots e'_m \rangle$, where $m \le n$, β is called a *prefix* of α if and only if (1) $e'_i = e_i$ for $i \le m-1$; (2) $e'_m \subseteq e_m$; and (3) all frequent items in $(e_m - e'_m)$ are alphabetically ordered after those in e'_m. Sequence $\gamma = \langle e''_m e_{m+1} \cdots e_n \rangle$ is called the *suffix* of α w.r.t prefix β, where $e''_m = (e_m - e'_m)$. For example, given a sequence $s = \langle (a)(bcd)(e)(f) \rangle$, sequences $\langle (a) \rangle$, $\langle (a)(b) \rangle$, and $\langle (a)(bc) \rangle$ are prefixes of s, while $\langle (a)(c) \rangle$ or $\langle (a)(cd) \rangle$ is not. Also, $\langle (bcd)(e)(f) \rangle$ is a suffix of s w.r.t the prefix $\langle (a) \rangle$, and $\langle (cd)(e)(f) \rangle$ is a suffix of s w.r.t the prefix $\langle (a)(b) \rangle$.

Let α be a sequential pattern in the database D. The *α-projected database*, denoted as D_α, is the collection suffixes of sequences in D w.r.t. prefix α.

PrefixSpan discovers sequential patterns in three steps. As an illustration, each step is described using the database in Example 2.1.

Step 1: Find frequent 1-sequences. The first step of PrefixSpan is to scan the database D once to find all frequent items. Each of these frequent items is a frequent 1-sequence. From the running example, the set of frequent 1-sequences contains: $\langle (a) \rangle$, $\langle (b) \rangle$, $\langle (d) \rangle$, and $\langle (f) \rangle$.

Step 2: Divide search space into smaller subspaces. The set of all frequent sequences can be divided into several groups such that the sequences within a group share the same prefix. For example, from the set of frequent 1-sequences

Prefix (α)	α–projected database
⟨(a)⟩	⟨(_b)(abf)(adf)⟩, ⟨(_bf)⟩, ⟨(_bf)⟩, ⟨ ⟩
⟨(b)⟩	⟨(abf)(adf)⟩, ⟨(f)⟩, ⟨(f)⟩, ⟨(f)(a)⟩
⟨(d)⟩	⟨(ab)(abf)(df)⟩, ⟨ ⟩, ⟨ ⟩, ⟨(bf)(a)⟩
⟨(f)⟩	⟨(adf)⟩, ⟨ ⟩, ⟨ ⟩, ⟨(a)⟩

Figure 2.7: Database projected on frequent 1-sequences

found in the first step, the set of all frequent sequences can be partitioned into four groups according to the four prefixes $\langle(a)\rangle$, $\langle(b)\rangle$, $\langle(d)\rangle$, and $\langle(f)\rangle$.

Step 3: Find frequent sequences in each subspace. Each group of sequential patterns can be mined by constructing corresponding projected databases and by mining each recursively. As an example, to find the frequent sequences with prefix $\langle(a)\rangle$, the database D is projected to get an intermediate database $D_{\langle(a)\rangle}$. For every data sequence s in D, $D_{\langle(a)\rangle}$ contains the suffix of s w.r.t. $\langle(a)\rangle$. Figure 2.7 shows the projected database projected on each of frequent 1-sequences. In the figure, an underscore '_' preceding item b indicates that the last element in the prefix, which is a, together with b, forms one element. After obtaining $D_{\langle(a)\rangle}$, the algorithm scans $D_{\langle(a)\rangle}$ to find all frequent 2-sequences having prefix $\langle(a)\rangle$. The scanning of $D_{\langle(a)\rangle}$ finds $\langle(\_b)\rangle$ and $\langle(\_f)\rangle$, each with the support count of 3. Thus, all frequent 2-sequences prefixed with $\langle(a)\rangle$ are found, that is, $\langle(ab)\rangle$ and $\langle(af)\rangle$. Then recursively, the database $D_{\langle(a)\rangle}$ is projected w.r.t. $\langle(ab)\rangle$ and $\langle(af)\rangle$ to obtain $D_{\langle(ab)\rangle}$ and $D_{\langle(af)\rangle}$, respectively. Each of these projected databases is recursively mined to obtain frequent sequences with the corresponding prefix. The recursive process continues until all frequent sequences with prefix $\langle(a)\rangle$ are found.

By projecting databases and counting items' supports, PrefixSpan only counts the supports of sequences that actually occur in the database. In contrast, a candidate sequence generated by GSP may not appear in the database at all. The time for generating such candidate sequence and checking whether such a candidate is a subsequence of database sequences is wasted. This factor contributes to the efficiency of PrefixSpan over GSP.

However, the cost of disk I/O might be high due to the creation and processing of the projected databases. To further improve its performance, two optimizations for minimizing disk projections are proposed (Pei et al. 2001). The *bi-level projection* technique, dealing with huge database, scans each sequence twice in the (projected) database so that fewer and smaller projections are generated. The *pseudo-projection* technique, avoiding physical projections, keeps the sequence-suffix by a pointer-offset for each sequence in the projection. However, the maximum mining performance can be achieved only when the database size is reduced to the size fit in the main memory, that is, by employing pseudo-projection after using bi-level optimization.

The DELISP (DELImited Sequential Pattern) algorithm (Lin, Lee & Wang 2002) extends PrefixSpan to discover the generalized sequential patterns, constrained with minimum gap, maximum gap and sliding window. The bounded projection technique eliminates invalid subsequence projections caused by unqualified maximum or minimum gaps. The window projection technique reduces redundant projections for adjacent elements satisfying the sliding window constraint. The delimited growth technique grows only patterns satisfying constraints.

Antunes and Oliveira (2003) develop an algorithm called GenPrefixSpan, which extends PrefixSpan to discover sequential patterns with gap constraints. The same authors also propose the SPaRSe (Sequential Pattern Mining with Restricted Search) algorithm for mining sequential patterns (Antunes & Oliveira 2004). The algorithm combines the candidate-and-test principle of GSP with the restriction of the search space obtained from the use of projected databases.

2.2.6 Pattern-Growth using Database Indexing

As an alternative to growing the frequent sequences with database projection, Lin and Lee (2002) propose the MEMISP (MEMory Indexing for Sequential Pattern Mining) algorithm which grows the frequent sequences with memory-indexing approach. In this approach there is no candidate generation and no database projection. Instead, MEMISP reads the database into memory, then through index advancement within an index set, MEMISP discovers patterns by a recursive *finding-then-indexing* technique. MEMISP only requires one database scan if the database fits in memory.

The following notations are needed to describe the MEMISP algorithm. Given a frequent sequence ρ and a frequent item x in the database D, ρ' is a *type-1* sequential pattern if it can be formed by appending item x as a new element of ρ, and is a *type-2* sequential pattern by extending the last element of ρ with x. The frequent item x is called the *stem* of the sequential pattern ρ' and ρ is called the *prefix pattern* (*P-pat* for short) of ρ'.

For example, given a frequent sequence $\langle(a)\rangle$ and a frequent item b, $\langle(a)(b)\rangle$ is a type-1 sequential pattern formed by appending b as a new element to $\langle(a)\rangle$, and $\langle(ab)\rangle$ is a type-2 sequential pattern formed by extending $\langle(a)\rangle$ with b. The $\langle(a)\rangle$ is the *P-pat* and the item b is the stem of both $\langle(a)(b)\rangle$ and $\langle(ab)\rangle$. The null sequence, denoted by $\langle\ \rangle$, is the *P-pat* of any frequent 1-sequence.

Assuming the database fits in the memory, MEMISP discovers all sequential patterns in three steps, as follows.

Step 1: Find all frequent items. In this step, the algorithm reads the database D into memory. While reading each sequence from the database, the algorithm computes the support count of every item, then finds the set of all frequent items. Using the running example database, the algorithm finds frequent items a, b, d, and f.

Similar to PrefixSpan, the set of all frequent sequences can be partitioned into four groups according to the four prefixes $\langle(a)\rangle$, $\langle(b)\rangle$, $\langle(d)\rangle$, and $\langle(f)\rangle$. Each

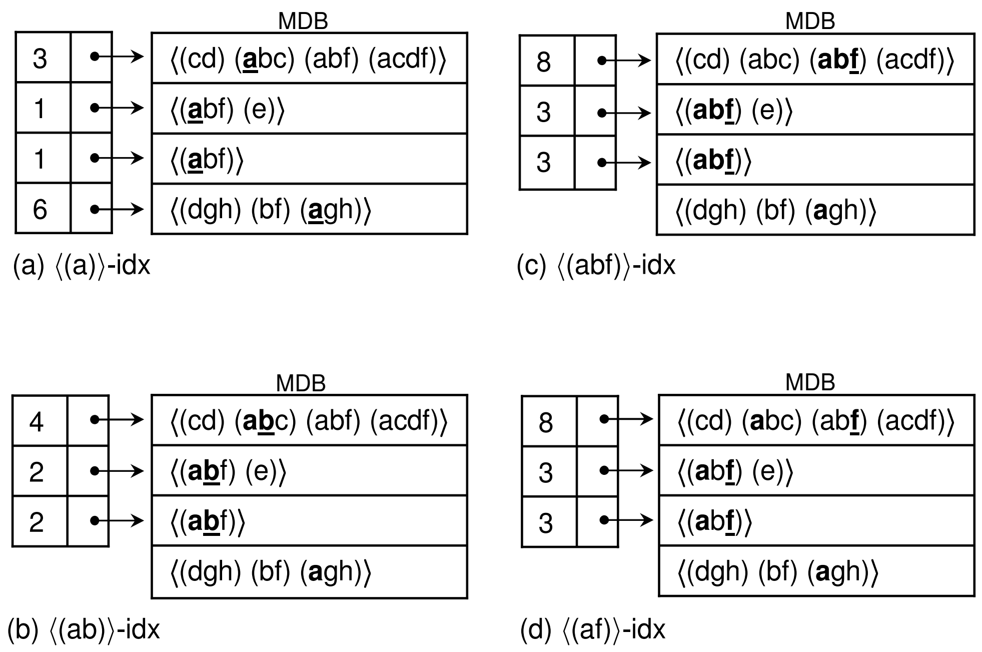


Figure 2.8: Example of index sets

group of frequent sequences can be mined by constructing corresponding index sets (step 2) and by mining each index set recursively (step 3).

Step 2: Constructing the index set. Given a *P-pat* and a stem x, this step constructs the index set ρ-*idx*, where is ρ is a sequential pattern formed by combining current *P-pat* and a stem x. An index set ρ-*idx* contains a set of the index's elements; each contains a pair *(ptr, pos)*, where *ptr* is a pointer to the data sequence $d \in D$, and *pos* is the first occurring position of x in the data sequence d (with respect to *P-pat*). An index element *(ptr, pos)* for a data sequence d is created only if d supports ρ.

Step 3: Mining the index set. Given the index set ρ-*idx*, the algorithm uses the index set to find stems w.r.t. *P-pat* $= \rho$. Any item appearing after the *pos* position in the data sequence d pointed by *ptr* of the entry (*ptr, pos*) in ρ-*idx* could be a potential stem. Thus, for every d existing in ρ-*idx*, the algorithm increases the support count of such items by one. The algorithm then determines the set of all stems having minimum support to form sequential patterns.

The following example illustrates the discovery of all frequent sequences with prefix $\langle (a) \rangle$ using steps 2 and 3. Using step 2, the algorithm constructs the index set $\langle (a) \rangle$-*idx* from a stem $x = a$ and *P-pat* $= \langle\ \rangle$. This index set is shown in Figure 2.8. In the figure, MDB denotes the in-memory database. In the index set $\langle (a) \rangle$-*idx*, the first element points to the first data sequence since the sequence supports $\langle (a) \rangle$. The value of *pos* in the first element is 3 because stem a is found at position 3 w.r.t. *P-pat* $= \langle\ \rangle$, indicated by the underline.

Then, using step 3, the algorithm mines $\langle (a) \rangle$-*idx* to find all frequent 2-sequences whose *P-pat* is $\langle (a) \rangle$. Let d_k denotes the data sequence of a customer-id k. The value of *pos* of an index element pointing to d_1 is 3, so the algorithm only needs to consider items occurring after position 3 in d_1. The support count of potential stems a, b, d, and f (for potential type-1 patterns) is increased by one. The support count of potential stems b, d, and f for potential type-2 patterns is also increased by one. The support counting continuous for items occurring after positions 1, 1, and 6 in d_2, d_3, and d_4, respectively. After validating the support counts, only two stems b and f are found, each with the support of $\frac{3}{4}$. These two stems are used to form type-2 patterns $\langle (ab) \rangle$ and $\langle (af) \rangle$.

The process continues by recursively applied steps 2 and 3 on *P-pat* $= \langle (a) \rangle$ and a stem $x = \{b, f\}$. Taking *P-pat* $= \langle (a) \rangle$ and $x = b$, step 2 creates $\langle (ab) \rangle$-*idx* (Figure 2.8(b)). In the figure, the d_4 is not pointed to by any index element because it does not support $\langle (ab) \rangle$. Applying step 3 on this index set, a stem f is found, resulting in a type-2 pattern $\langle (abf) \rangle$. Recursively creating and mining the index set $\langle (abf) \rangle$-*idx*, no more stems can be found (Figure 2.8(c)). Therefore, the process goes on by considering *P-pat* $= \langle (a) \rangle$ and $x = f$. The creation and mining of $\langle (af) \rangle$-*idx* would not find any stem (Figure 2.8(d)). All subsequent find-then-index processes on stem a with *P-pat* $= \langle\ \rangle$ now finish.

When the database is too large to fit into the memory, MEMISP mines the sequential patterns in two database scans using a partition-and-validation technique. The large database is partitioned so that each partition can be handled in memory by the algorithm. By mining these partitions, the algorithm collects the set of potential sequential patterns. In order to be frequent in the whole database, a sequence should be frequent in at least one of the partitions. Then the true sequential patterns can be identified through support counting against each data sequence in the database with only one extra database pass. Therefore, the MEMISP can mine the database of any size in two passes of database scan.

2.3 Mining Episodes and Periodic Patterns

The problem of discovering episodes from a long sequence of events was introduced by Mannila *et al.* (1995). An episode is a collection of events that occur relatively close to each other in a certain (partial) order, whose total span of time is constraint by a window. There are two basic approaches for discovering episodes. The first approach is based on the occurrences of the patterns in a sliding window along the sequence and used by the WINEPI algorithm (Mannila et al. 1995, Mannila, Toivonen & Verkamo 1997). The second one relies on the notion of minimal occurrences of the patterns and used by the MINEPI algorithm (Mannila & Toivonen 1996, Mannila et al. 1997). Further studies have been undertaken into variations that attempt to extend the basic episode discovery framework, either by improving the algorithms or by enriching the generated patterns. Furthermore, other studies have also been conducted that focus on the discovery of periodic patterns. A periodic pattern is an ordered list of events that repeats itself in the sequence of events.

This section reviews previous studies on the discovery of episodes and periodic patterns. In Section 2.3.1, several terms related to the event sequences and episodes are introduced. Section 2.3.2 describes two approaches of mining episodes by WINEPI and MINEPI. Section 2.3.3 discusses several extensions of episode mining. Previous studies on periodic patterns, especially partial periodic patterns, are presented in Section 2.3.4.

2.3.1 Preliminary Definitions

In the episode model, the data is a history of events, where each event has a type and a time of occurrence. Given a set E of event types, an *event* is a pair (e, t), where $e \in E$ and t is its occurrence time. An *event sequence* is a triple (s, T_s, T_e), where T_s is the starting time of the sequence, T_e is the ending time, and s is an *ordered sequence of events* of the form

$$s = \langle (e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n) \rangle$$

where e_i is an event type, and t_i is the associated occurrence time, with $T_s \leq t_i \leq t_{i+1} < T_e$ for all $i = 1, \ldots, n-1$. The following is an example of event sequence $(s, 1, 16)$ consisting of ten events:

$$\langle (a, 2), (b, 3), (a, 7), (c, 8), (b, 9), (d, 11), (c, 12), (a, 13), (b, 14), (c, 15) \rangle \qquad (2.1)$$

An *episode* α is a triple $(V, \leq, g)$, where V is a set of nodes, $\leq$ is a partial order on V, and $g \colon V \to E$ is a mapping that associates each node in the episode with an event type. A *size* of α, denoted $|\alpha|$, is $|V|$. In simpler terms, an episode is a partially ordered set of event types. When the order among the event types of an episode is total, it is called a *serial* episode. However, when there is no order at all, the episode is called a *parallel* episode. Parallel episodes are somewhat similar to itemsets (Section 2.2.1). As an example, $(a \to b \to c)$ is a serial episode of size 3 (the arrow is used to emphasize the total order). A parallel episode of size 3 with event types a, b, and c is denoted as (abc). In the first episode, each of the events in the episode occurs in order, while in the second, the events occur but the order is not important.

Let α and β be two episodes. An episode β is a *subepisode* of α, denoted as $\beta \preceq \alpha$, if all the event types in β appear in α as well, and if the partial order among the event types of β is the same as that for the corresponding event types in α. As an example, $(a \to c)$ is a subepisode of the serial episode $(a \to b \to c)$, while $(b \rightarrow a)$ is not. In the case of parallel episodes, this order constraint is not imposed, so every subset of the event types of an episode corresponds to a subepisode.

An episode is said to occur in an event sequence if there exist events in the sequence occurring in exactly the same order as the prescribed in the episode. For example, in the example event sequence (Equation 2.1), the events $(a, 2), (b, 3)$, and $(c, 8)$ constitute an occurrence of the serial episode $(a \rightarrow b \rightarrow c)$, while the events $(a, 7), (b, 3)$, and $(c, 8)$ do not, because for this serial episode to occur, a must occur before b and c. Both of these sets of events, however, are valid occurrences of the parallel episode (abc), since there are no restrictions with regard to the order in which the events must occur for parallel episodes.

2.3.2 Two Basic Approaches for Mining Episodes

2.3.2.1 Mining Episodes with WINEPI Algorithm

In this approach, the episodes are mined over a single event sequence and their statistical significance is measured as a percentage of windows containing the episodes (frequency). An episode is considered interesting if it fits into a specific window width, which is given by the user.

A *window* on event sequence $S = (s, T_s, T_e)$ is an event sequence $W = (w, t_s, t_e)$, where $t_s < T_e$, $t_e > T_s$, and w consists of events (e_i, t_i) from s where $t_s \leq t_i < t_e$. The *width* of the window W is denoted as $width(W) = t_e - t_s$. Given an event sequence $S = (s, T_s, T_e)$ and a window width win, $\mathcal{W}(S, win)$ denotes the set of all windows W on S such that $width(W) = win$. The number of windows in $\mathcal{W}(S, win)$ is $T_e - T_s + win - 1$.

The frequency is defined as a percentage of windows containing the episode. Given an event sequence S and a window width win, the *frequency* of an episode α in S is

$$fr(\alpha, S, win) = \frac{|\{w \in \mathcal{W}(S, win)| \ \alpha \text{ occurs in } W\}|}{|\mathcal{W}(S, win)|}$$

An episode α is a *frequent* if its frequency exceeds a given frequency threshold min_fr.

Given a sequence S, a class $\mathcal{E}$ of episodes, a window width win, and a frequency threshold *min_fr*, the task is to find all frequent episodes from S.

WINEPI discovers frequent episodes using the candidate generation-and-test principle of the Apriori algorithm (Agrawal & Srikant 1994). It starts by computing frequent episodes of size 1. These episodes are then combined to form candidate episodes of size 2, and then, by counting their frequencies, frequent episodes of size 2 are obtained. This process continues until all frequent episodes of all sizes are found. The definition of episode frequency guarantees that all subepisodes of a frequent episode are also frequent. Therefore, the candidate generation step considers an episode as a candidate only if all its subepisodes have been found frequent.

Once a set of all frequent episodes has been found, it can be used to generate episode rules. In this approach, an episode rule is a rule of the form $\beta \rightarrow \alpha$, where α and β are frequent episodes such that $\beta \prec \alpha$. Given an event sequence S and a window width win, the confidence of the rule is $\frac{fr(\alpha,S,win)}{fr(\beta,S,win)}$. The rule holds in S if its confidence exceeds a given confidence threshold *minconf*.

2.3.2.2 Mining Episodes with MINEPI algorithm

The episode discovery framework previously described employs the windows-based frequency measure for episodes. As an alternative approach, Mannila *et al.* (1997) have proposed the MINEPI algorithm based on the concept known as *minimal occurrences* of episodes. The minimal occurrence approach looks at the exact occurrences of episodes and the relationships between those occurrences, instead of looking at the windows and only considering whether an episode occurs in a window or not. By focusing on the occurrences of episodes, this approach facilitates the discovery of rules with two window widths, one for the left hand side and one for the whole rule, such as *if a and b occur within 15 seconds of one* *another, then c follows within 20 seconds.*

Given an episode α and an event sequence S, the interval $[t_s, t_e)$ is a *minimal occurrence* of α in S if the following two conditions are met:

1. α occurs in the window $W = (w, t_s, t_e)$ on S
2. α does not occur in any proper subwindow on W, i.e., not in any window $W' = (w', t'_s, t'_e)$ on S such that $t_s \leq t'_s$, $t'_e \leq t_e$, and $width(W') < width(W)$

The set of minimal occurrences of an episode α in a given event sequence is denoted by $mo(\alpha) = \{\ [t_s, t_e)|\ [t_s, t_e)$ is a minimal occurrence of $\alpha\}$.

Instead of using frequency, MINEPI uses the concept of support. The support of an episode α in an event sequence S is the number of minimal occurrences of α, which is $|mo(\alpha)|$. Given a minimum support *minsup*, an episode α is *frequent* if $|mo(\alpha)| \geq minsup$.

An *episode rule* is an expression $\beta[win_1] \rightarrow \alpha[win_2]$, where β and α are frequent episodes such that $\beta \prec \alpha$, and win_1 and win_2 are integers. This rule means that if the episode β has minimal occurrence at interval $[t_s, t_e)$ with $t_e - t_s \leq win_1$, then the episode α occurs at interval $[t_s, t'_e)$ for some t'_e such that $t'_e - t_s \leq win_2$. The confidence of an episode rule $\beta[win_1] \rightarrow \alpha[win_2]$ is defined as

$$\frac{|\{[t_s, t_e) \in mo_{win_1}(\beta) \mid occ(\alpha, [t_s, t_s + win_2))\}|}{|mo_{win_1}(\beta)|}$$

Given win_1 and β, $mo_{win_1}(\beta) = \{[t_s, t_e) \in mo(\beta) \mid t_e - t_s \leq win_1\}$. Further, given α and an interval $[u_s, u_e)$, define $occ(\alpha, [u_s, u_e)) =$ true if and only if there exists a minimal occurrence $[u'_s, u'_e) \in mo(\alpha)$ such that $u_s \leq u'_s$ and $u'_e \leq u_e$.

Given an event sequence S, a class $\mathcal{E}$ of episodes, and a time bounds win_1 and win_2, the task is to find all frequent episode rules of the form $\beta[win_1] \rightarrow \alpha[win_2]$, where $\alpha, \beta \in \mathcal{E}$, and $\beta \prec \alpha$.

Although MINEPI is an Apriori-based algorithm and consists of several iterations, it only requires one scan of the input sequence for discovering frequent episodes. In the first iteration, MINEPI computes the minimum occurrences of all episodes of size 1 from the input sequence and determines frequent episodes of size 1. In the remaining iterations, having found the set F_k of frequent episodes of size k ($k \geq 1$), the algorithm creates the set C_{k+1} of candidate episodes of size $k+1$. Then, the algorithm finds out which candidate episodes $\alpha \in C_{k+1}$ are really frequent by forming the set $mo(\alpha)$. If α is generated from α_1 and α_2, $mo(\alpha)$ is determined by performing a temporal join between $mo(\alpha_1)$ and $mo(\alpha_2)$.

After all frequent episodes have been found, the information about their minimal occurrences can be used to generate episode rules. For an episode rule $\beta[win_1] \rightarrow \alpha[win_2]$, its confidence can be computed as follows. For each $[t_s, t_e) \in mo(\beta)$ with $t_e - t_s \leq win_1$, locate the minimum occurrence $[u_s, u_e)$ of α such that $t_s \leq u_s$ and $[u_s, u_e)$ is the first interval in $mo(\alpha)$ with this property. Then check whether $u_e - u_s \leq win_2$.

2.3.3 Extensions of Episode Model

Several studies have been proposed to extend the episode model described earlier. Mannila and Toivonen (1996) extend MINEPI for discovering generalized episodes, which allow events in the episodes to have attributes. Casas-Garriga (2003) proposes an episode model that allows the user to choose the maximum distance between events in the episode, instead of the fixed window width as described earlier. The algorithm called Seq-Ready&Go automatically adjusts the window width based on the length of the episodes being counted. However, Meger and Rigotti (2004) have shown that this algorithm was not complete, and they have proposed a new algorithm, WinMiner, to discover episode rules under the maximum gap constraint and find, for each rule, the window size corresponding to a local maximum of confidence.

Harms *et al.* (2001) propose the Gen-REAR algorithm to generate representative episodal association rules (REAR) from a set of frequent closed episodes, which are generated by extending WINEPI. Later, the authors present the MOW-CATL approach for generating episodal association rules using minimal occurrences with constraint and time lag (MOWCATL) (Harms, Deogun & Tadesse 2002, Harms & Deogun 2004). Instead of generating episode rules from frequent episodes, MOWCATL directly mines rules with an antecedent part and a consequent part separated by a time lag. This way the rules can always be used for prediction and separate maximum length constraints can be specified for both parts of the rules. The difference between Gen-REAR and MOWCATL is that Gen-REAR uses a sliding window approach, and does not allow for a delay in time embedded within the relationships.

Mooney and Roddick (2004) propose the discovery of interacting episodes. In order to increase the expressiveness of the resulting episodes, episodes are combined with a subset of Allen's relations (Allen 1983) to express temporal relations among episodes. Initially, frequent episodes are mined, then *interacting episodes* are searched within each episode. Interacting episodes are subepisodes of the episode for which relations *during*, *overlaps*, or *meets* holds.

Padmanabhan & Tuzhilin (1996) extend the work of Mannila *et al.* (1995) to find temporal logic patterns in temporal databases. It proposes the use of *first-order temporal logic (FOTL)*, with operators like *Since*, *Until*, *Next*, *Always*, *Sometimes*, *Before*, *After*, and *While*, to express patterns in temporal databases. As an example, a serial episode $a \to b \to c$ can be expressed in temporal logic as *a before b and b before c.*

A graph-based approach to locate episode occurrences in a sequence has been described by Tronícek (2001). The idea is to employ a preprocessing step to build a finite automaton called DASG (Directed Acyclic Subsequence Graph), which accepts a string if and only if it is a subsequence of the given input sequence. This work is more suited for search and retrieval applications than for discovery of all frequent episodes.

Laxman *et al.* (2004) propose two new episode counting methods, namely, the *non-overlapped* occurrence count and the *non-interleaved* occurrence count, which are based on directly counting some suitable subset of occurrences of episodes.

These two methods are automata-based counting schemes and have the same complexity as the windows-based counting of WINEPI (Mannila et al. 1997).

2.3.4 Mining Periodic Patterns

A periodic pattern is an ordered list of events which repeats itself in the sequence of events. Two types of periodic patterns are *full periodic* patterns and *partial periodic* patterns. In full periodic patterns, every point in time contributes to the periodicity. For example, all the days in the year approximately contribute to the season cycle of the year. In partial periodic patterns only some of the time periods may exhibit periodic patterns. For example, a pattern stating that the prices of a specific stock are high every Friday and low every Tuesday is a partial periodic pattern, since it does not describe any regularity for the other week days (Aref et al. 2004). Cyclic association rules (see Chapter 2.1.3) are partial periodic patterns with *perfect* periodicity in the sense that each pattern reoccurs in every cycle, with 100% confidence (Han, Dong & Yin 1999).

In temporal data mining, most studies in mining periodic patterns have concentrated on discovering partial periodic patterns. The first approach for mining partial periodic patterns in symbolic time series is described by Han *et al.* (1998). They present an Apriori-like algorithm for mining partial periodic patterns and use the pattern confidence to measure how significant periodic patterns are. The confidence of a pattern is defined as the occurrence count of the pattern over the maximum number of periods of the pattern length contained in the input sequence. For example, the pattern $a * b$ is a partial periodic pattern of length 3. The length of the pattern is usually called the period of the pattern. The wild card symbol '*' is introduced to allow partial periodicity. It denotes the 'do not care' positions in a pattern, which can match any single set of symbols. Given a sequence of symbols $a\{b, c\}baebaced$, the occurrence count of $a * b$ is 2, and its confidence is $\frac{2}{3}$, where 3 is the maximum number of periods of length 3.

It is pointed out by Han *et al.* (1999) that the Apriori property used in the earlier algorithm is not as effective for mining partial periodic patterns as it is for mining frequent itemsets. This is because in the frequent itemset mining the number of frequent k-itemsets falls quickly as k increases, while in the partial periodic mining the number of frequent k-patterns shrinks slowly with increasing k. Therefore, based on the so called *max-subpattern hit set* property, Han *et al.* (1999) present a more efficient method for mining partial periodic patterns. An incremental version of this method is proposed by Aref *et al.* (2004). Elfeky *et al.* (2004) propose an algorithm for mining periodic patterns in a single pass of the data.

Periodicity mining algorithms usually require the user to specify the length of the period. To overcome this, Berberidis *et al.* (2002) have employed the Fast Fourier Transform of binary vectors for each symbol to obtain candidates period lengths, which then can be used by the algorithm of Han *et al.* (1999).

All of the above studies consider only synchronous periodic patterns. Periodicity may be occasionally disturbed due to some misses or skips in the sequence of pattern occurrences. This happens when some random noise events get inserted in between a sequence. Yang *et al.* (2000) propose to mine asynchronous periodic patterns that have missing occurrences and whose occurrences may be shifted due to disturbance. They introduce two parameters, namely *min_rep* and *max_dis*, to specify the minimum number of repetitions that is required within each segment of non-disrupted pattern occurrences and the maximum allowed disturbance between any two successive valid segments. The idea is that a pattern needs to repeat itself at least a certain number (*min_rep*) of times to demonstrate its significance and periodicity. However, the disturbance between two valid segments has to be within some reasonable bound, that is, *max_dis*.

Huang and Chang (2004a) propose a general model for mining asynchronous periodic patterns where each valid segment is required to be of maximum and at least *min_rep* contiguous matches of the pattern. They also propose three algorithms to discover the patterns, namely, SPMiner, MPMiner, and CPMiner.

Later, the PTV (Progressive Timelist-based Verification) algorithm is proposed to improve the performance of SPMiner and MPMiner (Huang & Chang 2004*b*).

Yang *et al.* (2001) consider the mining of surprising periodic patterns from a sequence of events. Instead of using the number occurrences, they use information gain metric to measure the degree of surprise (or significance) of the patterns. The goal is to search for patterns whose occurrence is significantly greater than expectation. Then, the same authors extend the information gain measure to include a penalty for gaps between pattern occurrences (Yang, Wang & Yu 2002). Recently, extensions for more robust pattern matching allowing random replacement (Yang, Wang & Yu 2003) and meta patterns (Yang, Wang & Yu 2004) have been proposed.

2.4 Summary

This chapter has provided a review of the discovery of temporal patterns from point-based sequential data. The review has considered, in some detail, methods for discovering temporal association rules, sequential patterns, frequent episodes, and (partial) periodic patterns. This review serves as a foundation upon which further research into discovering temporal rules from interval sequence data can be based. Furthermore, as a result of research undertaken in the construction of this review, a new type of rule called inter-transaction relative temporal association rules has been proposed. The mining of relative temporal association rules is presented in the next chapter (Chapter 3).

Chapter 3

Mining Relative Temporal Association Rules

As mentioned in the previous chapter, due to its richer rule semantics, the problem of finding temporal association rules has become an important research topic and is receiving a great deal of research interest. A short survey of temporal association rules is given together with an earlier discussion of this work in Winarko and Roddick (2003).

However, with some notable exceptions, both classical (static) and currently proposed temporal association rules express associations among items within the same transaction. It would be interesting to find the rules that represent some associative relationship among the field values from different transactions. Taking medical data as an example, a rule can have the following form:

$$(abc) \overset{<}{\rightarrow} (de),$$

which is equivalent to an assertion that *patients who have attributes (such as symptoms) a, b and c are also likely to later have recorded attributes (such as other symptoms or diagnoses) d and e.* Attributes a, b and c and attributes d and e are from different observations[1]. In this rule, the $<$ annotation means that

[1]The terminology of *transaction* is replaced by that of *observation* as the former is barely

a, b and c occur *before* d and e (á la the temporal constraints given by Allen, Freksa and others (Allen 1983, Freksa 1992, Roddick & Mooney 2005)).

As stated by Lu *et al.* (1998), rules that represent associations among items within the same transaction are called *intra-transaction* associations, while those that represent associations among items from different transactions are called *inter-transaction* associations. Sequential pattern discovery is considered intra-transactional in nature because each sequence is treated as one transaction and the mining process finds similarities among the sequences (Lu, Han & Feng 1998). Apart from the other more obvious differences, the model proposed here differs from the work on multi-dimensional inter-transaction mining (Lu et al. 1998, Tung, Lu, Han & Feng 1999, Lu, Feng & Han 2000) in that this work is focused on inter-transaction rule with the same client, object, customer, or patient identifier

Basing our work on the above ideas, a new type of rule, called *inter-transaction relative temporal association rules*, can be proposed and in this chapter an algorithm for mining them is described. The chapter is organized as follows. In Section 3.1, the relative temporal association rule model is described. In Section 3.2, the algorithm for mining temporal relative association rules is introduced. A performance study is presented in Section 3.3.

3.1 Model Description

In classical association rule, an *itemset* is defined as a non-empty set of items. To distinguish the itemsets in the classical association from the itemsets in this model, the itemsets in this model are termed *relative itemsets.* A relative itemset has the form of $l_1 < l_2 < \cdots < l_n$, where each l_i is an itemset $(j_1 \cdots j_k)$. Without loss of generality, this relative itemset can also be written as $\langle l_1\, l_2 \ldots\, l_n \rangle$.

Given a database D of client observations, each observation consists of the following fields: client-id, observation-time, and the items present in the observation.

applicable for datasets containing medical and other data. However the two terms are broadly interchangeable.

Client ID	Observation time	Items
1	5	h
1	10	c
2	10	a, b, d
2	15	c
2	20	d, f, g
2	25	b
3	15	c, e, g
4	5	c
4	10	d, g
4	15	d
4	20	b

(a)

Client ID	Client sequence
1	⟨ (h) (c) ⟩
2	⟨ (abd) (c) (dfg) (b) ⟩
3	⟨ (ceg) ⟩
4	⟨ (c) (dg) (d) (b) ⟩

(b)

Figure 3.1: Example database

Each client-id can have more than one observation with a different observation-time. This chapter follows the convention of Agrawal and Srikant (1995) and uses the term *client sequence* to refer to the list of observations, ordered by increasing observation timestamp.

Consider the example database shown in Figure 3.1. In the left table, the database contains a set of observations that have been sorted by observation timestamp within the client-id. In the right table, the database is presented as a set of client sequences. This dataset is used as a vehicle for describing the model.

Given a relative itemset $l_1 < l_2 < \cdots < l_n$ and a client sequence $\langle O_1\, O_2 \cdots O_m \rangle$, where O_i is the i-th observation and $m \geq n$, the sequence *contains* the relative itemset if there exist integer $i_1 < i_2 < \cdots < i_n$ such that $l_1 \subseteq O_{i_1}$, $l_2 \subseteq O_{i_2}$, $\ldots$, $l_n \subseteq O_{i_n}$. For example, a client sequence $\langle (abd)(c)(dfg)(b) \rangle$ contains a relative itemset $(b) < (c) < (dg)$ because $(b) \subseteq (abd)$, $(c) \subseteq (c)$, and $(dg) \subseteq (dfg)$. However, this relative itemset is not contained in the $\langle (c)(dg)(d) \rangle$.

Given the database D, the *support* of a relative itemset is the fraction of all sequences in D that contain the relative itemset. A relative itemset is *frequent* in D if its support exceeds a given minimum support threshold *minsup*. As an example, given the minimum support *minsup* = 40%, the relative itemset $(c) < (dg) < (b)$ is frequent becuase it has the support of 50% (supported by clients 2 and 4).

Although this observation dataset is similar to the one used for mining sequential patterns discussed in Agrawal and Srikant (1995), the rules generated by the proposed model are different. In addition, the temporal nature of the rule means that, unlike static rules, the same attribute can occur on both sides of the rule. That is, in this model, it is possible to have the following rule:

$$(ab) \overset{\leq}{\rightarrow} (a) \tag{3.1}$$

meaning that attributes a and b imply the continuation or re-occurrence of a in later periods, while

$$(d) \overset{>}{\rightarrow} (cd) \tag{3.2}$$

means that attribute d was often preceded by c and d in earlier time periods. Finally, note that

$$(d) \overset{>}{\rightarrow} (c) \overset{\leq}{\rightarrow} (d) \tag{3.3}$$

has a different semantics from Rule (3.2). In Rule (3.2) c and d must occur in the same timestamp while in Rule (3.3) the items c and d occur in different timestamps.

Formally, a *relative temporal association rule*, for the purposes of this work, is one structured as follows:

$$X \overset{temprel}{\longrightarrow} Y, \quad quals \tag{3.4}$$

where X and Y are frequent relative itemsets, *temprel* is a temporal relationship taken from, for example, those suggested by Allen (1983), Freksa (1992) and others, and *quals* is some rule quality qualifications (such as support and confidence). Based on the current definition of relative itemsets, the only possible temporal relationships are $<$ and $>$ (i.e. *before* and *after*, respectively), however later work to accommodate intervals will change this to handle more advanced constraints (see Chapter 5).

Furthermore, according to this definition, the rules can be used not only for predicting the future (Rule (3.1)), but also for predicting the past (Rule (3.2)). The important of predicting the past is illustrated in the following situation. If we are interested in detecting a certain event B that we cannot measure directly, it may manifest on other (measurable) events A_i that appear *after* the occurrence of B. In such a case, a rule may be used to predict an event in the past (Höppner 2003).

As in mining classical association rules, two measures of relative temporal association rules are: *support* and *confidence*. The support the rule is the support of X *temprel* Y. The confidence of the rule is defined as:

$$conf(X \xrightarrow{temprel} Y) = \frac{sup(X\ temprel\ Y)}{sup(X)} \tag{3.5}$$

Given a database D, the minimum support *minsup* and the minimum confidence $minconf$, the rule $X \xrightarrow{temprel} Y$ holds in D if its support exceeds *minsup* and its confidence exceeds $minconf$.

3.2 Mining Relative Temporal Association Rules

This section discusses a method for mining relative temporal association rules and in particular some issues and problems in the process. The problem of mining temporal relative association rules can be decomposed into two subproblems: finding all frequent relative itemsets and, for every frequent relative itemset, generating the relative temporal association rules.

3.2.1 Finding Frequent Relative Itemset

The algorithm to generate large relative itemsets consists of three steps: litemset phase, transform phase, and relative itemset phase. These main steps are based on the AprioriAll algorithm (Agrawal & Srikant 1995). Some modifications have been made in the data structure in order to make the algorithm more efficient (Bodon 2003).

Litemset Phase

This phase generates the set of all frequent itemsets within the same transaction (*intra-transaction*). This can be done by using any available algorithms, such as Apriori (Agrawal & Srikant 1994), or FP-growth (Han, Pei & Yin 2000), with the exception that the definition of support is modified. In these non-temporal algorithms, the support of an itemset is defined as the fraction of *observations* in which an itemset is present. Here, the support for an itemset is defined as the fraction of clients in which an itemset appears (at any time). For example, even though the itemset (b) appears in 3 observations (Figure 3.1), its support is said to be $\frac{2}{4}$ as it appears only in the second and fourth clients. Given a minimum support of 40%, a set of frequent itemsets that can be generated from the example database is shown in Figure 3.2(a).

Transform Phase

In this phase, each observation is replaced by the set of all frequent itemsets contained in that observation. Before the transform phase takes place, for efficiency, the set of frequent itemsets generated in previous phase is first mapped into a set of integers, as shown in Figure 3.2(a). The transformation process can be outlined as follows.

First, all non-frequent items are removed from the observation. As an example, the first observation of the second client, (abd), contains a non-frequent

Frequent itemset	Support	Mapped to
(b)	0.50	1
(c)	1.00	2
(d)	0.50	3
(g)	0.75	4
(dg)	0.50	5

(a) Frequent itemsets resulting from the litemset phase

Client ID	Transformed client sequence	After mapping
1	{ (c) }	-
2	{ (b), (d) } { (c) } { (d), (g), (dg) } { (b) }	{1, 3} {2} {3, 4, 5} {1}
3	{ (c), (g) }	-
4	{ (c) } { (d), (g), (dg) } { (d) } { (b) }	{2} {3, 4, 5} {3} {1}

(b) Transformed database resulting from the transform phase

F_1	F_2		F_3
⟨ 1 ⟩ [0.50]	⟨ 2 1 ⟩ [0.50]	⟨ 3 3 ⟩ [0.50]	⟨ 2 3 1 ⟩ [0.50]
⟨ 2 ⟩ [1.00]	⟨ 2 3 ⟩ [0.50]	⟨ 4 1 ⟩ [0.50]	⟨ 2 4 1 ⟩ [0.50]
⟨ 3 ⟩ [0.50]	⟨ 2 4 ⟩ [0.50]	⟨ 5 1 ⟩ [0.50]	⟨ 2 5 1 ⟩ [0.50]
⟨ 4 ⟩ [0.75]	⟨ 2 5 ⟩ [0.50]		⟨ 3 3 1 ⟩ [0.50]
⟨ 5 ⟩ [0.50]	⟨ 3 1 ⟩ [0.50]		

(c) Frequent relative itemsets resulting from the relative itemset phase

Figure 3.2: Output of each phase of the algorithm

item a, so it must be removed from the observation, resulting in the observation (bd). If this removal creates an empty observation, then this observation is not retained in the transformed client sequence. Consider the first observation of the first client (Figure 3.1). After the non-frequent item h is deleted, this observation becomes empty, thus it is dropped from the transformed client sequence.

Next, any client sequence that currently contains only one observation is dropped from the transformed sequence. The client sequence with one observation could only produce a frequent relative itemset of size 1, which has been found during the litemset phase. Therefore, the first and the third client sequences are dropped from the transformed sequence. However, they still contribute to the count of the total number of clients.

Finally, each remaining observation is replaced by the set of all frequent itemsets contained in that observation, then mapped into the set of integers representing frequent itemsets. As an example, the observation (dg) is replaced with $\{(d), (g), (dg)\}$, and then mapped into $\{3, 4, 5\}$, where integers $3, 4$, and 5 represent frequent itemsets (d), (g), and (dg), respectively. The observation (bd) is replaced with $\{(b), (d)\}$ because an itemset (bd) is not frequent, then $\{(b), (d)\}$ is mapped into $\{1, 3\}$. Given an example database D in the Figure 3.1, the transformed database D_T is shown in Figure 3.2(b). In the rest of this chapter, each integer value representing a frequent itemset is called an *item*.

Relative Itemset Phase

This phase generates frequent relative itemsets (*inter-transaction frequent itemsets*) from the transformed database D_T. The algorithm is based on the *Apriori* algorithm and is shown in Algorithm 3.1. In the algorithm, F_k is a set of frequent relative itemsets of size k and C_k is a set of candidate relative itemset of size k. The F_1 is initialized by using the set of frequent itemsets generated in the litemset phase, that is, $F_1 = \{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle, \langle 5\rangle\}$.

The candidate generation to obtain C_k is performed by joining F_{k-1}. Let p and q be frequent relative itemsets of size $(k-1)$ in F_{k-1}, p and q can be joined if their first $(k-2)$ items are in common. The resulting candidate relative itemsets are $p[1]\cdots p[k-2]p[k-1]q[k-1]$ and $p[1]\cdots p[k-2]q[k-1]p[k-1]$. For example, the relative itemset $\langle 2\ 1\rangle$ and $\langle 2\ 3\rangle$ in F_2 (Figure 3.2(c)) can be joined to create the candidates $\langle 2\ 1\ 3\rangle$ and $\langle 2\ 3\ 1\rangle$.

After this joining process, the pruning procedure will delete all candidate $c \in C_k$ such that c contains a relative itemset of size (k-1) that is not in F_{k-1}. For example, the candidate relative itemset $\langle 2\ 1\ 3\rangle$ is deleted from C_3 because it contains a relative itemset $\langle 1\ 3\rangle$ that is not in F_2. The pruning procedure cannot be applied during the generation of C_2. Therefore, if the size of F_1 is $|F_1|$, the

```
1: $F_1$ = {Frequent relative itemset of size 1};
2: for ($k = 2; F_{k-1} \neq \emptyset; k++$) do
3:     Generate $C_k$ from $F_{k-1}$;
4:     for each client sequence $s$ in $D_T$ do
5:         Increment the count of all candidate in $C_k$ that are contained in $s$;
6:     end for
7:     $F_k$ = Candidate in $C_k$ with minimum support;
8: end for
```

Algorithm 3.1: Pseudo code for Generating frequent relative itemsets

join process will produce C_2 whose size is equal to $|F_1| * |F_1|$.

To improve the performance of the algorithm, especially in dealing with the processing of C_2, the algorithm is modified using the methods suggested in Bodon (2003). First, to reduce the memory usage, C_2 is store in a two-dimensional array, instead of hash tree. Then, to accelerate the search, the support counting is done by pairing method. Consider a client sequence $\langle O_1\ O_2 \cdots O_n\rangle$ in the transformed database. Each item from an observation O_i is combined with each item from an observation O_j, where $i < j$, $i = 1, 2, \cdots, (n-1)$ and $j = 2, 3, \cdots, n$. Each time a pair of items (k, l) is found, the support of a candidate relative litemset $\langle k\ l\rangle$ is increased by one. For example, given a client sequence $\langle\{2\}\ \{4,5\}\ \{3\}\rangle$, the resulting combinations are (2, 4), (2, 5), (2, 3), (4, 3), and (5, 3).

3.2.2 Generating Relative Temporal Association Rules

After all frequent relative itemsets have been found, relative temporal association rules can be generated from them using the following procedure. For a frequent relative itemset $\alpha = \alpha_1 < \alpha_2 \cdots < \alpha_n$, α is divided into two parts, α_{left} and α_{right}, such that $\alpha_{left} = \alpha_1 < \alpha_2 < \cdots < \alpha_i$ and $\alpha_{right} = \alpha_{i+1} < \alpha_{i+2} < \cdots < \alpha_n$, where $i = 1, 2, \cdots (n-1)$. Each pair of α_{left} and α_{right} can generate two possible rules: $\alpha_{left} \overset{\leq}{\rightarrow} \alpha_{right}$ or $\alpha_{right} \overset{\geq}{\rightarrow} \alpha_{left}$.

This procedure is illustrated using the following example. Consider a frequent relative itemset $\alpha = \langle 2\ 5\ 1\rangle$, representing the frequent relative itemset $\alpha = (c) < (dg) < (b)$, with the support of 50% in Figure 3.2(c). This frequent relative itemset produces two pairs of α_{left} and α_{right}:

$$\alpha_{left} = (c) < (dg) \text{ and } \alpha_{right} = (b) \tag{3.6}$$

and

$$\alpha_{left} = (c) \text{ and } \alpha_{right} = (dg) < (b) \tag{3.7}$$

A pair in (3.6) generate two possible rules: $(c) < (dg) \overset{<}{\rightarrow} (b)$ and $(b) \overset{>}{\rightarrow} (c) < (dg)$. The first rule has the confidence of $conf = \frac{sup(c<dg<b)}{sup(c<dg)} = \frac{2}{2} = 100\%$. Similarly, the second rule also has the confidence of 100%. Given a minimum confidence $minconf = 75\%$, both rules hold in the database. On the other hand, a pair in (3.7) generate two possible rules: $(c) \overset{<}{\rightarrow} (dg) < (b)$ and $(dg) < (b) \overset{>}{\rightarrow} (c)$, with the confidences of 50% and 100%, respectively. As a result, only the second rule holds.

3.3 Evaluation

To evaluate the proposed method, the algorithm to generate frequent relative itemsets was implemented using Java language. Two sets of experiments were conducted on a 2.4GHz, 512 Mb PC running Windows 2000 Professional. The experiments were run using synthetic data. The method to generate synthetic observations is described in Agrawal and Srikant (1994). The input parameters are shown in Table 3.1. Eight datasets were generated by setting the values of $N = 1000$, $|L| = 2000$, $|D|$ = 10K, 25K, 50K, 75K, 100K, and varying the values of $|C|$, $|T|$, and $|I|$. The eight datasets are shown in Table 3.2.

The first set of experiments was to evaluate the performance of the algorithm by measuring its processing times and counting the number of frequent relative itemsets generated by the algorithm. The algorithm was run on the first four datasets (Table 3.2) by varying the values of minimum support. Figure 3.3 shows the execution times of the algorithm as the minimum support threshold is decreased from 10% to 2%. When the minimum support is high, the execution time

Table 3.1: Parameters

$\|D\|$	Number of clients
$\|C\|$	Average number of observation per client
$\|T\|$	Average number of items per observation
$\|I\|$	Average size of a frequent itemset
$\|L\|$	Number of frequent itemsets to be used
N	Number of items

Table 3.2: Eight datasets for the experiments

	$\|C\|$	$\|T\|$	$\|I\|$	# observations
D10K-C5-T5-I2	5	5	2	54,499
D10K-C5-T5-I3	5	5	3	54,722
D10K-C10-T3-I2	10	3	2	104,643
D10K-C10-T3-I3	10	3	3	105,589
D25K-C5-T5-I3	5	5	3	137,582
D50K-C5-T5-I3	5	5	3	275,697
D75K-C5-T5-I3	5	5	3	411,980
D100K-C5-T5-I3	5	5	3	550,011

is low as only limited number of frequent relative itemsets are produced, as shown in Figure 3.4.

The second set of experiments was performed to evaluate the scalability of the algorithm. It was done by running the algorithm on different sizes of databases, i.e., $|D|$ = 10K, 25K, 50K, 75K, and 100K, and varying the values of minimum supports. The values of other parameters were kept the same (see Table 3.2). It can be seen from Figure 3.5 that the algorithm has linear scalability with the size of databases.

It is also shown in Figure 3.6 the processing time required by each phase of the algorithm when $|D|$ = 100K and the minimum support is varied from 10% to 2%. It can be seen from the figure that the biggest portion of the processing time is spent during the litemset phase of the algorithm. Replacing the Apriori-based algorithm used in this phase with more efficient algorithm would make the algorithm running faster.

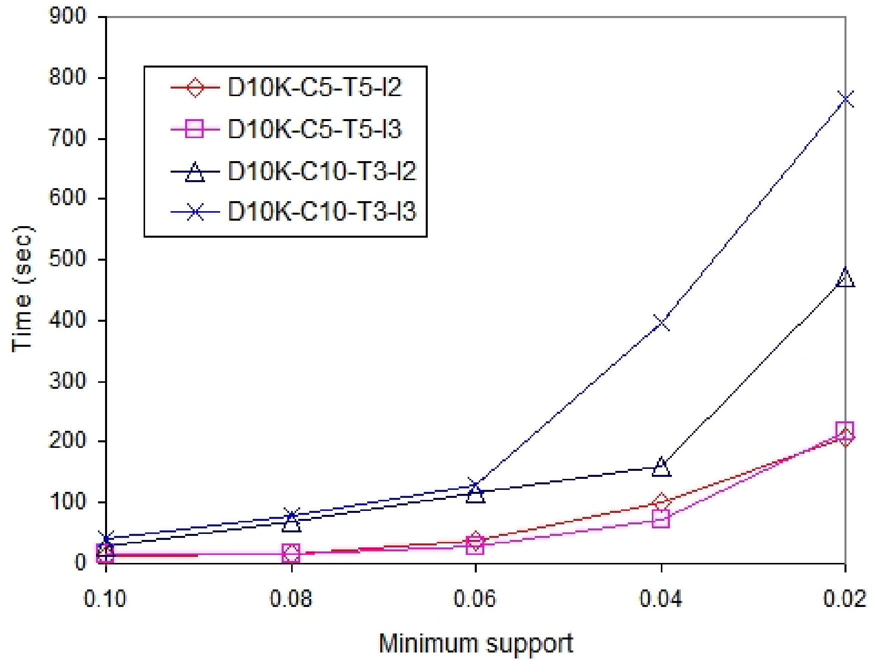


Figure 3.3: Effect of decreasing minimum support on the processing time

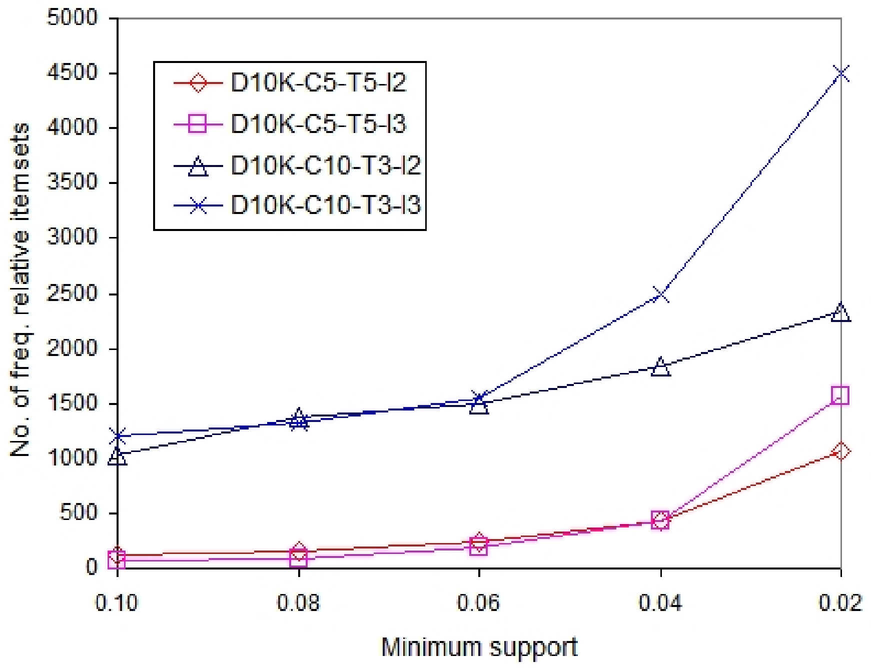


Figure 3.4: Effect of decreasing minimum support on the number of patterns

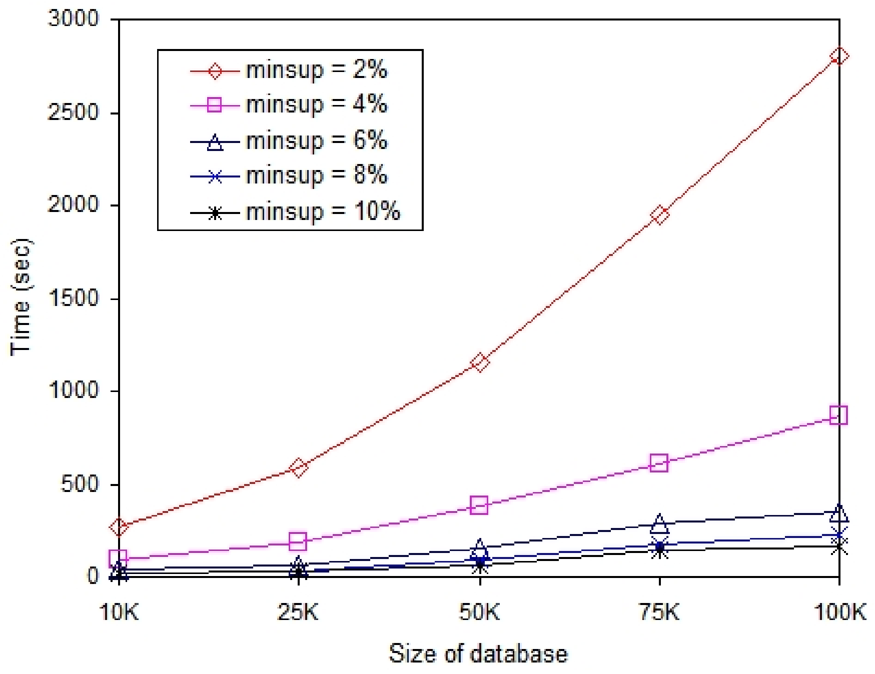


Figure 3.5: Effect of increasing database size on the processing time

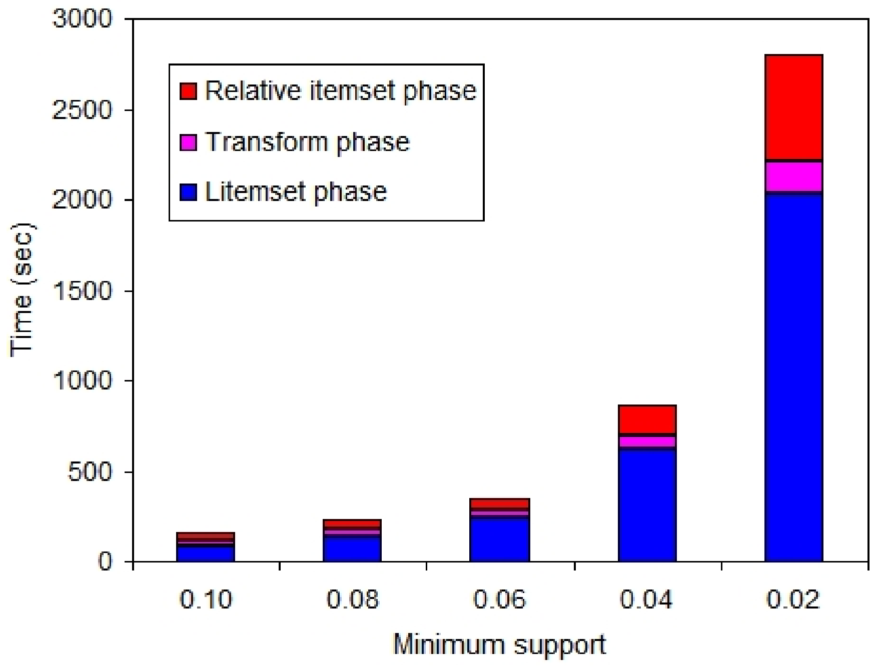


Figure 3.6: Processing time required by each phase of the algorithm

3.4 Summary

This chapter has proposed a new type of rules called inter-transaction relative temporal association rules. In order to discover the rules, a set of frequent relative itemsets are first generated using the algorithm similar to the AprioriAll algorithm. Some modifications to the AprioriAll algorithm have been introduced to improve the performance of the algorithm. Then, the rules are generated from the set of frequent relative itemsets. The experimental results showed that the algorithm is operational. However, this work also shows that basing the algorithm on more efficient algorithms, such as the ideas contained in the FP-Growth algorithm could be used instead. Other ideas of improving Apriori-based algorithms proposed by Bodon (2003) can also be used, for example, using trie data structure instead of hash-tree, and brave candidate generation.

The idea of inter-transaction relative temporal association rules has underlain the introduction of observation intervals which would yield a richer relative temporal constraint set. As a result, the discovery of richer temporal association rules has been proposed in this thesis. The data model used for the discovery is a database containing a set of interval sequences, where each interval sequence is a list of intervals during which the observation is valid. This topic is presented in Chapter 5.

Chapter 4

Review of Mining Time Interval Patterns

Chapter 2 has presented numerous studies in the temporal data mining task of pattern discovery. These studies concentrate on discovering temporal patterns from point-based sequential data, where events in the sequence are instantaneous. According to Böhlen *et al.* (1998), in many applications events are not instantaneous; they instead occur over a time interval. For instance, 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 several such events, a series of interval events is formed (Villafane et al. 2000). Because events are extended in time, different events may overlap in time and interact. Temporal data mining methods can be used to extract patterns from these interacting interval events.

Several applications exist where the discovery of temporal relations among interval events can provide insight about the operation of the system in question. Consider a sign language database that contains useful linguistic information on a variety of grammatical and syntactic structures, as well as manual and non verbal fields. Detecting relations between the above structures and fields could be useful to the linguist and may help them discover new type of relations they have not known before. Another application is in network monitoring. Multiple types of events occurring over certain time periods can be stored in a log, and the goal is to detect general temporal relations of these events that with high probability would describe regular patterns in the network (Papapetrou et al. 2005). Other application areas include analysis of weather data (Höppner 2001), muscle activity (Mörchen et al. 2004), audio data (Mörchen & Ultsch 2004), and video data (Mörchen 2006). Despite these facts, research on mining temporal patterns from interval-based sequential data has received little attention.

This chapter reviews previous studies related to the discovery of temporal patterns from interval-based sequential data. In this context, there are two types of interval data models commonly used for the pattern discovery, namely, interval sequence databases and a long sequence of intervals. While interval sequence databases can be given naturally (e.g., medical patient data, insurance contract, etc.), a long sequence of intervals is often obtained from converting numerical time series into a symbolic interval sequence. Moreover, most temporal patterns are formulated using Allen's temporal interval relations.

This chapter is organised as follows. Section 4.1 discusses the source of interval data. Section 4.2 describes temporal operators used to facilitate the formulation of temporal patterns. Section 4.3 discusses the discovery of temporal patterns from a long sequence of intervals, while Section 4.4 describes the discovery of temporal patterns from interval sequence databases.

4.1 Sources of Interval Data

Time series are one of the main sources of interval sequence data. Since many pattern discovery algorithms require symbolic interval sequences and cannot be applied to time series data, the time series are usually first segmented and transformed into sequences of labeled intervals. The labels denote a property in the original series within the associated time interval. Then from the sequence of labeled intervals, the algorithms discover frequent temporal patterns. This process is illustrated in Figure 4.1.

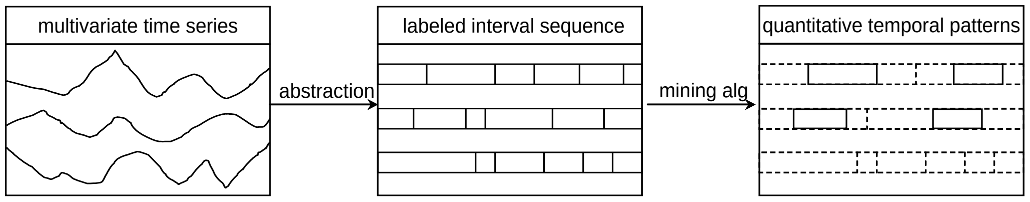


Figure 4.1: Transform time series into a sequence of intervals (Höppner, 2003)

Several methods for converting a continuous time series into a series of labeled intervals have been proposed. In general, there are two approaches for partitioning time series, supervised and unsupervised approaches (Höppner 2002, Höppner 2003). In the supervised approach, the attributes of interest are defined a priori and labels from this given set are assigned to portions of the time series. In other words, in the supervised approach, the shapes of interest are determined in advance. For non-zero length of time interval, there are seven basic shapes for describing local trends in a function f, corresponding to the possible combination of positive/zero/negative first and second derivatives, f' and f'' (Höppner 2002, Höppner 2003). These seven basic shape descriptors are *constant, linearly increasing, linearly decreasing, convexly increasing, convexly decreasing, concavely increasing*, and *concavely decreasing*, as shown in Figure 4.2. If these basic shapes are used to describe the time series, the series can be divided into subsequences by estimating the first and second derivatives via differencing. The main problem with this approach is how to distinguish noise from significant features. Noise makes the series oscillate around the true profile, and introduces a large number of tiny segments and local extrema.

There are at least two methods to deal with noise. The first method uses function approximation techniques and extracts the description of the time series from the approximating function, instead of the original series. In this method,

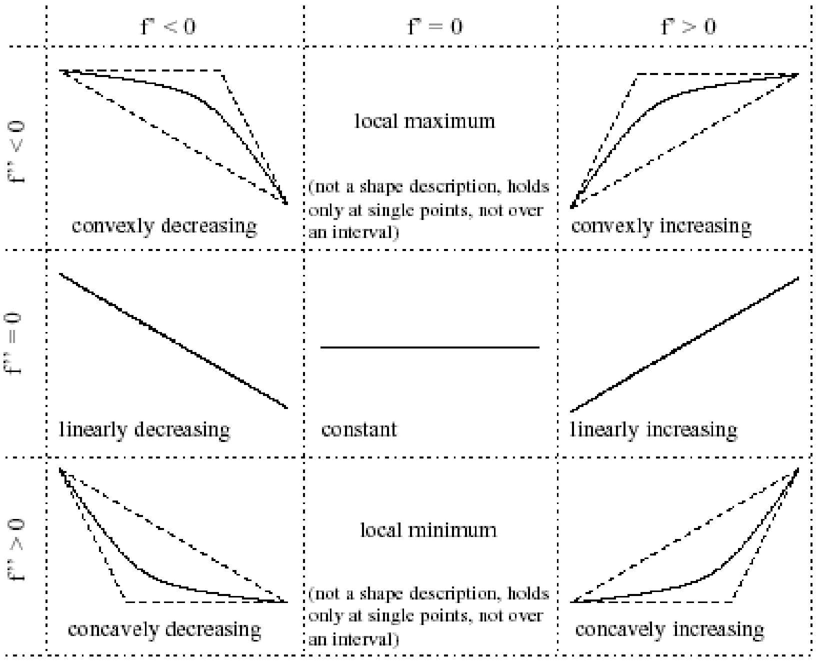


Figure 4.2: Seven basic shapes (Höppner, 2003)

the problem of handling noise is taken care of by the applied approximation techniques. The second method uses smoothing techniques to get more robust estimates of the first and second derivatives. In this case, the handling of noise corresponds to selecting an appropriate smoothing filter.

Figure 4.3 shows the partitioning of time series (weather data) into a sequence of labeled intervals using predefined labels (Höppner 2001). The kernel smoothing (Ramsay & Silverman 1997) has been applied to compensate the noise and to get more robust estimates of the first and second derivatives. Then, the smoothed series are segmented into highly increasing, increasing, level, decreasing, and highly decreasing segments.

On the other hand, in the unsupervised approach there is no such set of labels given in advance. The set of labels has to be derived from the data. This can be done by identifying similar parts in the time series via clustering. Clustering is usually used to partition data entities such that similar data objects belong to the same group and dissimilar data objects belong to different groups. In this approach, small portions of time series can be considered as data objects,

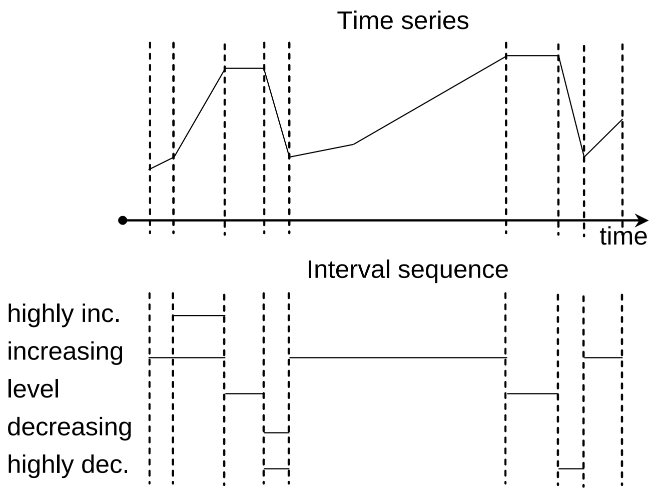


Figure 4.3: Example of partitioning time series using supervised approach (Höppner, 2001)

and every cluster can be considered as an inductively derived label for a group of similar portions. However, traditional clustering techniques partition a set of attribute vectors rather than portions of a time series. Therefore, the problem is how to represent segments of the original time series as new data entities that is more appropriate for traditional clustering methods.

There are four methods for learning shapes of time series by clustering (Höppner 2002, Höppner 2003). First, in *clustering of embedded subsequences*, a window of constant width is slid along the series. The content of each window is transformed into a vector of observations. Clustering is performed on this collection of subsequences (Das et al. 1998, Karimi & Hamilton 2000). However, Keogh *et al.* (2003) have claimed that clustering of time series subsequences is meaningless. This is because the output of subsequence time series clustering does not depend on input, and is therefore meaningless. Second, in *clustering of embedded models*, an abstract representation of the series is embedded in a vector, and clustering is performed on these 'embedded models' rather than the embedded subsequences. The third method, *clustering by warping costs*, does not transform the time series into another representation, but defines an $n \times n$ symmetric dissimilarity matrix, where n is the number of series. An element (i, j) of the matrix denotes the

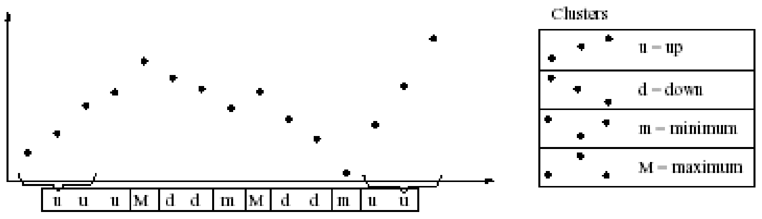


Figure 4.4: Example of partitioning time series using unsupervised approach (Höppner, 2003)

warping cost of warping time series i to series j. Then relational clustering algorithms are applied to cluster the sequences into homogeneous groups. The fourth method, *clustering using Markov models*, is to learn a Hidden Markov model (HMM) or Markov chain for each subsequence and to cluster via the resulting probability models.

As an example, Figure 4.4 illustrates the procedure of segmenting time series into labeled interval sequence using clustering of embedded sequences. Given the series, a window of width 3 is a contiguous subsequence consisting of three consecutive values. By sliding the window along the series, a set of windows (subsequences) of width 3 is formed. This set is then clustered, and the resulting clusters are shown on the right. The labels for each window is given below the series. A labeled interval sequence can be formed by concatenating all consecutive identical labels. It can be seen from the figure that the resulting sequence contains eight intervals.

Sometimes no abstraction step is required because the data is already in interval event form, for example, diseases of a patient, insurance contract, and period in which a certain DNA sequence occurs. This is the type of data ideally stored in temporal databases, which store temporal data, i.e., data that is time-dependent (time varying). Consider medical data consisting of patients' history, in which each patient's history is described by a series of values, such as the body temperature, cholesterol level, blood pressure etc. Each of these values is associated with an interval representing a period of time during which the value holds. Similar examples can be found in many areas that rely on the observation of evolutionary processes, such as environmental studies, economics and many natural sciences. Even though temporal databases support the temporal data mining process, they are not essential for performing temporal data mining (Roddick & Spiliopoulou 2002). To date, algorithms for mining such interval data have been developed around the simpler notion of a flat file data (Kam & Fu 2000, Papapetrou et al. 2005, Winarko & Roddick 2005).

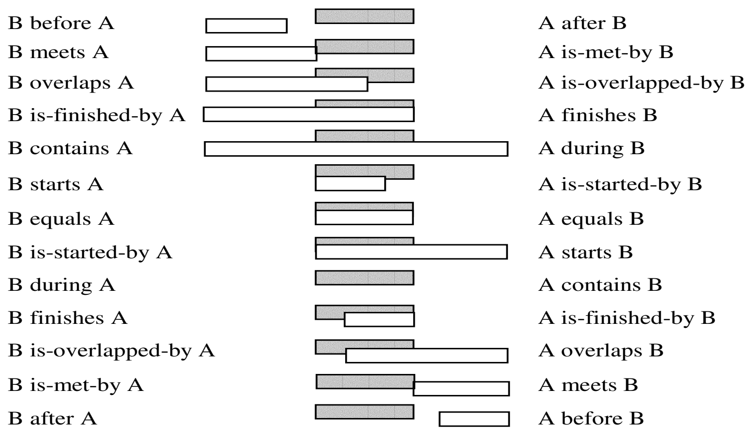
It is also possible to obtain interval data from point-based data, for example, by aggregating similar events to intervals of equal events, and then applying interval event mining to the new data. This approach is taken by Lee *et al.* (2002) who present a new data mining technique to discover temporal rules from interval data originated from point-based data. A pre-processing algorithm is used to summarise data with time points and generalise it into interval data. A temporal relation algorithm is then used to discover temporal rules among transactions from interval data by extending the AprioriAll algorithm (Agrawal & Srikant 1995).

4.2 Time Interval Operators

Several interval operators have been proposed in the literature to describe the relationships among intervals. Allen (1983) introduces temporal relationships between intervals and operators for reasoning about relations between intervals. For any pair of intervals there are 13 possible relationships, as illustrated in Figure 4.5. As shown in the figure, a relationship *B meets A* means that an interval *B* terminates at the same point in time at which an interval *A* starts. Its inverse relationship is *A is-met-by B*.

The Allen relations have been widely used in research in data modelling (e.g. temporal data modelling and temporal databases), temporal or spatial reasoning, as well as data mining. In temporal reasoning, research using Allen's interval logic has been concerned with finding tractable sub-classes of the algebra (Nebel &

Bürckert 1995, Drakengren & Jonsson 1997, Krokhin, Jeavons & Jonsson 2003). In spatial reasoning, they have been extended to describe spatial relations in 2- or 3-dimensional space (Guesgen 1989). In data mining, they have been used for the formulation of temporal patterns involving intervals (see Sections 4.3 and 4.4).



Abbreviations	
b	before
m	meets
o	overlaps
fi	is-finished-by
c	contains
s	starts
e	equals
si	is-started-by
d	during
f	finishes
oi	is-overlapped-by
mi	is-met-by
a	after

Figure 4.5: Allen's interval relationships

Freksa (1992) has generalised Allen's work by introducing *semi-intervals*, with the following 11 operators: *older* (**ol**), *younger* (**yo**), *head to head* (**hh**), *survives* (**sv**), *survived by* (**sb**), *tail to tail* (**tt**), *precedes* (**pr**), *succeeds* (**sd**), *contemporary of* (**ct**), *born before death of* (**bd**), and *died after birth of* (**db**). The power of such a generalisation lies in the fact that if either one or the other is not known some inference is often still possible, sometimes without loss of information. For example, although information may be known about the date of birth or death of a person but not both, this does not prevent some inferences being made regarding events in history.

Freksa has also introduced the notion of 'conceptual neighbourhoods' to accommodate coarse knowledge. Two relations between pairs of events are *conceptual neighbour* if they can be directly transformed into one another by continuously deforming (i.e., shortening, lengthening, or moving) the events. For example, the relations *before* and *meets* are conceptual neighbours since they can be transformed into one another by lengthening one of the events. However, the relations *before* and *overlaps* are not conceptual neighbours since transformation by continuous deformation can only take place via the relation *meets*. A conceptual neighbourhood is a set of relations between pairs of events that are path-connected through conceptual neighbour relations. Based on this concept, Allen's thirteen relations can be arranged according to their conceptual neighbourhood.

Roddick and Mooney (2005) have combined Allen's interval relations with the five point-interval relations of Vilain (1982) considering the relative positions of the interval midpoints. A total of 49 relations is obtained, e.g., nine different versions of overlaps. Some of the different overlaps relations can be interpreted as *small*, *medium*, or *largely* overlapping intervals. The motivation is to handle data with coarse time stamps and data from streams with arbitrary local order. The authors also describe the relation between the models of Allen and Freksa and the respective extension to midpoints and/or intervals of equal durations.

As was mentioned in the previous section, interval data can be obtained by abstracting time series. It is pointed out by Mörchen (2006) that Allen's relations have severe disadvantages when used for pattern discovery from interval data obtained from time series. One of the disadvantages is that symbolic interval sequences obtained from numeric time series inherit the noise existing in the original data. The interval boundaries gained from pre-processing steps like discretisation of values or segmentation are subject to noise in the measurements. Such time points should thus be considered approximate. Using Allen's relations to formulate patterns, such slight variations of interval boundaries can create fragmented patterns that describe the same intuitive relationship between intervals with different operators. This is illustrated in Figure 4.6 in which three pairs of almost equal intervals have different relationships. Any pattern using one of Allen's relations that requires equality of two interval boundaries can be altered by changing one boundary by a small time unit only. To overcome this drawback, one approach is to relax the strictness of Allen's relations by using a threshold to consider temporally close interval boundaries equal (Aiello, Monz, Todoran &

Worring 2002).

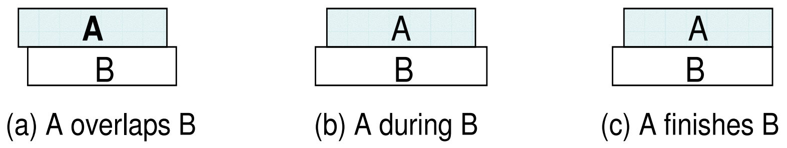


Figure 4.6: Three relationships resulting from two almost equal intervals

In another approach, Ultsch (2004) defines the Unification-based Temporal Grammar (UTG) that contains an approximate version of Allen's *equals* operator called *more or less simultaneous.* The main elements of the UTG are Events and Sequences. The Events combine several *more or less simultaneous* intervals, a robust version of Allen's *equals.* The start and end points of the intervals are not required to be exactly equal; they only need to be within a small time interval. The number of intervals in an Event is restricted to the dimensionality of the interval series. The Sequences describe an ordering of several Events with *immediately followed by* or *followed by after at most t time units.* A further generalisation, called *coincides*, is proposed by dropping the constraint on the boundary points, only requiring some overlap between the intervals (Mörchen & Ultsch 2004). The concept of coincidence describes the intersection of several intervals. This is equivalent to the disjunction of Allen's *overlaps*, *starts*, *during*, *finishes*, the four corresponding inverses, and *equals.* The *coincides* operator represents this temporal concept.

Mörchen (2006) proposes the TSKR (Time Series Knowledge Representation), which extends the UTG by allowing an arbitrary number of coinciding parts of intervals in contrast to the fixed number of complete intervals within Events, and by relaxing the total order in Sequences to a partial order. The TSKR is a hierarchical language describing the temporal concepts of duration, coincidence, and partial order in an interval time series. Its basic primitives are labeled interval called *Tones*, representing duration. Simultaneously occurring Tones form *Chord*, representing coincidence. Several Chords connected with a partial order form a *Phrase.*

4.3 Mining Patterns from a Long Sequence of Intervals

Only a few previous studies consider the discovery of temporal patterns from a long sequence of intervals, which might originate from multivariate time series. Villafane *et al.* (2000) propose a technique to discover containment relationships from interval time series. A containment lattice is constructed from the intervals, and then the containment relationships are discovered with the *Growing Snake Traversal* method. Hoppner (2001) mines temporal rules expressed with Allen's interval logic and a sliding window to restrict the pattern length. The patterns are mined with an Apriori-like algorithm. In other work, temporal patterns are extracted from a symbolic interval sequence obtained from multivariate time series and formulated using the UTG (Guimarães & Ultsch 1999, Mörchen et al. 2004, Mörchen & Ultsch 2004). In Mörchen (2006) the patterns are expressed using the TSKR.

This section describes in more detail the temporal pattern discovery framework proposed by Höppner (2001), which generalises the discovery of episodes in event sequences (Mannila et al. 1997) to interval sequences.

Let S denote the set of all possible states (labels). A *state interval* (b, s, f) denotes a state $s \in S$ holds during a period of time $[b, f)$[1], where b is the *start-time* and f is the *end-time*, when the state no longer holds. A state sequence on S is a series of triples defining state intervals

$$(b_1, s_1, f_1), (b_2, s_2, f_2), (b_3, s_3, f_3), (b_4, s_4, f_4), \cdots$$

where $b_i \leq b_{i+1}$ and $b_i < f_i$ hold.

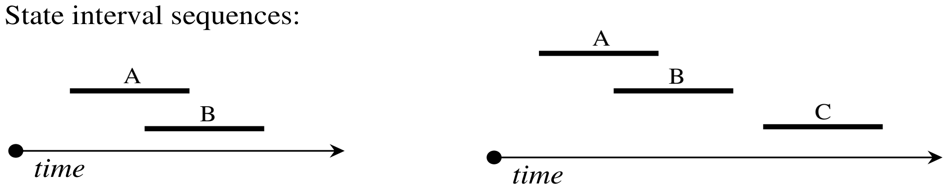
This model uses Allen's temporal interval logic to describe the relationships between state intervals. Given n state intervals (b_i, s_i, f_i), $1 \leq i \leq n$, a *temporal pattern* of size n is a pair (s, M), where $s\colon \{1, \ldots, n\} \to S$ maps index i to the

[1]It is assumed that the begin time is inclusive but the end time is not.

corresponding state, and M is an $n \times n$ matrix whose elements $M[i, j]$ denote the relationship between intervals $[b_i, f_i)$ and $[b_j, f_j)$. The *size* of a temporal pattern α is the number of intervals in α, denoted by $dim(\alpha)$. If the size of α is n, then α is called a n-pattern.

Many state sequences map to the same temporal pattern. These sequences are called *instances* of the pattern. A temporal pattern has no specific temporal extensions because it has been abstracted from the time intervals given in a specific sequence. However, the pattern instances themselves do have a temporal extension. Figure 4.7 shows two temporal patterns with their corresponding state sequences. As depicted in the figure, the value of $M[i, j]$ is always the inverse of that of $M[j, i]$.

State interval sequences:



Temporal patterns:

$$\begin{array}{c|cc} & A & B \\ \hline A & = & o \\ B & oi & = \end{array} \qquad \begin{array}{c|ccc} & A & B & C \\ \hline A & = & o & b \\ B & oi & = & b \\ C & a & a & = \end{array}$$

Figure 4.7: Example of temporal patterns

A temporal pattern $\alpha = (s_\alpha, M_\alpha)$ is a *subpattern* of $\beta = (s_\beta, M_\beta)$, denoted by $\alpha \sqsubseteq \beta$, if $\dim(s_\alpha, M_\alpha) \leq \dim(s_\beta, M_\beta)$ and there is an embedding (injective mapping) $\pi \colon \{1, \ldots, \dim(s_\alpha, M_\alpha)\} \rightarrow \{1, \ldots, \dim(s_\beta, M_\beta)\}$ such that

$$\forall i, j \in \{1, \ldots, \dim(s_\alpha, M_\alpha)\}\colon$$
$$s_\alpha(i) = s_\beta(\pi(i)) \wedge M_\alpha[i, j] = M_\beta[\pi(i), \pi(j)]$$

In simple terms, a temporal pattern α is a subpattern of β if α can be obtained by removing some states (and the corresponding relationships) from β. Consider the 2-pattern and 3-pattern in the Figure 4.7. The first pattern is a subpattern of the second because the first pattern can be obtained by removing the state C and its corresponding relationships (with A and B) from the second.

To simplify the pattern notation, the model uses the so called *normalized temporal patterns*. The basic idea is to order the state intervals in time with increasing index. Given a state sequence, sorting the state intervals in the sequence lexicographically (by the start time, end time, and state) results in a normalized pattern. Let (b_i, s_i, f_i) and (b_j, s_j, f_j) be any two state intervals in the pattern instance. If the start times of both state intervals are different, the ordering is based on the start times, and the resulting relation between (b_i, f_i) and (b_j, f_j) is *before, meets, overlaps, is-finished-by*, or *contains* (See Figure 4.5). If the start times of both intervals are equal and the end times are different, the ordering is based on the end times, and the possible relation is *starts*. If (b_i, f_i) and (b_j, f_j) are identical, the ordering is based on the states (alphabetically, $s_i < s_j$ or $s_j < s_i$), and the relation is *equals*. Both temporal patterns in Figure 4.7 are already in the normalized form.

As in the discovery of episodes (see Chapter 2.3.2), given a long sequence of intervals, a sliding window is used to restrict the pattern length. The patterns are considered interesting if they can be observed within this window. The *support* of a pattern α, denoted by $sup(\alpha)$, is defined as the total time in which the pattern can be observed within the sliding window. A pattern is called *frequent* if its support exceeds a threshold *minsup*.

The task is to discover all frequent (normalized) temporal patterns from a single sequence of state intervals. The algorithm for the discovery of temporal patterns is based on the Apriori algorithm (Agrawal & Srikant 1994), extended to deal with a sequence of intervals. The algorithm requires several passes over the input sequence. The first pass over the sequence counts the support of every single state (also called candidate 1-patterns), and generates a set of frequent 1-patterns. This set is then used to create candidate 2-patterns, and by determining their supports (using the second pass over the sequence) the frequent 2-patterns are found. Analogously, these frequent 2-patterns are used to first obtain candidate 3-patterns and then using the third pass the frequent 3-patterns are found, and so on. This procedure is repeated until no more frequent patterns can be found.

To generate a candidate $(k+1)$-patterns, the algorithm joins any two frequent k-patterns that share a common $(k-1)$-pattern as prefix. As an example, the two 2-patterns *(A meets B)* and *(A meets C)* share the primitive 1-pattern A as a common prefix. Therefore, both can be joined to generate candidate 3-patterns. To obtain a candidate 3-pattern, the missing relationship between B and C has to be determined. The law of transitivity for interval relations (Allen 1983) shows that the possible set of interval relations is {*is-started-by, equals, starts*}. In normalized form, only two out of these three possible relationships remain, that is, {*equals, starts*}. In addition to the pruning method based on the law of transitivity, the pruning technique that is used for the discovery of association rules (Agrawal & Srikant 1994) can still be applied to temporal patterns, because the property that every k-subpattern of a $(k+1)$-candidate must be frequent also holds in this case.

After all frequent temporal patterns have been found, the rule of the form $X \rightarrow Y$ can be constructed from every pair frequent patterns X and Y, with $X \sqsubseteq Y$. The confidence of the rule is $conf(X \rightarrow Y) = \frac{sup(Y)}{sup(X)}$.

4.4 Mining Patterns from Interval Sequence Databases

This section discusses the discovery of temporal patterns from interval sequence databases proposed by Kam and Fu (2000) and Papapetrou *et al.* (2005).

Kam and Fu (2001) have proposed *A1 temporal patterns* to represent relationships between intervals found in a set of interval sequences. Let $\mathcal{E} = \langle E_1, E_2, \ldots, E_n \rangle$ be an ordered set of event intervals, called *event interval sequence* (interval sequence for short). Each event interval E_i is a triple (b_i, e_i, f_i), where e_i is an event type (label), b_i is the event start time, and f_i is the end time. This work requires that intervals in the interval sequence are ordered by the end times. As an example, an interval sequence consisting of five intervals in Figure 4.8 is written as $\mathcal{E} = \langle(1, A, 8), (8, C, 15), (3, B, 19), (23, C, 37)\rangle$. A database $D = \{\mathcal{E}_1, \mathcal{E}_2, \cdots, \mathcal{E}_k\}$ is a set of interval sequences.

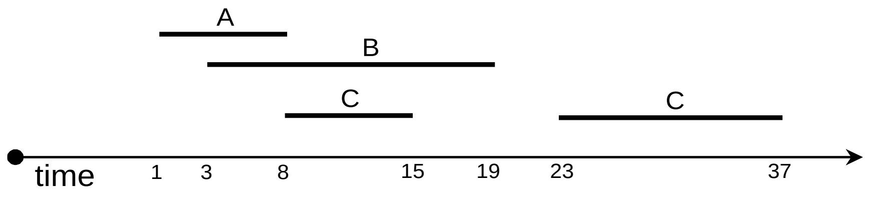


Figure 4.8: Example of an interval sequence

Given an interval sequence, A1 temporal patterns are formulated using Allen's interval relations and have the following form:

$$(\ldots((A_1\ rel_1\ A_2)\ rel_2\ A_3)\ldots rel_{k-1}\ A_k),$$

where A_i is an event type and rel_i is a temporal relation between intervals associated with A_i and A_{i+1}. Since the sequence is ordered by the end times, the possible set of relations is {*before, equals, meets, overlaps, during, starts, finishes*}. The *size* of a pattern is the number of events in the pattern. A pattern with the size k is called the k-pattern. A1 patterns only allow the concatenation of temporal relations on the right hand side. For example, for the sequence shown in Figure 4.8, the resulting A1 pattern is '*(((A meets C) overlaps B) before C)*'

The support of an A1 pattern α is the fraction of all interval sequences in the database D that contain α. An interval sequence *contains* a pattern α if all the events in α also appear in the sequence with the same relations between them, as defined in α. Given a minimum support *minsup*, a pattern α is frequent in D if its support is greater than or equal to *minsup*. The task is to find all frequent A1 patterns.

The algorithm to discover frequent A1 patterns is based on the Apriori algo-

Seq-id	Interval sequence
1	(5, A, 10), (8, B, 12), (16, C, 18), (14, D, 20), (17, B, 22)
2	(8, A, 14), (9, B, 15), (19, C, 21), (16, D, 22)
3	(12, A, 20), (22, B, 25), (29, C, 31), (28, D, 32)

(a) Example of interval sequence database

A	B	C	D
(1, 5, 10) (2, 8, 14) (3, 12, 20)	(1, 8, 12) (1, 17, 22) (2, 9, 15) (3, 22, 25)	(1, 16, 18) (2, 19, 21) (3, 29, 31)	(1, 14, 20) (2, 16, 22) (3, 28, 32)

(b) The database represented as item_list

A overlaps B	A before B	C during D
(1, 5, 12) (2, 8, 15)	(1, 5, 22) (3, 12, 25)	(1, 14, 20) (2, 16, 22) (3, 28, 32)

(c) Part of frequent 2-pattern item_list

Figure 4.9: An interval sequence database and fragment of mining process

rithm (Agrawal & Srikant 1994). However, instead of using the horizontal format, the algorithm transforms the database into the vertical data format similar to the one used in the SPADE algorithm (Zaki 2001) (See Chapter 2.2.4). In this vertical data format, each event type is associated with a list of triples containing sequence-id (*sid*), start time, and end time. In this work, such a list is called an *item_list*. Figure 4.9(a) shows an example of a database containing four interval sequences, each being ordered by the end times. In Figure 4.9(b), the database has been transformed into vertical data format containing the *item_lists* of items in the database.

The proposed algorithm consists of several iterations. Each iteration, say iteration k, starts with a seed set of frequent patterns found in the previous iteration, F_{k-1}. This set is used for generating a set of candidate patterns, by adding one atomic event in F_1 to patterns in the seed set F_{k-1}. The *item_list* of a candidate k-pattern is determined by merging (union) the *item_lists* of its generating patterns (in F_1 and F_{k-1}). Similar to the SPADE algorithm, the use of vertical data format allows the algorithm to count the support of a candidate pattern by counting the number of distinct sequence-id in its *item_list*. The candidates that have sufficient support become the frequent k-patterns. Figure 4.9(c) shows the *item_lists* of three frequent patterns of size 2 (when the *minsup* = 50%). The algorithm terminates when it cannot find any frequent k-patterns at the end of the current pass.

Papapetrou *et al.* (2005) studied the discovery of frequent *arrangements* from interval sequence databases described above. In contrast to the previous model, their framework requires that the intervals in a sequence are ordered by the start times. For example, a sequence in Figure 4.8 is represented as $\mathcal{E} = \langle(1, A, 8), (3, B, 19), (8, C, 15), (23, C, 37)\rangle$. Moreover, this work only considers five types of temporal relations between two intervals: *meets, matches, overlaps, contains*, and *follows* as shown in Figure 4.10. The notation *follow(A,B)* in the figure is read as *B follows A*.

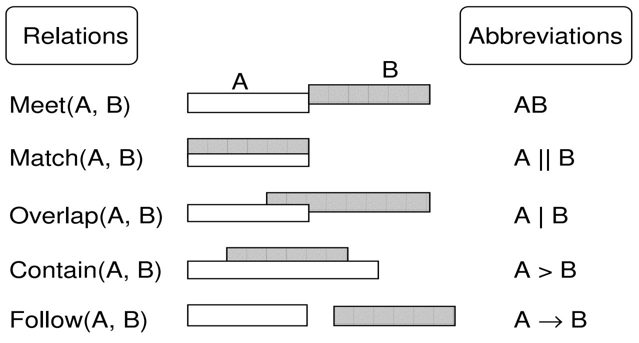


Figure 4.10: Five relations used in arrangements

An *arrangement* $\mathcal{A}$ is defined as $\mathcal{A} = \{\mathcal{E},\ R\}$, where $\mathcal{E}$ is a set of n event intervals that occur in $\mathcal{A}$, and $R = \{r(E_i, E_j) \mid \forall i < j,\ i = 1, \ldots, (n-1),\ j = 2, \ldots, n\}$. Each $r(E_i, E_j) \in R$ defines a temporal relation between E_i and E_j, where $r \in \{$*meets, matches, overlaps, contains, follows*$\}$. The size of an arrangement $\mathcal{A}$ is equal to $|\mathcal{E}| = n$. An arrangement of size n is called n-arrangement. For example, consider two arrangements shown in Figure 4.11. The first arrangement on the left has $R = \{$*overlap(A,B), follow(A,C), follow(B,C)*$\}$, while the second has $R = \{$*overlap(A,B), follow(A,C), overlap(B,C)*$\}$.

Given an interval sequence, the sequence *contains* an arrangement $\mathcal{A} = \{\mathcal{E}, R\}$, if all events in $\mathcal{A}$ also appear in the sequence with the same relation between them, as defined in R. Based on this definition, an interval sequence shown in Figure 4.8 contains an arrangement in Figure 4.11(a), but it does not contain the one in Figure 4.11(b). The *support* of an arrangement is the fraction of all sequences in the database that contain the arrangement.

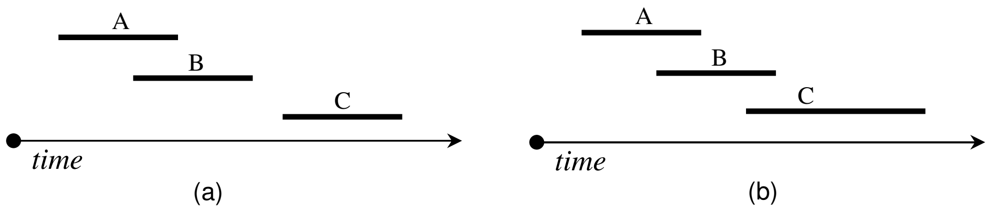


Figure 4.11: Two arrangements of size 3

Given a minimum support threshold *minsup*, an arrangement is *frequent* in the sequence database D if its support exceeds *minsup*. The mining problem is to discover all frequent arrangements in the interval sequence database D.

The authors propose two algorithms for discovering frequent arrangements. The algorithms use a tree-based enumeration structure, called an *arrangement enumeration tree*, similar to the one introduced by Bayardo (1998). The breadth first search (BFS) based approach uses an arrangement enumeration tree to discover the set of frequent arrangements. The depth first search (DFS) based method further improves the performance of BFS by reaching longer arrangements faster and eliminating the need for examining smaller subsets of these arrangements.

4.5 Summary

This chapter has provided a survey of previous studies on the discovery of temporal patterns from interval data. Despite its importance, this area has not received enough attention from researchers. Essentially, there are two types of interval data model on which the mining methods can be applied, namely a long interval sequence, and a set of short interval sequences. While most previous pattern discovery methods use Allen's temporal interval relations, its subset or its extensions, others employ other interval operators such as the UTG and the TSKR.

This thesis contributes to this area by proposing the discovery of richer temporal association rules from interval sequence databases (Winarko & Roddick 2005, Winarko & Roddick 2007). The richer temporal association rules are formulated using the pattern formulation proposed by Höppner (2001). The algorithm called ARMADA is proposed to discover the rules. The discovery of richer temporal association rules is presented in the next chapter (Chapter 5).

Chapter 5

ARMADA: Mining Richer Temporal Association Rules

This chapter considers the discovery of richer temporal association rules from a database that contains a set of interval sequences. Discovering temporal rules from interval-based sequential data is certainly more complex and requires a different approach from mining patterns from point-based sequential data, such as mining sequential patterns or episodes. An interval has duration and therefore the generated patterns have different semantics than simply *before* and *after*.

In this chapter, a new algorithm called ARMADA is proposed for discovering richer temporal association rules. ARMADA first generates temporal patterns by extending the MEMISP algorithm (Lin & Lee 2002) for mining sequential patterns (see Chapter 2.2.6). After the temporal patterns have been found, they are then can be used to generate temporal rules called *richer temporal association rules*. This work is different from the work of Höppner (2001), which discovers temporal rules from a long sequence of intervals rather than from a set of interval sequences.

The chapter is organised as follows. Section 5.1 defines the problem of discovering richer temporal association rules. Section 5.2 describes ARMADA for discovering the rules from interval sequence databases. Section 5.3 discusses the *maximum gap* time constraint. Section 5.4 evaluates the algorithm by running several sets of experiments.

5.1 Problem Statement

This section defines the problem of discovering richer temporal association rules from interval sequence databases. In order to make this chapter self-contained, the definition of state sequences and (normalized) temporal patterns already mentioned in Chapter 4.3 is repeated here.

Definition 5.1 (State sequence) Let S denote the set of all possible states. A state $s \in S$ that holds during a period of time $[b, f)$ is denoted as (b, s, f), where b is the *start-time* and f is the *end-time*. The (b, s, f) is called a *state interval*. A *state sequence* on S is a series of triples defining state intervals $\langle(b_1, s_1, f_1), (b_2, s_2, f_2), \ldots, (b_n, s_n, f_n)\rangle$, where $b_i \leq b_{i+1}$ and $b_i < f_i$.

Definition 5.2 (Temporal pattern) Given n state intervals (b_i, s_i, f_i), $1 \leq i \leq n$, a *temporal pattern* of size n is defined by a pair (s, M), where $s : \{1, \ldots, n\} \rightarrow S$ maps index i to the corresponding state, and M is an $n \times n$ matrix whose elements $M[i, j]$ denote the relationship between intervals $[b_i, f_i)$ and $[b_j, f_j)$. The *size* of a temporal pattern α is the number of intervals in the pattern, denoted as $\dim(\alpha)$. If the size of α is n, then α is called a n-pattern.

As for Höppner (2001), the model proposed in this chapter also uses *normalized* temporal patterns. Given a state sequence, a normalized temporal pattern can be created by sorting the state intervals in the sequence lexicographically (by the start time, end time, and state). As was mentioned in the previous chapter, normalized temporal patterns only require seven relations out of thirteen Allen's relations, namely, *before (b), meets (m), overlaps (o), is-finished-by (fi), contains (c), equals (=)*, and *starts (s)*, as shown in Figure 5.1. The first five relationships are when the start times differ. In this case, the ordering is based on the start times. If both intervals are identical, the ordering is based on the states, which results in *A equals B*, instead of *B equals A*. If the start times are the same and the end times are different, the ordering is based on the end times.

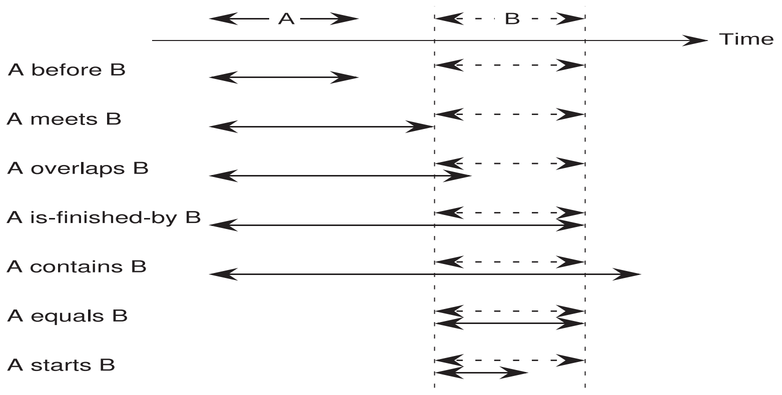


Figure 5.1: Seven relations used in normalized temporal patterns

As an example, Figure 5.2 shows three normalized temporal patterns, each with its corresponding instance of pattern. In the remainder of this thesis, for brevity, labels on the rows of the matrix are not shown when writing a normalized temporal pattern, because they are always similar to the column labels. Besides that, the value of elements under the diagonal elements of the matrix is replaced with a symbol '*' as in the normalized temporal pattern these values are not considered.

Definition 5.3 (Subpattern) A temporal pattern $\alpha = (s_\alpha, M_\alpha)$ is a subpattern of $\beta = (s_\beta, M_\beta)$, denoted $\alpha \sqsubseteq \beta$, if $\dim(s_\alpha, M_\alpha) \leq \dim(s_\beta, M_\beta)$ and there is an injective mapping $\pi : \{1,\ldots,\dim(s_\alpha, M_\alpha)\} \rightarrow \{1,\ldots,\dim(s_\beta, M_\beta)\}$ such that

$$\forall i, j \in \{1,\ldots,\dim(s_\alpha, M_\alpha)\}:$$
$$s_\alpha(i) = s_\beta(\pi(i)) \wedge M_\alpha[i,j] = M_\beta[\pi(i), \pi(j)]$$

Informally, a temporal pattern α is a subpattern of β if α can be obtained by removing some states (and the corresponding relationships) from β. Consider the pattern in Figure 5.2. A pattern α_1 is a subpattern of α_2, but it is not a subpattern of α_3. The pattern α_1 can be obtained by removing a state D and its

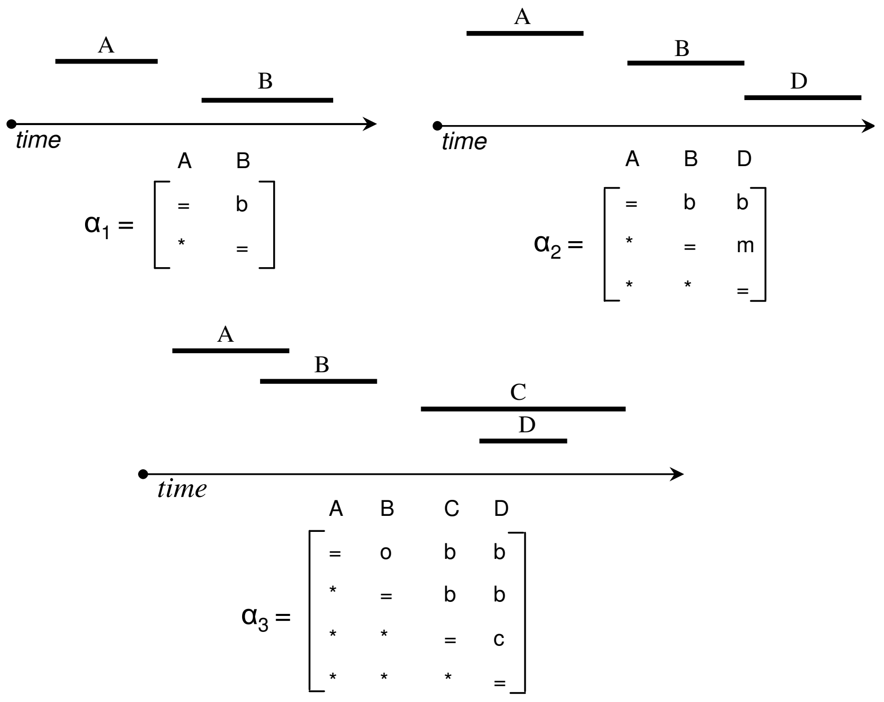

A
B
time
A B
α1 = [= b; * =]
A
B
D
time
A B D
α2 = [= b b; * = m; * * =]
A
B
C
D
time
A B C D
α3 = [= o b b; * = b b; * * = c; * * * =]

Figure 5.2: Three normalized temporal patterns

corresponding relationships from α_2, on the other hand, removing states C and D, and their corresponding relationships from α_3 would not result in α_1.

Definition 5.4 (Database) Given a temporal database $D = \{t_1 \ldots t_n\}$, each record t_i consists of a *client-id*, a *temporal attribute*, a *start-time*, and an *end-time*, where *start-time* < *end-time*. It is assumed that the interval between the *start-time* and *end-time*, indicating the interval during which the record values are valid, is a relatively short interval (as compared to the total period under analysis). Each *client-id* can be associated with more than one record.

In most databases, several temporal attributes can be recorded. Each of these attributes represents a different temporal dimension of the data. For example, in a medical database the date of birth of a patient, the dates of medical examinations, the dates of important medical incidents and other dates concerning different facts of the evolution of the health of a patient can be recorded (Koundourakis & Theodoulidis 2002). In these cases, one or more temporal attributes can be picked as the target of the mining process.

If all records in the database D with the same *client-id* are grouped together and ordered by increasing *start-time*, the records associated with a single client can be regarded as a state sequence and D would have one such sequence corresponding to each client. Each state sequence is called the client state sequence (or *client sequence* for short). As a result, the database D can be viewed as a set of client sequences.

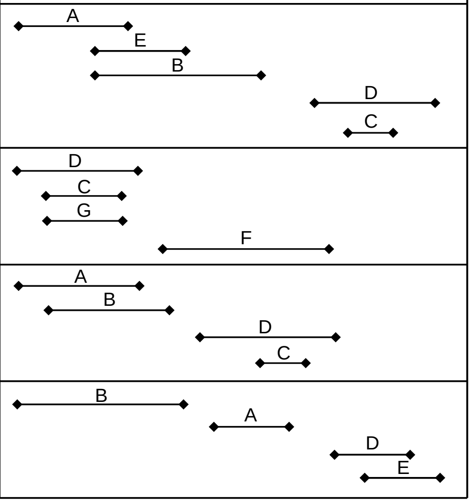
As an example, consider a temporal database D shown in Figure 5.3, which stores a list of clinical records. Each record contains a *patient-id*, a *disease-code* and a pair of ordered time points, indicating the period during which the patient exhibited a given disease. The last column in the table visualises the relative position of intervals associated with the diseases. In the different format, the database contains four client sequences (one for each *patient-id*), namely d_1, d_2, d_3, and d_4. The state intervals in each client sequence have been sorted on the *start-time*, the *end-time*, and the *disease-code.*

The definition of support is different from the one defined in Höppner (2001). In this work, the support for a temporal pattern is defined as the fraction of client sequences in the database which contain the pattern, not as the number of windows in which the pattern can be observed.

Definition 5.5 (Support) Given a database D, a client sequence $d \in D$ *contains* a pattern $\alpha = (s_\alpha, M_\alpha)$ if $(s_\alpha, M_\alpha) \sqsubseteq (s_d, M_d)$, where (s_d, M_d) is a pattern that represent the relationships between intervals in the client sequence. The *support* of a pattern α is defined as $sup(\alpha) = \frac{|D_\alpha|}{|D|}$, where $|D_\alpha|$ is the number of client sequences that contain the pattern α, and $|D|$ is the number of client sequences in the database D.

Definition 5.6 (Frequent temporal pattern) Given a minimum support *minsup*, a pattern is called *frequent* if its support is greater than or equal to *minsup.*

For example, a temporal pattern *(A overlaps B)* is contained in client sequences d_1 and d_3, but *(A before B)* is not. Using a *minsup* of 40%, the set of all

Patientid	Disease code	Start time	End time	Relative position of intervals in each sequence
1	A	2	7	 A E B D C D C G F A B D C B A D E
1	E	5	10	
1	B	5	12	
1	D	16	22	
1	C	18	20	
2	D	8	14	
2	C	10	13	
2	G	10	13	
2	F	15	22	
3	A	6	12	
3	B	7	13	
3	D	14	20	
3	C	17	19	
4	B	8	16	
4	A	18	21	
4	D	24	27	
4	E	25	28	

Sid	Client state sequence
d_1	(2, A, 7), (5, E, 10), (5, B, 12), (16, D, 22), (18, C, 20)
d_2	(8, D, 14), (10, C, 13), (10, G, 13), (15, F, 22)
d_3	(6, A, 12), (7, B, 13), (14, D, 20), (17, C, 19)
d_4	(8, B, 16), (18, A, 21), (24, D, 27), (25, E, 28)

Figure 5.3: Example database consisting of clinical records

frequent temporal patterns are shown in Table 5.1.

Definition 5.7 (Richer temporal association rule) A richer temporal association rule is an expression $\beta \rightarrow \alpha$, where α and β are frequent temporal patterns such that $\beta \sqsubset \alpha$. The confidence of the rule $\beta \rightarrow \alpha$ is defined as

$$conf(\beta \rightarrow \alpha) = \frac{sup(\alpha)}{sup(\beta)}$$

Consider frequent patterns $\beta = \begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \left[\begin{array}{cc} = & o \\ * & = \end{array}\right] \end{array}$ and $\alpha = \begin{array}{c} \begin{array}{ccc} A & B & D \end{array} \\ \left[\begin{array}{ccc} = & o & b \\ * & = & b \\ * & * & = \end{array}\right] \end{array}$, where $\beta \sqsubset \alpha$ and each has the support of 50% (see Table 5.1). A rule $\beta \rightarrow \alpha$ can be generated with the confidence of 100%. The rule can be interpreted as *if A overlaps B*

Table 5.1: A set of frequent temporal patterns

1-patterns	$\langle A\rangle$ $(sup = 0.75)$, $\langle B\rangle$ $(sup = 0.75)$, $\langle C\rangle$ $(sup = 0.75)$, $\langle D\rangle$ $(sup = 1.00)$, $\langle E\rangle$ $(sup = 0.50)$
2-patterns	$\begin{array}{cc} A & B \\ = & o \\ * & = \end{array}$ $(sup = 0.50)$, $\begin{array}{cc} A & D \\ = & b \\ * & = \end{array}$ $(sup = 0.75)$, $\begin{array}{cc} A & C \\ = & b \\ * & = \end{array}$ $(sup = 0.50)$, $\begin{array}{cc} B & D \\ = & b \\ * & = \end{array}$ $(sup = 0.75)$, $\begin{array}{cc} B & C \\ = & b \\ * & = \end{array}$ $(sup = 0.50)$,
	$\begin{array}{cc} D & C \\ = & c \\ * & = \end{array}$ $(sup = 0.75)$
3-patterns	$\begin{array}{ccc} A & B & D \\ = & o & b \\ * & = & b \\ * & * & = \end{array}$ $(sup = 0.50)$, $\begin{array}{ccc} A & B & C \\ = & o & b \\ * & = & b \\ * & * & = \end{array}$ $(sup = 0.50)$, $\begin{array}{ccc} B & D & C \\ = & b & b \\ * & = & c \\ * & * & = \end{array}$ $(sup = 0.50)$, $\begin{array}{ccc} A & D & C \\ = & b & b \\ * & = & c \\ * & * & = \end{array}$ $(sup = 0.50)$
4-patterns	$\begin{array}{cccc} A & B & D & C \\ = & o & b & b \\ * & = & b & b \\ * & * & = & c \\ * & * & * & = \end{array}$ $(sup = 0.50)$

occurs, then it is highly likely that D will also occur after A and B.

Given a database D, the problem of mining richer temporal association rules is to generate all richer temporal association rules that have confidence greater than or equal to the user-specified minimum confidence *minconf*.

5.2 ARMADA - Mining Richer Temporal Association Rules

As in the mining of association rules (Agrawal & Srikant 1994), the mining of richer temporal association rules can be decomposed into two subproblems, first, to find all frequent temporal patterns that have support above the minimum support and, second, to generate the rules from the frequent patterns. Section 5.2.1 describes ARMADA for discovering frequent temporal patterns, assuming that the database fits into memory. Section 5.2.2 outlines the method for discovering frequent temporal patterns from large databases that do not fit in memory. Then, the method to generate the rules is given in Section 5.2.3.

5.2.1 Discovering Frequent Temporal Patterns

ARMADA discovers frequent temporal patterns in three steps. First, the algorithm reads the database into memory. While reading the database, it counts the support of each state and generates frequent 1-patterns. The algorithm then constructs an index set for each frequent 1-pattern and finds frequent patterns using the state sequences indicated by elements of the index set. Finally, using a recursive *find-then-index* strategy, the algorithm discovers all temporal patterns from the in-memory database. Each of these steps is described below. As an illustration, the algorithm is used to discover frequent patterns from an example database shown in Figure 5.3 and a minimum support *minsup* of 40%. Pseudo code of ARMADA is shown in Algorithm 5.1.

Definition 5.8 (Prefix pattern) Given a pattern ρ, where $dim(\rho) = n$, and a frequent state s in the database, a pattern ρ' of size $(n + 1)$ can be formed by adding the s as a new element to ρ and setting the relationships between s and each element of ρ. The frequent state s is called *stem* of the pattern ρ' and ρ is the *prefix pattern* (*prefix* for short) of ρ'.

```
Input: : a temporal database $D$, $minsup$
Output: : all frequent normalized temporal patterns
1: read $D$ into $MDB$ (in-memory database) to find all frequent states
2: for each frequent state $s$ do
3:     form a pattern $\rho = \langle s \rangle$, output $\rho$
4:     construct $\rho$-idx = $CreateIndexSet(s, \langle\ \rangle, MDB)$
5:     call $MineIndexSet(\rho, \rho\text{-}idx)$
6: end for
```

Algorithm 5.1: Pseudo code of ARMADA

Step 1 - Reading the Database into Memory

In this first step, the algorithm reads the database D into memory, which will be referred to as *MDB* hereafter. While reading each client sequence from the database, the algorithm computes the support count of every state, then finds the set of all frequent states. From the example database, the algorithm finds frequent states A, B, C, D, and E. The state A is supported by 3 client sequences, i.e., the client sequences d_1, d_3, and d_4. Each of these states will become a frequent patterns of size 1 (see Table 5.1).

Similar to the sequential patterns, the set of all frequent patterns can be grouped into several groups such that the patterns within a group share the same prefix. For example, from the set of frequent 1-patterns found in this step, the set of all frequent patterns can be grouped into five groups according to the five prefix patterns: $\langle A \rangle$, $\langle B \rangle$, $\langle C \rangle$, $\langle D \rangle$, and $\langle E \rangle$. Each group of frequent patterns then can be discovered by constructing corresponding index set and mining each recursively, as shown in the following steps.

Step 2 - Constructing the Index Set

Let ρ' be a pattern formed by combining a prefix pattern ρ and a stem s. An index set ρ'*-idx* is a collection of client sequences that contain a pattern ρ'. Each element of the index set contains three fields, namely, *ptr_cs*, *a_intv*, and *pos*. The *ptr_cs* is a pointer to the client sequence, *a_intv* is a list of intervals in the client sequence which produce a pattern ρ', and *pos* is the first occurring position of s in the client sequence with respect to ρ. The pseudo code for constructing the index set is shown in Algorithm 5.2. The third parameter in the algorithm, *range-set*, is a set of client sequences for indexing, whose value is either *MDB* or an index set.

```
1: // Construct the index set $\rho'$-idx
2: // $\rho'$ is a pattern formed by combining $\rho$ and $s$
3: // range-set is a set of client sequences for indexing
4: CreateIndexSet(s, $\rho$, range-set):
5: for each client sequence d in range-set do
6:   if range-set = MDB then
7:     start-pos = 0
8:   else
9:     start-pos = pos
10:  end if
11:  for pos = (start-pos+1) to |d| do
12:    if stem state s is first found at position pos in d then
13:      insert (ptr_cs, a_intv, pos) to the index set $\rho'$-idx, where ptr_cs points to d
14:    end if
15:  end for
16: end for
17: return index set $\rho'$-idx
```

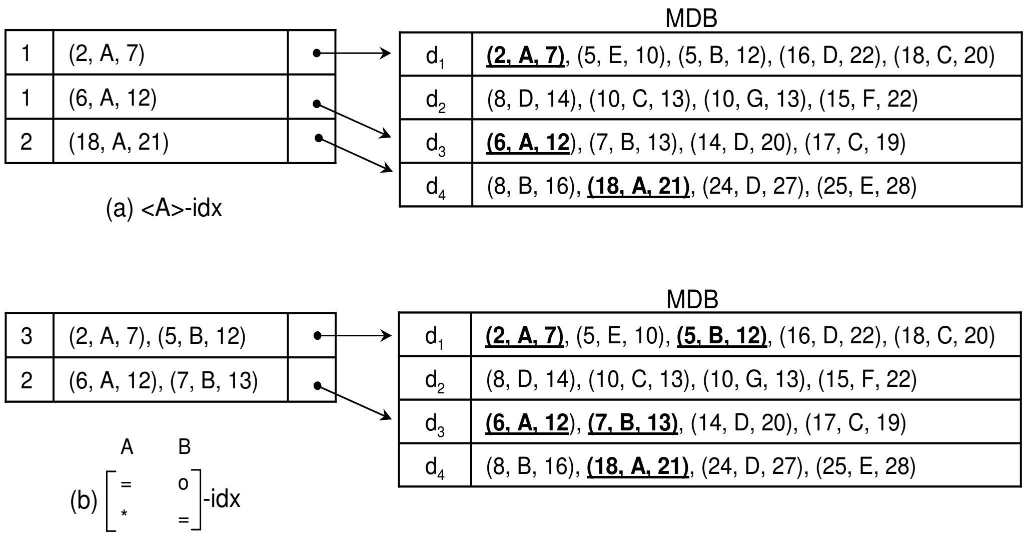
Algorithm 5.2: Pseudo code for constructing an index set

Step 3 - Mining Patterns from the Index Set

Given an index set ρ*-idx*, the goal of mining the index set ρ*-idx* is to find stems with respect to the prefix ρ. Any state in the indexed client sequences whose position is larger than the value of *pos* could be a potential stem (with respect to ρ). Thus, for every client sequences in ρ*-idx*, the algorithm increases the support count of such state by one. Afterward, the algorithm determines which of the states are frequent and become stems. Each of these stems will be combined with the prefix ρ to generate a frequent pattern ρ'. Then, recursively, the index set ρ'-*idx* is constructed and mined until no more stem can be found. The pseudo code for mining an index set is shown in Algorithm 5.3.

```
1: // Mine patterns from an index set ρ-idx
2: MineIndexSet(ρ, ρ-idx):
3: for each d pointed by index elements of ρ-idx do
4:    for pos = pos + 1 to |d| in d do
5:       count(s) = count(s) + 1, where s is a potential stem state
6:    end for
7: end for
8: find S = the set of stems s
9: for each stem state s ∈ S do
10:    output the pattern ρ' by combining prefix ρ and stem s
11:    call CreateIndexSet(s, ρ, ρ-idx) // to construct the index set ρ'-idx
12:    call MineIndexSet(ρ', ρ'-idx) //to mine patterns with index set ρ'-idx
13: end for
```

Algorithm 5.3: Pseudo code for mining an index set



1	(2, A, 7)	•
1	(6, A, 12)	•
2	(18, A, 21)	•

(a) <A>-idx

MDB

d_1	**(2, A, 7)**, (5, E, 10), (5, B, 12), (16, D, 22), (18, C, 20)
d_2	(8, D, 14), (10, C, 13), (10, G, 13), (15, F, 22)
d_3	**(6, A, 12)**, (7, B, 13), (14, D, 20), (17, C, 19)
d_4	(8, B, 16), **(18, A, 21)**, (24, D, 27), (25, E, 28)

3	(2, A, 7), (5, B, 12)	•
2	(6, A, 12), (7, B, 13)	•

(b) $\begin{array}{cc} A & B \\ \left[\begin{array}{c} = \\ * \end{array}\right. & \left.\begin{array}{c} o \\ = \end{array}\right] \end{array}$-idx

MDB

d_1	**(2, A, 7)**, (5, E, 10), **(5, B, 12)**, (16, D, 22), (18, C, 20)
d_2	(8, D, 14), (10, C, 13), (10, G, 13), (15, F, 22)
d_3	**(6, A, 12)**, **(7, B, 13)**, (14, D, 20), (17, C, 19)
d_4	(8, B, 16), **(18, A, 21)**, (24, D, 27), (25, E, 28)

Figure 5.4: Example of index sets

The following example illustrates the discovery of all frequent patterns having prefix $\langle A\rangle$ using steps 2 and 3. Using step 2, the algorithm constructs the index set $\langle A\rangle$-*idx*, by calling *CreateIndexSet(A, $\langle\ \rangle$, MDB)*. The resulting index set is shown in Figure 5.4(a). As shown in the figure, the index set $\langle A\rangle$-*idx* contains a set of client sequences that support $\langle A\rangle$. The value of *pos* of an index element pointing to d_1 is set to 1 because, with respect to the current prefix $\rho = \langle\ \rangle$, in d_1 a stem A is found at position 1. Analogously, in d_3 and d_4, a stem A is found at positions 1 and 2, respectively. The *a_intv* contains an interval corresponding to a pattern $\rho = \langle A \rangle$. Note that d_2 is not pointed to by any pointer in the index set because it does not contains a stem A (w.r.t prefix $\rho = \langle\ \rangle$).

Using step 3, the algorithm mines $\langle A \rangle$-*idx* to find all stems with respect to the prefix $\langle A \rangle$, by calling *MineIndexSet*($\langle A \rangle$, $\langle A \rangle$-*idx*). Each client sequence pointed by the index element is processed, by checking any state interval appearing after the *pos* position. The first element of $\langle A \rangle$-*idx*, which points to d_1, has the value of *pos* 1. Thus the search for potential stems only focus on the state intervals occurring after position 1. As a result, the algorithm increases the support count of a potential stem E for a potential pattern $p2_E = \begin{array}{c} \begin{array}{cc} A & E \end{array} \\ \begin{bmatrix} = & o \\ * & = \end{bmatrix} \end{array}$ by one. There are also potential stems B for a pattern $p2_B$, D for a pattern $p2_D$, and C for a pattern $p2_C$, where $p2_B = \begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \begin{bmatrix} = & o \\ * & = \end{bmatrix} \end{array}$, $p2_D = \begin{array}{c} \begin{array}{cc} A & D \end{array} \\ \begin{bmatrix} = & b \\ * & = \end{bmatrix} \end{array}$, and $p2_C = \begin{array}{c} \begin{array}{cc} A & C \end{array} \\ \begin{bmatrix} = & b \\ * & = \end{bmatrix} \end{array}$, respectively. Using the same process, the algorithm performs the support count for the states occurring after position 1 and 2 at the client sequences d_3 and d_4, respectively. After validating the support counts, the resulting stems are B ($sup = 50\%$), C ($sup = 50\%$), and D ($sup = 75\%$).

The process continues by recursively constructing and mining an index set ρ'-*idx*, where ρ' is formed by combining the prefix $\rho = \langle A \rangle$ and $s \in \{B, C, D\}$. Let first consider combining the prefix $\rho = \langle A \rangle$ and $s = B$. A pattern $\rho' = \begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \begin{bmatrix} = & o \\ * & = \end{bmatrix} \end{array}$ is outputted, and an index set ρ'-*idx* is constructed by calling *CreateIndexSet(B,* $\langle A \rangle$, $\langle A \rangle$-*idx*). The resulting index set is shown in Figure 5.4(b). When the procedure is called, the value of the third parameter, *range-set*, is the index set $\langle A \rangle$-*idx*, not the *MDB*. It means that in creating ρ'-*idx*, the algorithm only needs to consider the set of client sequences in $\langle A\rangle$-*idx* (i.e., d_1, d_3, and d_4), rather than all client sequences in *MDB*. With respect to the prefix $\langle A\rangle$, a stem B is at position 3 in d_1 and 2 in d_3. These values are stored at the field *pos* of the elements of the index set. The interval values of a state B in d_1 and d_3 are added to the array *a_intv*. There is no entry created for d_4 because it does not support a pattern ρ'. After creating ρ'-*idx*, the index set $\langle A\rangle$-*idx* is not discarded but it is stored for later use. Mining the index set ρ'-*idx*, the algorithm find stems C and D, which will form patterns $p3_C = \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} = & o & b \\ * & = & b \\ * & * & = \end{bmatrix} \end{array}$ and $p3_D = \begin{array}{c} \begin{array}{ccc} A & B & D \end{array} \\ \begin{bmatrix} = & o & b \\ * & = & b \\ * & * & = \end{bmatrix} \end{array}$, respectively.

The recursive process continues on the prefix $\rho = \begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \begin{bmatrix} = & o \\ * & = \end{bmatrix} \end{array}$ and a stem $s \in \{C, D\}$. Taking the prefix ρ and a stem C, the algorithm outputs a pattern $\rho' = p3_C$, and constructs ρ'-*idx*. The algorithm mines ρ'-*idx*, but it cannot find stems. Therefore, the recursive processing of the prefix ρ and a stem C cannot go further. The algorithm now considers the prefix ρ and a stem D. A pattern $\rho' = p3_D$ is outputted, then after creating and mining ρ'-*idx*, a stem C is found.

Continue the recursive process by taking the prefix $\rho = p3_D$ and a stem C, the algorithm outputs a pattern ρ' and creates an index set ρ'-*idx*, where $\rho' = \begin{array}{c} \begin{array}{cccc} A & B & D & C \end{array} \\ \begin{bmatrix} = & o & b & b \\ * & = & b & b \\ * & * & = & c \\ * & * & * & = \end{bmatrix} \end{array}$. The mining of ρ'-*idx* finds no more stems, so the recursive process cannot continue. At this stage, the algorithm has finished processing of the prefix $\rho = \langle A\rangle$ and a stem B.

Therefore, the algorithm now has to repeat the process by considering the prefix $\rho = \langle A\rangle$ and the remaining stems found during the mining of $\langle A\rangle$-*idx*, i.e., stems C and D. To make it short, the processing of $\rho = \langle A \rangle$ and a stem C would generate a frequent pattern $\begin{array}{cc} A & C \\ \left[\begin{array}{cc} = & b \\ * & = \end{array}\right] \end{array}$. Similarly, the processing of $\rho = \langle A \rangle$ and a stem D would result in temporal patterns $\begin{array}{cc} A & D \\ \left[\begin{array}{cc} = & b \\ * & = \end{array}\right] \end{array}$ and $\begin{array}{ccc} A & D & C \\ \left[\begin{array}{ccc} = & b & b \\ * & = & c \\ * & * & = \end{array}\right] \end{array}$. At this stage, the processing of a stem A with the prefix $\rho = \langle \ \rangle$ has finished. All frequent patterns having the prefix $\langle A \rangle$ have been found. Other frequent patterns can be discovered by recursively applying steps 2 and 3 on stems B, C, D, and E with prefix $\rho = \langle \ \rangle$.

5.2.2 Handling Large Databases

The above algorithm only works if the database fits into memory. If the database is too large to fit into memory, the frequent temporal patterns are discovered by *partition-and-validation* technique, as shown in Algorithm 5.4. First, the database is partitioned so that each partition can be processed in memory by ARMADA. In order to be frequent in the database, a temporal pattern has to be frequent in at least one partition. Therefore, the set of potential frequent patterns can be obtained by collecting the discovered patterns after running ARMADA on these partitions. The next step is the validation step, in which the actual frequent patterns can be identified through support counting against the data sequences with only one extra database pass. Therefore, ARMADA requires two passes of database scan to mine large databases that do not fit into memory.

5.2.3 Generating Temporal Association Rules

After all frequent temporal patterns have been discovered, temporal association rules can be generated from the patterns. Algorithm 5.5 shows the generation of

```
Input: a database $D$, $minsup$
Output: a set of frequent temporal patterns $F$
1: for each partition $D_i \subset D$ do
2:    $F_i$ = ARMADA($D_i$, $minsup$)
3: end for
4: $C = \cup_n F_i$
5: for each sequences $d \in D$ do
6:    Increment support count of all $c \in C$ contained in $d$
7: end for
8: $F = \{c \in C | \; sup(c) \geq minsup\}$
```

Algorithm 5.4: Pseudo code of ARMADA for processing large databases

richer temporal association rules from a set of frequent patterns.

```
Input: a set $F$ of all frequent patterns, $minconf$
Output: a set of temporal association rules
1: for all $\alpha \in F$ do
2:    for all $\beta \sqsubset \alpha$ do
3:       if $\frac{sup(\alpha)}{sup(\beta)} \geq minconf$ then
4:          Generate the rule $\beta \rightarrow \alpha$
5:       end if
6:    end for
7: end for
```

Algorithm 5.5: Pseudo code for generating richer temporal association rules

5.3 Maximum Gap Time Constraint

As with all temporal data mining algorithms, ARMADA can easily generate a very large number of frequent temporal patterns. One of the reasons is that in the above model, time gaps between intervals in the temporal patterns are not specified so that some uninteresting patterns are likely to appear. As an example, consider the database in Figure 5.3, in which without specifying the maximum gap, a temporal pattern *(A before C)* is frequent with the support of 50% (see Table 5.1). However, this pattern may be insignificant because the time gap between states A and C is too wide. Therefore, this model introduces a *maximum gap* time constraint to reduce the number of generated patterns and reinforce the significance of mining results.

Definition 5.9 (Maximum gap) Let α be a state sequence containing n state intervals (b_i, s_i, f_i), where $1 \leq i \leq n$, $b_i \leq b_{i+1}$ and $b_i < f_i$. The time gap between state intervals i and j, for $i < j$, is defined as $gap(i, j) = b_j - f_i$. The *maximum gap* of the sequence α is defined as $\delta(\alpha) = max\{gap(i, j)|i < j, i = 1, \ldots, (n-1) \text{ and } j = 2, \ldots, n\}$.

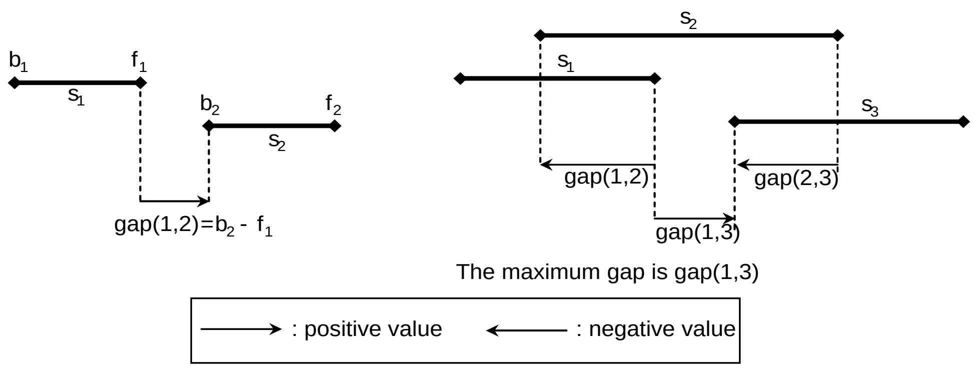


Figure 5.5: Determining gap and maximum gap in the state sequence

Figure 5.5 provides an example of determining the gap between two state intervals and the maximum gap in the sequence containing three state intervals. The figure shows that even though the absolute value of $gap(1, 3)$ is the smallest, but it is considered as the maximum gap of the sequence, because it is the only gap with a positive value.

Definition 5.10 (Containment) Given a user specified *maxgap*, a client sequence *d contains* a pattern $\alpha = (s_\alpha, M_\alpha)$ if (s_α, M_α) is a subpattern of (s_d, M_d), and the maximum gap of state intervals that take part in the pattern α is less than or equal to *maxgap*.

When the maximum gap constraint is specified, a client sequence that previously supports (contains) a pattern may no longer support it. As an example, consider a client sequence d_1 in Figure 5.3. Originally this client sequence supports a pattern *(B before D)*. If the maximum gap constraint is imposed by taking $maxgap = 2$, the sequence no longer supports the pattern because the maximum gap of state intervals involved in the pattern is bigger than *maxgap* (the gap between B and D is equal to 4). This is also the case for a client sequence d_4.

Note that if the value of *maxgap* is a positive integer, the constraint only affects intervals that have temporal relation *before*. Setting the $maxgap = \infty$ results in the original model as described in Section 5.1, where there is no time constraint specified.

To mine frequent patterns with the maximum gap constraint, the algorithm discussed in Section 5.2.1 can still be applied but it has to be modified so that when searching for stems it also checks the gap between intervals. The method to generate temporal rules from frequent pattern described in Section 5.2.3 is still applicable because the property that if a pattern p is frequent then so all its subpatterns still holds.

5.4 Evaluation

To assess the performance of the proposed algorithm, ARMADA was implemented in Java on a 2.4GHz Athlon PC with 512MB of RAM running Windows 2000 Professional. The user interface of ARMADA is presented in Appendix B.1. Several experiments were conducted using the synthetic and real datasets.

5.4.1 Experiments on Synthetic Data

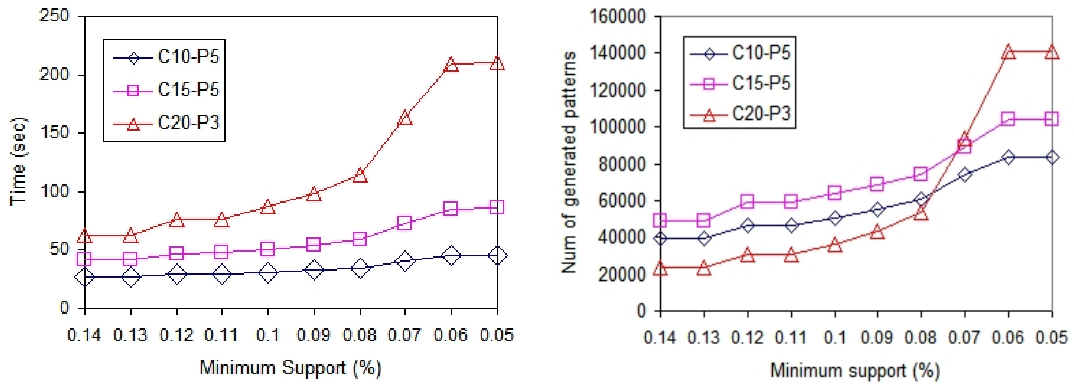
The synthetic data generation program takes five parameters, namely, the number of client sequences ($|D|$), average size of client sequences ($|C|$), number of maximal potentially frequent temporal patterns (N_P), average size of potentially frequent temporal patterns ($|P|$), and number of states (N). The data generation model is based on the one used for mining association rules (Agrawal & Srikant 1994) with some modification to model the interval sequence database.

The synthetic data generation procedure can be outlined as follows. First, a random pool of potentially frequent patterns are created. The number of potentially frequent patterns is N_P. A potential frequent pattern is generated by first picking the size of the pattern (the number of states in the pattern) from a Poisson distribution with mean equal to $|P|$. Then, each state in the pattern is chosen randomly (from N state types). The temporal relations between consecutive states are also determined randomly. Since the normalized temporal patterns are used (see Section 5.1), the temporal relations are chosen from the set {*before, meets, overlaps, is-finished-by, contains, starts, equals*}. If the pattern contains two similar consecutive states, their temporal relation is set to *before*. Each state in the pattern is then assigned an interval value according to its temporal relation with the state that comes before it. The interval value of the first state in the pattern is chosen randomly. Second, after all potentially frequent patterns are generated, an interval sequence database D is generated, consisting of $|D|$ client sequences. Each client sequence is generated by first determining its size, which is picked from a Poisson distribution with mean equal to $|C|$. Then, each client sequence is assigned a series of potentially frequent patterns. More detail about the data generator is presented in Appendix B.2.

In this evaluation, four sets of experiments were performed, by varying the minimum supports, the maximum gaps, the number of states, and the size of databases (number of sequences). In each set of experiments, the processing times of the algorithm and the number of generated frequent patterns were recorded.

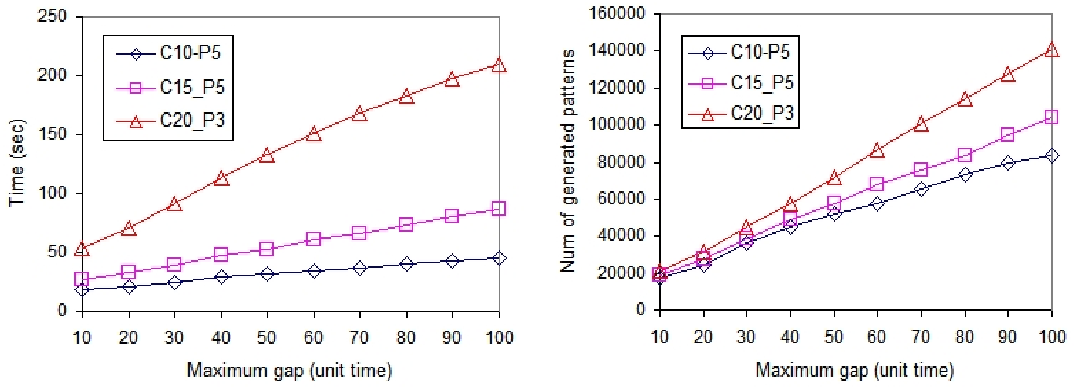
First, the effect of varying minimum support on the the processing times was investigated. Three datasets were used, each has the value of N = 1000 and N_P = 2000. The values of $|C|$ and $P|$ were varied, using three values of $|C|$: 10, 15, and 20, and two values of $|P|$: 3 and 5. The number of client sequences $|D|$ was set to 10,000. The algorithm was run on each dataset, by varying the values of minimum support from 0.05% to 0.14%, and setting the value of maximum gap to 100 time units.

Figure 5.6 shows the number of generated patterns and the processing times as the values of minimum support decreases. As expected, as the minimum support decreases, the processing times increase for all datasets (Figure 5.6(a)). This is because as the minimum support decreases, the number of generated patterns increases (Figure 5.6(b)), resulting in increasing processing times. The dataset with longer sequences requires more processing times compared to that with shorter sequences. Consider the dataset *C20-P3* (Figure 5.6(a)), even though its generated patterns are not always the highest, its processing times are the highest for all values of minimum support.



(a) (b)

Figure 5.6: Effect of decreasing minimum support



(a) (b)

Figure 5.7: Effect of increasing maximum gap

The second set of experiments looked into the effect of varying maximum gap on the processing times. In these experiments, the datasets used were the same as in the previous experiments. However, these experiments set the value of minimum support constant at 0.05%, but varied the values of maximum gap from 10 to 100 time units. As shown in Figure 5.7(a), the processing times increase as the values of maximum gap increase. Similar to the previous experiments, when the maximum gap increases, more frequent patterns will be generated (Figure 5.7(b)), which resulting in increasing processing times. The dataset with longer sequences requires more processing times compared to that with shorter sequences.

For the third set of experiments, two sets of datasets were used. The first set has $|C| = 10$ and $|P| = 5$, while the second set has $|C| = 15$ and $|P| = 5$. The database size was constant at $|D| = 50{,}000$ and the value of $N_P = 2000$. The values of minimum support and maximum gap were set to 0.05% and 100, respectively. Figure 5.8 shows the processing times and the number of generated patterns when the number of states (N) is increased from 1000 to 10,000. It can be seen in Figure 5.8(a) that the processing times increase as the number of states is increased. However, the number of generated patterns tends to decrease as the number of states increases (Figure 5.8(a)).

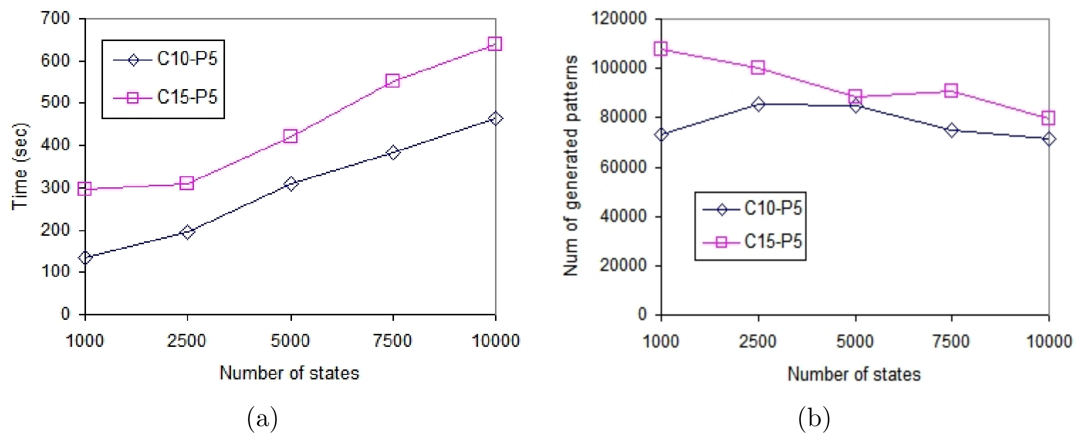
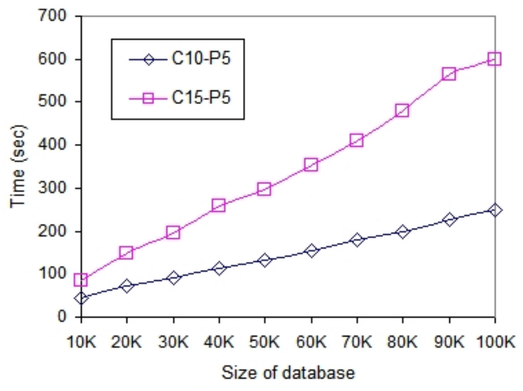
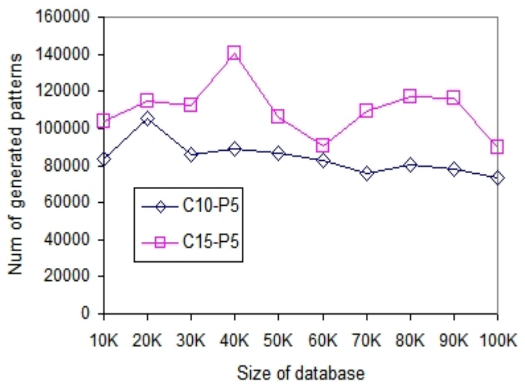


Figure 5.8: Effect of increasing number of states

The last set of experiments investigated how the algorithm scales up as the size of the database ($|D|$) increases. Similar to the previous experiments, two sets of datasets were used, the first one has $|C| = 10$ and $|P| = 5$, while the second one has $|C| = 15$ and $|P| = 5$. All datasets have the values of $N_P = 2000$ and $N = 1000$. The minimum support was set to 0.05% and the maximum gap was set to 100. Figure 5.9(a) shows the algorithm scales up linearly as the size of databases increases from $10K$ to $100K$, regardless of fluctuation on the number of generated patterns.



(a)



(b)

Figure 5.9: Effect of increasing database size

5.4.2 Experiments on Real Data

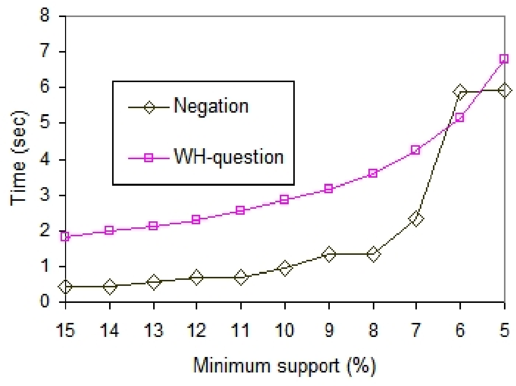
In addition to using synthetic datasets, we have also performed a series of experiments on a real dataset. The dataset is the ASL database created by the National Center for Sign Language and Gesture Resources at Boston University, which is available online at: http://www.bu.edu/asllrp/. The Sign-Stream(TM) database used in the experiment contains a set utterances, where each utterance associates a segment of video with a detailed transcription. Every utterance can be considered as a state sequence which contains a number of ASL gestural and grammatical fields (e.g. eye-brow raise, head tilt forward, wh-question), each one occurring over a time interval. The overall list of field names and labels included in the database are given in Table 5.2 (Papapetrou 2006).

In this experiments, the algorithm was run on the subsets of sentences from

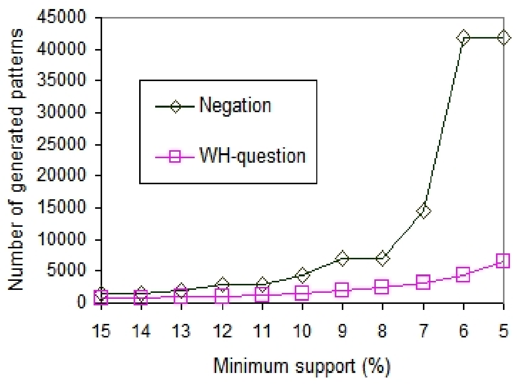
Table 5.2: List of field names and labels

Fields	Fields Names	Fields Tables
Head position	head pos: tilt fr/bk	hp: tilt fr/bk s
	head pos: turn	hp: turn
	head pos: tilt side	hp: tilt side
	head pos: jut	hp: jut
Head movement	head mvmt: nod	hm: nod
	head mvmt: shake	hm: shake
	head mvmt: side to side	hm: side$< - >$side
	head mvmt: jut	hm: jut
Body	body lean	body lean
	body mvmt	body mvmt
	shoulders	shoulders
Eyes, Nose, and Mouth	eye brows	eye brows
	eye gaze	eye gaze
	eye aperture	eye apert
	nose	nose
	mouth	mouth
	English mouthing	English mouthing
	cheeks	cheeks
Neck	neck	neck
Grammatical information	negative	negative
	wh question	wh question
	yes-no question	yes-no question
	rhetorical question	rhq
	topic/focus	topic/focus
	conditional/when	cond/when
	relative clause	rel. clause
	role shift	role shift
	subject agreement	subj agr
	object agreement	obj agr
	adverbial	adv
Part of Speech	POS	POS
	Non-dominant POS	POS2
Gloss Fields	main gloss	main gloss
	non-dominant hand gloss	nd hand gloss
Text Fields English	translation english	english

the database: those that contained marking of a negation, and another that contained marking of wh-question. The first dataset contains 65 state sequences (utterances), while the second dataset contains 730 state sequences, with an average number of items per sequence equal to 26 and 32 respectively. The algorithm has been tested by varying supports from 5% to 15% and setting the maximum gap to 200. The processing times of the algorithm and the number of generated frequent patterns were recorded. The experimental results are shown in Figure 5.10. As shown in Figures 5.10(a), the processing times increase for both datasets as the minimum support decreases. As in the case of synthetic data above, as the minimum support decreases the number of generated patterns increases (Figure 5.10(b)), resulting in increasing processing times. The dataset with longer and more sequences requires more processing times compared to that with shorter and less sequences. As can be seen from the figures, even though the WH-question data always has lower number of generated patterns than the Negation data, the processing times of the former are mostly higher than that of the latter.



(a)



(b)

Figure 5.10: Effect of decreasing minimum support (ASL database)

5.5 Summary

This chapter has presented the discovery of richer temporal association rules from interval sequence database. A new algorithm, ARMADA, is proposed to discover the rules. The algorithm utilizes a simple index advancing to grow longer temporal patterns from the shorter frequent ones, so that it does not require candidate generation or database projection. The algorithm has been implemented, and several sets of experiments have been conducted to evaluate the performance of the algorithm. The algorithm looks promising as a method for discovering richer temporal association rules from interval sequence databases. In addition, this chapter has also proposed a maximum time constraint to reduce the number of patterns generated by the algorithm, which in turn reduce the number of generated rules.

As for other techniques for rule discovery, ARMADA could generate a large number of rules. Finding interesting rules is a difficult task when the number of rules is large. This problem is made worse as richer temporal association rules have more complex structure than, for example, association rules. This thesis addresses this problem by proposing a retrieval system to facilitate the finding of interesting rules from a set of discovered rules. The retrieval system is described in Chapter 7.

Chapter 6

Review of Set and Sequence Retrieval

The retrieval of data objects on set-valued attributes (for short set retrieval) is an important research topic with wide areas of applications. A significant amount of today's stored data consists of records with set-valued attributes (i.e. attributes that are sets of items). Set-valued attributes are extensively used in object-oriented databases to represent an object's multivalued attribute (Ishikawa, Kitagawa & Ohbo 1993), in multimedia databases representing objects inside an image (Rabitti & Zezula 1990), and in data mining applications representing basket market data (Morzy & Zakrzewicz 1998). Although advanced database systems, such as nested relational or object-oriented database systems, provide the means to store set-valued attributes in the databases, they do not provide language primitives or indexes to process and query such attributes. Furthermore, some of the existing index structures proposed in the database literature, for example, (B+ trees (Comer 1979) and R trees (Guttman 1984), etc.), are not designed to fully support set value manipulation in general. Therefore, new types of index structure have been proposed in the literature to support queries on set-valued attributes, namely, signature files and inverted files.

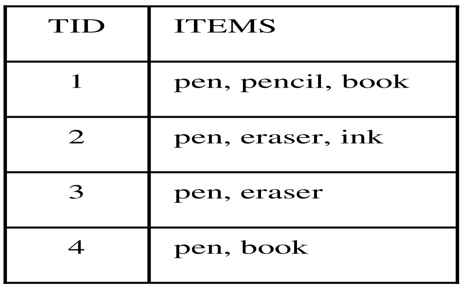
One of application domains that would benefit from the possibility of performing efficient querying on sets is data mining. Several data mining techniques rely on excessive set processing, especially in the case of mining association rules using the Apriori family of algorithms. Shifting these computations from the data mining algorithms to the database engines could result in considerable time savings. Efficient set retrieval is also useful during pattern post-processing for the selection of discovered association rules according to user-defined criteria, or for the querying of the database against association rules to identify transactions that satisfy certain criteria. Recently the set retrieval using signature files has also been used as a basis for sequential patterns retrieval.

In order to provide the support for the retrieval of richer temporal association rules described in Chapter 7, this chapter reviews current work on set retrieval using inverted files and signature files. In addition, this chapter also reviews sequential pattern retrieval using signature files. The chapter is organised as follows. Section 6.1 defines different types of queries normally used in the set retrieval. Section 6.2 describes the retrieval of sets using inverted files. Section 6.3 discusses the basic concept of signature files for set retrieval, followed by the discussion of false drop probability (Section 6.4), and signature file organisations (Section 6.5). Section 6.6 describes the sequential pattern retrieval using signature files.

6.1 Types of Queries in Set Retrieval

Formally, set retrieval is defined as retrieval of data objects based on the set predicates from a large number of set-valued objects stored in the database (Kitagawa & Fukushima 1996). The most common class of queries used in set retrieval is *subset queries* that search for sets that contain the query set. However, in addition to the subset queries, set retrieval also needs to support *superset queries*, that is, queries looking for subsets of the query set. Furthermore, retrieving sets equal to a query set, i.e., *equality queries*, though a more simple case, is also considered.

In the data mining field, an example of data models supporting set values and their retrieval is a market basket data, shown in Figure 6.1. The database contains four transactions, each of which is tagged with the transaction identifier (TID). ITEMS is a set attribute and represents the set of items bought in each transaction. In this case, each ITEMS attribute value is called a *target set*. Suppose a *query set* $Q = \{pen, book\}$, a subset query $ITEMS \supseteq Q$ retrieves transactions that contain both *pen* and *book*. Superset queries retrieve all transactions that are proper subset of the query set. If the query set Q is a set of products on sale, a superset query $ITEMS \subseteq Q$ can be used to find the transactions that are entirely included in the reduced product set. Equality queries retrieve transactions equal to the query set.



TID	ITEMS
1	pen, pencil, book
2	pen, eraser, ink
3	pen, eraser
4	pen, book

Figure 6.1: Example of market basket database

A formal definition of these set queries is given below. To simplify the discussion, without loss of generality, the focus is only on the set-valued attribute of data objects; other attributes, if available, are ignored.

Definition 6.1 (Set queries) Let D be a collection of target sets, each associated with the data object identifier. Let T and Q denote the target set and query set, respectively. The commonly used queries in set retrieval are as follows.

1. Subset query ($T \supseteq Q$): Find target sets in D that have the query set as a subset.

2. Superset query ($T \subseteq Q$): Find target sets in D that are subset of the query set.

3. Equality query ($T \equiv Q$): Find target sets in D equal to the query set.

6.2 Set Retrieval using Inverted Files

Inverted files have been used extensively in text retrieval (Moffat & Zobel 1996, Zobel, Moffat & Ramamohanarao 1998). Their main application is for supporting partial match retrieval, which is basically subset queries. This section describes the use of inverted files for supporting set retrieval. Given a collection of target sets, D, an inverted file consists of a *directory* containing all distinct values in D and, for each value in the directory, an *inverted list* that stores a list of references to all occurrences of this value in the database, that is a list of references to target sets containing the value. Consider the market basket data containing four transactions and five items in Figure 6.1. Using this database an inverted file index can be created and is shown in Figure 6.2. An interted list of an item is presented as $\langle n; tid_1, \cdots, tid_n \rangle$, where n is the number of transactions in which an item appears, followed by transaction identifiers (tids). As shown in the figure, the inverted list of the item *book* is $\langle 2; 1, 4 \rangle$ because it appears in two transactions, namely, transactions 1 and 4. If D contains a large number of items, the search values in the directory are usually stored in the B-tree.

Helmer and Moerkotte (1999) have adapted inverted files for set retrieval and modified the inverted list by also storing the cardinality of the target set with each target set reference, so that set queries can be answered more efficiently by using the cardinalities as a quick pre-test. As an example, the inverted list of the item *book* becomes $\langle 2; (1, 3), (4, 2) \rangle$.

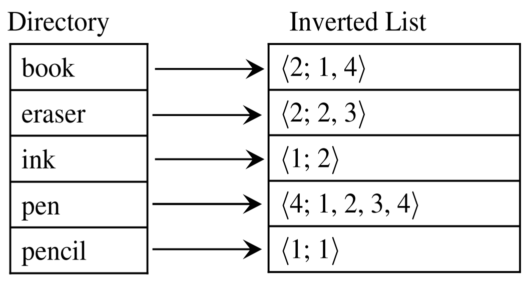


Figure 6.2: Inverted file of the market basket database

After an inverted file has been built, subset queries can be processed as follows:

for each item in the query set the appropriate list is fetched, and then all those lists are intersected. The result of intersection contains a list of references to sets that contains the query set. The equality queries are processed the same way as with subset queries, but can be improved by eliminating all references to target sets whose cardinality is not equal to the query set cardinality. When evaluating superset queries, all lists associated with the values in the query set are retrieved. Then the number of occurrences of each reference appearing in the retrieved lists is counted. A reference whose number of occurrences is not equal to the cardinality of its set is eliminated. The existence of such a reference means that the reference appears in the lists associated with the values that are not in the query set, so that its set cannot be a subset of the query set

Helmer and Moerkotte (1999) compare the performance of three signature-based indexes against that of the inverted file index in processing equality, subset and superset queries. They conclude that the inverted file index structure dominated other index structures for subset and superset queries in terms of query processing time. Kouris *et al.* (2004) have used the inverted file index to improve the performance of an Apriori-based algorithm in discovering association rules. The index is accessed during support counting so that instead of reading the original database, the mining algorithm scans the inverted file index stored in memory. Tuzhilin and Liu (2002) use inverted file indexing scheme for querying multiple sets of discovered association rules. A comparison between inverted files and signature files is also studied by Zobel *et al.* (1998) and Carterette and Can (2005).

6.3 Set Retrieval using Signature Files

The purpose of using signature files in set retrieval is to filter out the non-qualifying data objects. The basic idea is to represent the set-valued attribute of data objects into bit patterns, called *signatures*, and store them in a separate file which acts as a filter to eliminate the non-qualifying data objects when processing set queries. A signature failing to match the query signature guarantees that the corresponding object can be ignored. Therefore, unnecessary object access is prevented. Since direct set comparisons are very expensive, using signatures as filters can speed up query processing in set retrieval.

6.3.1 Methods for Generating Set Signatures

In set retrieval with signature files, a *target signature* is generated for each target set and stored in the signature file. A number of signature generation methods have been proposed by Faloutsos and Chistodoulakis (1987) in the context of text retrieval. These methods are Word Signatures (WS), Superimposed Coding (SC), Bit-Block Compresion (BC), and Run Length Encoding (RL). The description of each method for generating target signatures is given below.

1. **Word signature (WS)**. In the WS method each element of the target set is hashed into a bit pattern of a certain length. These patterns, called word signatures, are then concatenated to form the target signature.

2. **Superimposed Coding (SC)**. In the SC method, each element in a target set is hashed to a binary bit pattern called an *element signature*. All element signatures have F bit length, and exactly m bits are set to '1', where $m < F$. F is called the *length* of a signature, while m is called the *weight* of an element signature. Then, a target signature is obtained by bit-wise OR-ing (*superimposed coding*) element signatures of all the elements in the target set.

3. **Bit-Block Compression (BC)**. The signature extraction process for BC is similar to SC. The difference is that the original size (length) of the signature, designated as B, is large, and for each element of a target set only one bit is set to '1' (i.e., $m = 1$). As a result, the bit vector B of the set signature is sparse. Therefore, before storing the signature, B is divided into groups of consecutive bits of size b and compressed.

4. **Run Length Encoding (RL)**. The RL method is similar to both SC and BC. It differs from BC only in the compression method. RL records the distances between the positions of bits with value '1'.

Of these four methods, the most commonly used method in set retrieval is the superimposed coding (Helmer & Moerkotte 1999, Ishikawa et al. 1993, Tousidou, Bozanis & Manolopoulos 2002, Morzy & Zakrzewicz 1998). Therefore, for the rest of this chapter, unless stated otherwise, it will be assumed that the superimposed coding is used to generate set signatures. Figure 6.3 illustrates the generation of set signature using the superimposed coding when the value of $F = 8$ and $m = 2$.

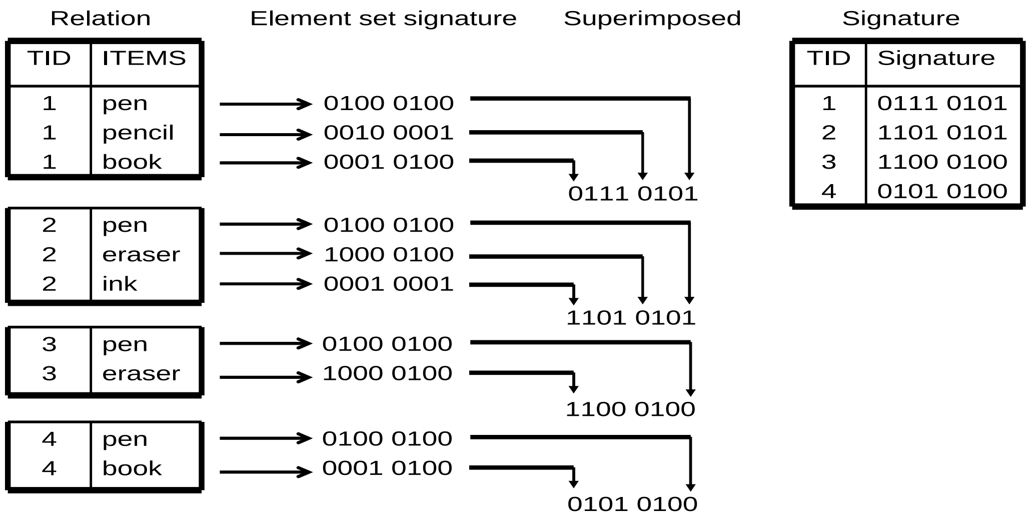


Figure 6.3: Generating signature using superimposed coding

Property 6.1 (Properties of set signatures) Let $sig(s)$ and $sig(t)$ be signatures of sets s and t, respectively. Set signatures have three properties, as follows:

1. $s \supseteq t \rightarrow sig(s) \wedge sig(t) = sig(t)$[1]
2. $s \subseteq t \rightarrow sig(s) \wedge sig(t) = sig(s)$
3. $s = t \rightarrow sig(s) = sig(t)$

These properties are useful during the processing of set queries because they can be used as a quick pre-test to determine whether a target set satisfies a query

[1]The symbol $\wedge$ denotes the bit-wise AND operation.

condition or not (see Section 6.3.2 below).

6.3.2 Processing Set Queries

The processing of set queries with signature files is conducted in two steps. In the first step, called the *filtering step*, a *query signature* is generated from the query set (in the same way as the target signature). Then, each target signature in the signature file is examined over the query signature for potential match. The corresponding target set becomes a *drop* if the target signature satisfies the following condition (according to the type of the query):

1. $T \supseteq Q$: *target signature* $\wedge$ *query signature* $=$ *query signature.*
2. $T \subseteq Q$: *target signature* $\wedge$ *query signature* $=$ *target signature.*
3. $T \equiv Q$: *target signature* $=$ *query signature.*

A target set that becomes a drop has a potential to satisfy the query condition. On the other hand, a target set whose signature does not satisfy the condition can be ignored because it will not satisfy the query condition. The second step is the *false drop resolution.* Each drop is retrieved and examined to see whether it actually satisfies the query condition. Drops that fail the test are called *false drops*, while the qualified data objects are called *actual drops.*

As an example, consider four transactions and their signatures in Figure 6.3. Given a target set $Q = \{pen,\ book\}$, the subset query is to find all transactions that contain the target set Q. From the figure, the signature of Q is '01010100' (the fourth transaction). After matching the target signatures against the query signature for a condition *'target signature* $\wedge$ *query signature* $=$ *query signature'*, transactions 1, 2, and 4 satisfy the condition and become drops. Further examination results in transactions 1 and 4 that actually satisfy the query condition, that is, they contain a query set Q. On the other hand, transaction 2 does not contain the query set. Transactions 1 and 4 are called actual drops, while transaction 2 is called a false drop.

6.4 False Drop Probability

False drops occur due to the collision of set signatures and depend solely on the method to generate signatures and not on other factors such as signature file organisation. False drops not only affect the number of block accesses (I/O time), but also affect the CPU time in order to decide whether a target set should be returned to the user. Therefore, one of the problems of the signature file methods is how to control the false drops. The number of false drops is usually measured using *false drop probability.* Suppose F_d is the false drop probability, F_d is defined as:

$$F_d = \frac{\text{false drops}}{N - \text{ actual drops}} \tag{6.1}$$

where N is the total number of target sets.

A number of studies have addressed the problem of estimating the false drop probability (Faloutsos & Christodoulakis 1987, Zezula & Tiberio 1990, Kitagawa, Fukushima, Ishikawa & Ohbo 1993). The estimated value of F_d is useful in the optimisation of signature file parameters. The more accurate the estimation of F_d, the better the estimation of signature file parameters, which in turn provides superior performance in real applications.

Faloutsos and Christodoulakis (1987) have analysed the false drop probability of the four signature extraction methods. However, they do not consider storage structures and their effects on the query processing performance. Such comparison is presented by Zezula and Tiberio (1990).

Kitagawa *et al.* (1993) have derived formulas to estimate the false drop probability for each type of set query defined in Section 6.1 under superimposed coding. Assuming that all target sets have the same cardinality D_t, the false drop prob-

Table 6.1: Symbols

Symbol	Definition
F	Size of signature (in bits)
m	Weight of a signature element
D_t	Cardinality of a target set T
D_q	Cardinality of a query set Q
$\bar{m}$	Weight of every target or query signature

ability for each query is given in the following formula:

$$T \supseteq Q: \quad F_d \approx \left(1 - e^{-\frac{mD_t}{F}}\right)^{mD_q} \tag{6.2a}$$

$$T \subseteq Q: \quad F_d \approx \left(1 - e^{-\frac{mD_q}{F}}\right)^{mD_t} \tag{6.2b}$$

$$T \equiv Q: \quad F_d = \frac{1}{{}_F C_{\bar{m}}} \tag{6.2c}$$

Table 6.1 provides the meaning of symbols used in the formulas.

The value of F_d would be minimum (optimal) for the following value of m (Kitagawa et al. 1993):

$$T \supseteq Q: \quad m_{opt} = \frac{F \ln 2}{D_t} \tag{6.3a}$$

$$T \subseteq Q: \quad m_{opt} = \frac{F \ln 2}{D_q} \tag{6.3b}$$

$$T \equiv Q: \quad m_{opt} = \frac{F \ln 2}{D_t} \tag{6.3c}$$

6.5 Signature File Organisation

If the false drop rate is low, the number of accesses to the storage level can be reduced. However, the efficiency of filtering is determined mainly by the storage structures of signature files that support the filtering process. A number of approaches have been proposed in signature file organisations, and in general can be classified into four categories, namely, *sequential*, *bit-slice*, *hierarchical*, and *partitioned* signature file organisations. The following four subsections describe each of these signature file organisations.

6.5.1 Sequential Organisation

The Sequential Signature File (SSF) is the simplest file structure, in which signatures are stored sequentially in the signature file. Figure 6.4 shows SSF consisting of N signatures with $F = 8$. SSF is easy to implement and requires low storage space and low update cost. However, during the retrieval, a full scan of the signature file is required. That is, in processing subset, superset, and equality queries, every signature in the signature file needs to be accessed (sequentially) to determine which signature qualifies. The performance of SSF is not dependent on the query signature weight, but linearly dependent on the size of the file. Therefore, SSF works well for a data file with a small size, however, its performance becomes a problem when the size of a data file is large.

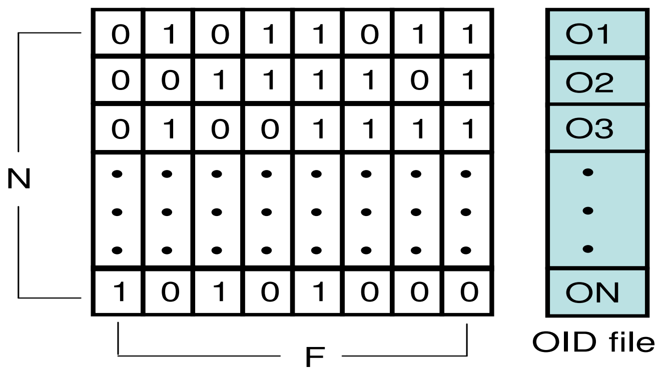


Figure 6.4: SSF file organisation

6.5.2 Bit-Sliced Organisation

A variation of sequential file organisation, known as *bit-slice* file organisation, has been proposed to avoid reading unnecessary bits from the file and to improve the performance of SSF. Robert (1979) introduces the Bit-Slice Signature File (BSSF), which stores signatures in a column-wise manner. For a set of signatures of length F, BSSF uses F files (called *bit-slice files*), one for each bit position of the set signatures. The structure of BSSF for N signatures and $F = 8$ is shown in Figure 6.5. The advantage of such an arrangement is that during retrieval only a part of the F bit-slice files have to be scanned. Given a query signature '1001 0100', to process the subset query only three bit-slices corresponding to bit position of '1' in the query signature need to be accessed. On the other hand, to process the superset query five bit-slices corresponding to bit position of '0' need to be accessed. As a result, the search cost of BSSF is lower than that of SSF, while its update cost is more expensive because an insertion of a new signature requires about F disk accesses (one for each bit-slice file). Therefore, BSSF is more suitable for stable files or archives.

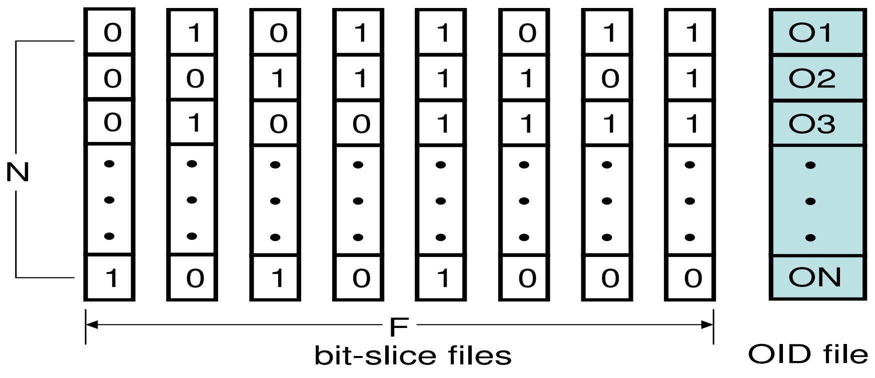


Figure 6.5: BSSF file organisation

While the performance of SSF is not dependent on the weight of the query signature, the performance of BSSF is dependent on it. In processing subset queries, as the weight of the signature becomes smaller, the number of bit-slices which needs to be scanned decreases. This generally leads to reduction of retrieval cost. In contrast, in processing superset queries, the cost reduction becomes generally larger as the weight of the signature becomes bigger.

Therefore, to achieve optimal performance when processing both subset and superset queries, these conflicting requirements should be compromised. As an approach to this problem, Kitagawa and Fukushima (1996) propose the Composite Bit-Sliced Signature File (C-BSSF). In C-BSSF, each signature S is a concatenation of two sub-signatures S_1 and S_2, whose design parameters are determined independently of each other. Therefore, C-BSSF can be regarded as a concatenation of two BSSFs with different design parameters sharing the same OID file. C-BSSF becomes BSSF when $F_1 = 0$ or $F_2 = 0$.

Several other improvements of BSSF have also been proposed, by means of vertical or horizontal decomposition. Lin and Faloutsos (1992) propose the Frame-Sliced Signature File (FSSF), in which each signature of length F is vertically decomposed into k frames of s bits each ($k \cdot s = F$). The set signature is generated as follows: for each element in the set, n ($n \leq k$) distinct frames are selected and m bit positions are set to '1' in each selected frame by using a hash function. In processing a subset query, for example, if there are only n_q frames which include '1' in the query signature, then only the n_q frames out of k need to be looked up to find drops. As a result, the vertical decomposition in FSSF contributes to the reduction of the signature look-up cost. In processing a superset query, similar reduction can be achieved by focusing on bit positions set to '0' in the query signature. When $s = 1$, namely each frame has one bit length, FSSF becomes BSSF. In general, the retrieval cost of FSSF is lower than that of BSSF, while update is rather expensive in FSSF.

Kitagawa *et al.* (1996) propose the Partitioned Frame-Sliced Signature File (P-FSSF) which combines the vertical decomposition of FSSF and the horizontal decomposition of the FP-partitioned signature file (Lee & Leng 1989). The storage space of the signature file is horizontally decomposed into partitions and each partition is vertically decomposed into frame-slices. Each frame-slice corresponds to one disk page. For subset queries, the vertical decomposition of FSSF attains much cost reduction when the query signature weight is small, while horizontal decomposition of the FP-partitioned signature file is effective when the query signature weight is large. As for superset queries, FSSF gives much contribution when the query signature weight is large, while the FP-partitioned signature file is effective when the query signature weight is small. Therefore, by incorporating both vertical and horizontal decomposition schemes, P-FSSF is more efficient than both FSSF and the FP-partitioned signature file.

6.5.3 Hierarchical Organisation

The bit-slice approach still needs to examine every signature in the file but only a part of it. In order to avoid reading every signature in the signature file, the hierarchical file organisation uses several levels of signatures. The higher levels perform coarse filtering before the signature on the lower levels are consulted.

An example of hierarchical file organisation is S-tree (Deppisch 1986). An S-tree is a height balanced dynamic tree (all leaves are on the same level) similar to a B^+-tree (Comer 1979). The leaves of the tree store the target signatures, while its internal nodes store signatures that are formed by superimposing the signatures of their children nodes. The S-tree is defined by two integer parameters K and k. The root can accommodate at least two and at most K signatures, whereas all other nodes can accommodate at least k and at most K signatures. The tree height for n signatures is at most $h = \lceil log_k n - 1 \rceil$ (Tousidou, Nanopoulos & Manolopoulos 2000). Figure 6.6 shows the structure of S-tree that contains 15 target signatures with the value of $K = 4$ and $k = 2$.

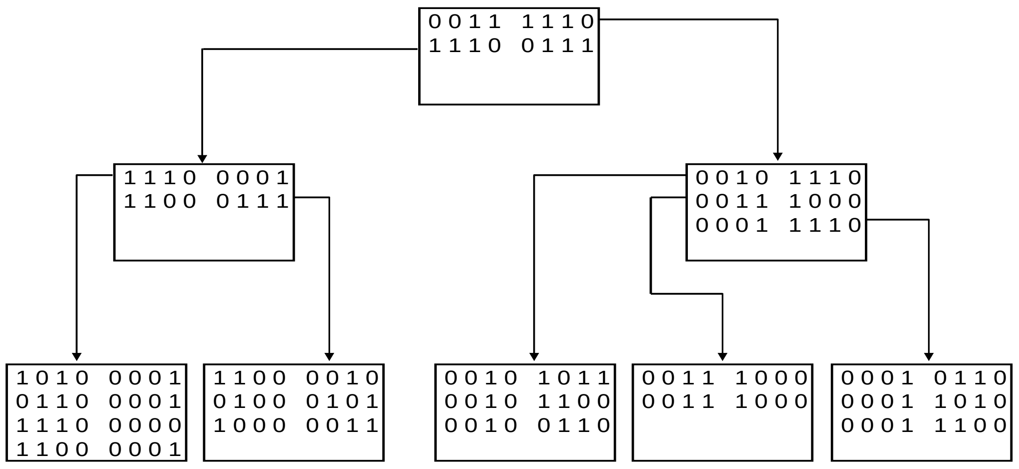


Figure 6.6: Structure of S-tree (K=4 and k=2)

Given a query signature *sig(Q)*, the query processing on S-tree starts by comparing the query signature to the signatures store in the root. More than one signature may satisfy this comparison. The process is repeated recursively to access all child nodes whose signatures match and go down to the leaf nodes.

For subset and equality queries, signatures in the root and other internal nodes are considered match with the query signature if *sig(Q)* $\subseteq$ *sig(child node).* For superset queries, there has to be a nonempty intersection, i.e., $|$*sig(Q)* $\cap$ *sig(child node)*$| \geq k$. At the leaf nodes all data items whose signatures match with the query signature are fetched and verified. Matching signatures in leaf nodes are determined by the appropriate bitwise operation (see Property 6.1).

6.5.4 Partitioned Organisation

The partitioned file organisation approach avoids reading every signature by grouping the signatures into several partitions such that all signatures in a partition hold the same part, called the *signature key.* The signature key used is usually a substring of the signature. By partitioning the signatures, some of the partitions need not to be searched during the execution of a query, so that the number of accesses can be reduced.

Lee and Leng (1989) propose the FP-partitioned signature file and examine several methods of partitioning static signature file. The basic idea of the FP-partitioned signature file is to decompose the set of signatures into partitions using the fixed number of prefix bits as a key. For example, when a two-bit prefix is used as a key, there are four partitions corresponding to '00' through '11'. Since the main implementation problem of any partitioned organisation is the definition of keys, this work proposes three partitioning schemes with different key specifications, namely *Fixed Prefix*, *Extended Prefix*, and *Floating Key.* There has been an attempt to make these schemes dynamic (Lee & Leng 1990). However, the way of determining the keys still remains static, which limits the dynamic features of the file organisations.

Another partitioning scheme is Quickfilter (Rabitti & Zezula 1990, Zezula, Rabitti & Tiberio 1991), which is supported by the dynamic storage structure linear hashing (Litwin 1980). The signature key is defined as the h-bit or the $(h-1)$-bit suffix of the signature. The value of h varies with the size of the file, N, according to the relationship $2^{h-1} < N \leq 2^h$. Quickfilter considers any signature key as a binary number, the value of which determines the partition number into which the signature is stored.

Helmer and Moerkotte (1999) propose the Extendible Signature Hashing (ESH) similar to Quickfilter. However, instead of using linear hashing, ESH uses extendible hashing (Fagin, Nievergelt & Strong 1979) as the underlying hashing scheme. ESH consists of two parts, a *directory* and *buckets*. The buckets are used to store the signatures of target sets. Given a target signature, a bucket that stores the signature is determined by looking at a d-bit prefix of the target signature. Each possible bit combination of the prefix corresponds to an entry in the directory pointing to the corresponding bucket. As a result, for the d-bit prefix the directory has 2^d entries.

6.6 Sequential Pattern Retrieval

Set retrieval with signature files described above does not consider the order of items within the sets, as in the case of indexing/retrieval of sequential patterns. To overcome this limitation, a new approach has been proposed for the retrieval of sequential patterns (Zakrzewicz 2001, Morzy, Wojciechowski & Zakrzewicz 2001, Nanopoulos, Zakrzewicz, Morzy & Manolopoulos 2003). The basic idea is to convert the sequential patterns into sets, called *equivalent sets*, that can accommodate not only the items in sequences but also the ordering of the items. After that, the same steps used in the set retrieval (Section 6.3) can be applied on the equivalent sets.

6.6.1 Representing Sequential Patterns as Sets

In order to represent sequential patterns in equivalent sets, two mapping function are required: *item mapping* and *order mapping* functions. Both functions are defined below.

Definition 6.2 (Item mapping) Let I be a set of items. An item mapping function $f(i)$ is a function that transforms item $i \in I$ to into an integer value.

For example, for $I = \{a, b, c, d, e\}$, the mapping $f(a) = 1$, $f(b) = 2$, $f(c) = 3$, $f(d) = 4$, and $f(e) = 5$ can be used.

Definition 6.3 (Order mapping) An order mapping function $g(x, y)$ is a function that map a sequential pattern of the form $\langle (x)(y) \rangle$ (i.e., x before y, denoted as $x < y$) into an integer value.

Since a function g takes into account the ordering of items, it should have a property that $g(x, y) \neq g(y, x)$. As an example, if $g(x, y) = 6 \cdot f(x) + f(y)$, then $g(a, b) = 6$ and $g(b, a) = 13$. The use of a constant value 6, which is one larger than the largest $f(x)$, will guarantee that $g(x, y) \neq g(y, x)$, for all $x, y \in I$.

Definition 6.4 (Equivalent Set) Given a sequential pattern $p = \langle X_1 X_2 \ldots X_n \rangle$, where X_i is an element of the sequence, the equivalent set $E(p)$ of p is defined as:

$$E(p) = \left(\bigcup_{x \in X_1, \ldots, X_n} \{f(x)\} \right) \bigcup \left(\bigcup_{x, y \in X_1, \ldots, X_n, x < y} \{g(x, y)\} \right),$$

For example, let $p = \langle (ab)(c)(d) \rangle$ be a sequential pattern. Using the mapping functions that described above, the equivalent set $E(p)$ of p is:

$$\begin{aligned} E(p) = & \{f(a), f(b), f(d), f(d)\} \\ & \cup \{g(a, c), g(b, c), g(a, d), g(b, d), g(c, d)\} \\ = & \{1, 2, 3, 4\} \cup \{9, 15, 10, 16, 22\} \\ = & \{1, 2, 3, 4, 9, 15, 10, 16, 22\} \end{aligned}$$

An equivalent set is the union of two sets: the one resulting by considering each element separately and the other from considering pairs of items between different elements of a sequence. The equivalent set has the property that for two sequential patterns p and q if q is contained in p, then $E(q) \subseteq E(p)$. Therefore, by representing sequential patterns as equivalent sets, the sequential pattern retrieval problem can be solved using techniques employed in set retrieval described in the previous sections.

6.6.2 Partitioning Equivalent Sets

The size of an equivalent set quickly increases when the number of the original sequence elements increases. A partitioning technique is proposed to divide an equivalent set into a collection of smaller subsets. It is expected that by partitioning equivalent sets, the resulting signatures would have smaller collision probability and fewer false drops, which in turn can reduce data scan cost (Zakrzewicz 2001, Morzy et al. 2001).

Definition 6.5 (Partitioning of equivalent sets) Given a user-defined value β, the equivalent set E of a sequential pattern p is partitioned into a collection of $E_1, \ldots, E_k$ subsets by:

1. dividing p into $p_1, \ldots, p_k$ subsequences, such that $\bigcup_{i=1}^{k} p_i = p$, $p_i \cap p_j =$ for $i \neq j$, and
2. having E_i be the equivalent set of p_1, where $|E_i| < \beta$, $1 \leq \beta \leq k$.

Suppose p is a sequential pattern, then the partitioning of its equivalent set is processed as follows. At the beginning, the first element of p becomes the first element of p_1. Then, as long as the equivalent set of p_1 has a length smaller than β, the following elements of p are included in p_1. When this condition does not hold, a new subsequence, p_2, is started. The same process continues until all the elements of p have been examined. As an example, given $p = \langle(ab)(c)(d)(af)(b)(e)\rangle$ and $\beta = 10$, the result of partitioning is $p_1 = \langle(ab)(c)(d)\rangle$ and $p_2 = \langle(af)(b)(e)\rangle$. In this case, $|E(p_1)| = 9$ and $|E(p_2)| = 9$. Notice that $|E(p_1)| + |E(p_2)|$ is smaller than $|E| = 32$.

The relation between two partitioned sequential patterns is as follows. Suppose a sequential pattern p is partitioned into $p_1, \ldots, p_k$. Given a sequential pattern q, q is contained in p if there exists a partitioning of q into $q_1, \ldots, q_m$ such that q_1 is contained in p_{i_1}, q_2 is contained in p_{i_2},…, q_m is contained in p_{i_m}, and $i_1 < i_2 < \ldots < i_m$.

6.7 Summary

This chapter has provided a survey of previous work on set retrieval with inverted files and signature files. There are many studies concerning inverted file and signature file techniques. This review focuses on the most relevant topics related to our research. Many interesting issues related to application of signature files, for example wireless broadcasting and filtering (Lee & Lee 1999), indexing and retrieval in OODBs (Lee & Lee 1992, Nørvåg 1999, Chen 2004), and time series indexing (André-Jönsson & Badal 1997), could not be included in this survey. In addition, this chapter has also reviewed the method for sequential pattern retrieval which is based on the method of set retrieval using signature files.

Chapter 7

Retrieval of Discovered Temporal Rules

7.1 Importance of Post-Processing Discovered Rules

Many rule discovery algorithms in data mining generate a large number of rules, often greatly exceeds the size of the underlying database, and only a small fraction of that large volume of rules is of any interest to the user (Imielinski & Virmani 1998). It is generally understood that interpreting the discovered rules to gain a good understanding of the domain is an important phase of the knowledge discovery process. However, when there are a large number of generated rules, identifying and analysing interesting rules become difficult. Providing the user with a list of rules ranked by their confidence and support might not be a good way of organizing the set of rules as this method would overwhelm the user. Besides, not all rules with high confidence and support are interesting. The rules with high confidence and support can fail to be interesting for several reasons, that is, they correspond to prior knowledge or expectation, they refer to uninteresting attributes or attributes combinations, and they are redundant (Klemettinen et al.

1994). Therefore, to be useful, a data mining system must manage the large amount of generated rules by offering flexible tools for further selecting rules.

Several approaches for post-processing discovered association rules have been reported. One approach is to group 'similar' rules among the discovered rules (Toivonen, Klemettinen, Ronkainen, Hatonen & Mannila 1995, Lent, Swami & Widom 1997, Liu et al. 1999, Liu, Hu & Hsu 2000). This approach works well for a moderately large number of rules. However, for a larger number of rules it will produce too many clusters.

A more flexible approach is by identifying the rules that are of special importance to the user through templates or *data mining queries*. This approach can complement the rule grouping approach. Templates have been used to specify interesting and uninteresting classes of rules (association and episode rules) (Klemettinen et al. 1994). The importance of data mining queries has been highlighted by Imielinski and Mannila (1996) by introducing the concept of Inductive Databases, in which Knowledge and Data Discovery Management Systems (KDDMS) manage KDD applications just as DBMSs successfully manage business applications. In addition to allowing the user to query the data, KDDMS also give the users the ability to query patterns, rules, and models extracted from these data.

Several query languages have been proposed in the literature that provide the user the ability to query the data, for example, Mine-Rule (Meo, Psaila & Ceri 1996), DMQL (Han et al. 1996), and OLE DB (Netz, Chaudhuri, Fayyad & Bernhardt 2001). However, these languages are not designed for querying the discovered rules, but for generating rules from the data. A more powerful data mining query language called MSQL is proposed, which can be used not only for rule generation, but also for querying the discovered rules (using SelectRules operator) (Imielinski & Virmani 1999). A comparative study on Mine-Rule, DMQL, OLE DB, and MSQL is presented by Botta *et al.* (2004). They conclude that one of the main limits of the four languages is the weak support of rule post-processing. Their post-processing capabilities are limited to a few predefined built-in primitives. Tuzhilin & Liu (2002) have proposed Rule-QL for querying multiple sets of association rules and developed efficient algorithms for processing the queries.

This chapter deals with the post-processing problem of richer temporal association rules, an area in which little research to date has been conducted. In particular, this chapter focuses on developing a retrieval system that can be utilised for facilitating the selection of interesting rules during the post-processing of a large set of discovered richer temporal association rules. In this chapter, the general framework of the post-processing model is described. This chapter also proposes a query language TAR-QL for specifying the criteria of rules to be retrieved. The query language specification is based on the subset of Rule-QL, extended by adding several additional operators and functions to deal with richer temporal association rules. The major part of this chapter concentrates on developing low-level methods for evaluating queries involving rule format conditions. In order to improve the performance of the methods, the indexing technique based on signature files is proposed.

When processing queries on a small database of rules, sequential scanning of the rules followed by straightforward computations of query conditions is adequate. But as the database grows, this procedure can be too slow, and indexes should be built to speed up the queries. The problem is to determine what types of indexes are suitable for improving queries involving the rule format conditions. As discussed in Chapter 6, signature files have been used for supporting set and sequence retrieval. In richer temporal association rules, the antecedent and consequent of a rule are normalized temporal patterns, which can be considered as an extended form of sequences. Therefore, signature files are a suitable indexing methods for the rules.

The rest of the chapter is organized as follows. Section 7.2 defines terms used in the chapter. Section 7.2 describes the general framework of the post-processing model. Section 7.4 defines basic queries that can be applied to the database of rules. Section 7.5 describes the method to construct the signature files. In Section 7.6, methods for processing the queries with signatures files are developed. Section 7.7 presents experiment results of the proposed methods.

7.2 Definitions

The rule set, called *rulebase*, contains richer temporal association rules generated from the database of interval sequences using ARMADA described in Chapter 5. Let R be a rulebase, then R contains the rules of the form $\beta \rightarrow \alpha$, where α and β are normalized temporal patterns such that $\beta \sqsubset \alpha$ (see Definition 5.7). Each rule is associated with at least the support and confidence values. In this chapter, a temporal pattern β is called LHS (left-hand side) of the rule, while α is RHS (right-hand side) of the rule.

It was mentioned in Chapter 4.3 that a normalized temporal pattern does not contain temporal extensions because it has been abstracted from the time intervals in a specific interval sequence. Normalized temporal patterns are defined over a set of state S and a set of interval relations {*equals (=), contains (c), is-finished-by (fi), starts (s), overlaps (o), meets (m), before (b)*}. Let Rel = {=, *c, fi, s, o, m, b*} be the set of these seven relations. As an example, Figure 7.1 presents four normalized temporal patterns and two richer temporal association rules generated from the patterns. It is assumed that the temporal patterns are defined over a set of states $S = \{A, B, C, D\}$.

7.3 Framework of the Post-processing

The KDD process and methodology discussed in Fayyad *et al.* (1996) underline the importance of interaction and iteration. Typically, the iterative process occurs between the discovery and presentation phases, as shown in Figure 7.2(a).

$$p_1 = \begin{array}{cc} A & B \\ \hline \left[\begin{array}{cc} = & b \\ * & = \end{array}\right] \end{array} \qquad p_2 = \begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \left[\begin{array}{cc} = & o \\ * & = \end{array}\right] \end{array}$$

$$p_3 = \begin{array}{c} \begin{array}{ccc} A & B & D \end{array} \\ \left[\begin{array}{ccc} = & b & b \\ * & = & m \\ * & * & = \end{array}\right] \end{array} \qquad p_4 = \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \left[\begin{array}{cccc} = & o & b & b \\ * & = & b & b \\ * & * & = & c \\ * & * & * & = \end{array}\right] \end{array}$$

r_1: $p_1 \rightarrow p_3$ [sup = 0.60, conf = 0.80]

r_2: $p_2 \rightarrow p_4$ [sup = 0.75, conf = 0.75]

Figure 7.1: Example of temporal patterns and temporal rules

This chapter adopts a different approach introduced by Klementtinen (1996) and illustrated in Figure 7.2(b). This approach puts the emphasis of the iteration process on the presentation phase. Two reasons to justify this approach are mentioned in Klementtinen (1996). First, in exploratory data analysis, it can be hard to specify beforehand what is considered interesting. If the goal is to discover previously unknown information, this can only be achieved by iterative exploration. Second, pattern generation is often time consuming. Although many efficient algorithms exist, the pattern discovery phase is still a barrier for smooth interaction.

The framework has been used to support interactive exploration of large collection of association and episode rules generated from telecommunication alarm databases (Klemettinen et al. 1996). The approach uses templates to make a selection from a rulebase, as well as to eliminate a selection from consideration. In Tuzhilin and Adomavicius (2002), the use of templates is extended to the biological domain to address the problem of analysing very large numbers of discovered rules in the generated from microarray data. When there are multi-

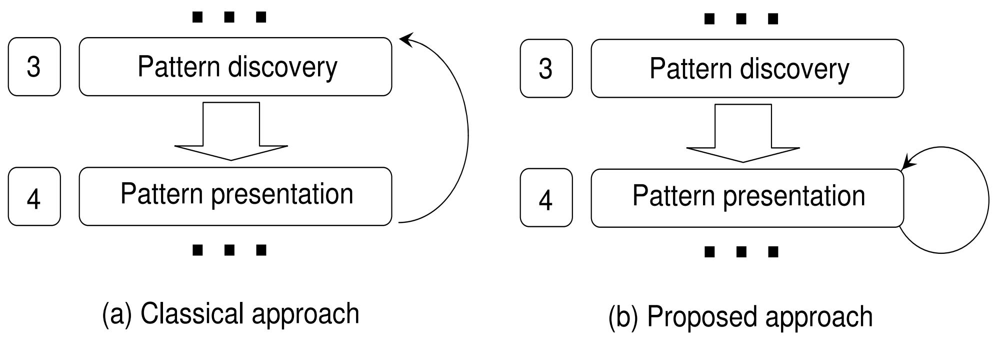


Figure 7.2: Classical vs proposed approaches

ple sets of discovered rules that have been generated over several time periods, this framework can be employed for analysing the behavior of rules (Baron & Spiliopoulou 2001, Liu, Ma & Lee 2001, Tuzhilin & Liu 2002).

In the context of richer temporal association rules, the framework is schematically shown in Figure 7.3. As shown in the figure, the framework puts the emphasis on the rule presentation phase where interesting rules iteratively can be found without repeating rule discovery phase that is more time consuming. The rulebase is generated at once with ARMADA, and the iteration is performed mostly in the presentation phase through the retrieval system that provides facilities for selecting interesting rules.

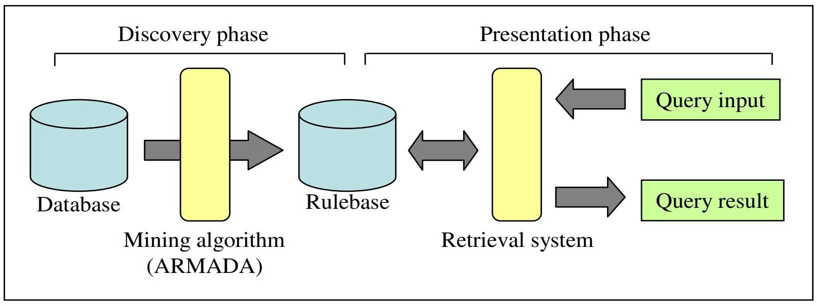


Figure 7.3: The post-processing framework

In order to be useful for a wide range of rule retrieval scenarios, the retrieval system should provide flexible accesses to the rulebase. Although the templates are useful for specifying the criteria of association rules to be retrieved, they are not fully suitable for the richer temporal association rules, due to the complex format of the rules. As an alternative, this chapter considers the use of query languages for selecting the rules. It was mentioned above that although several data mining query languages have been proposed, most of them are designed for rule mining and only a few provide enough support of rule retrieval. In contrast to this trend, the query language required in this framework is only to facilitate the retrieval of rules from the rulebase and is never concerned with the mining data itself. Rule-QL (Tuzhilin & Liu 2002) is one of the languages satisfying this requirement.

However, since Rule-QL is specifically designed for association rules, it must be extended to deal with richer temporal association rules. Therefore, in this chapter, a query language TAR-QL is proposed as a query specification language for selecting richer temporal association rules. TAR-QL is based on the subset of Rule-QL, extended by adding several additional operators and functions to deal with richer temporal association rules. The complete language specification of TAR-QL is given in Appendix C. However, for the purpose of discussing the proposed indexing method that supports the query evaluation, the structure of a basic TAR-QL query together with examples are presented in Section 7.4 below.

7.4 Types of Queries on Temporal Rules

This section describes basic queries that can be applied to the rulebase. As illustration the queries are expressed in SQL-like expressions of the proposed TAR-QL language. The syntax of the basic queries is:

```
SELECT r
FROM Rulebase r
WHERE Conditional_Expression
```

The FROM clause specifies the rulebase over which the queries are performed, r is the rule variable defined over the rulebase, and the SELECT clause selects the variable that satisfies the condition of the WHERE clause. Three types of simple conditions after the WHERE clause are *pruning conditions*, *rule state conditions*, and *rule format conditions*.

Pruning conditions

These conditions enable the user to express queries involving attributes of rules, such as support, confidence, and size of rules. These could have the format:

```
Rule_funct(r) <rel_op> const,
```

where *Rule_funct* is one of the functions defined on a rule such as SUP, CONF, SIZE_LHR, and SIZE_RHS defining support, confidence, and the size of a rule (the number of states in the rule). The *rel_op* is one of the relational operators $\leq$, $\geq$, $<$, $>$, $=$, and *const* is a constant.

Query 7.1 Find rules from the rulebase whose support is greater than 50% and confidence is greater than 70%, and whose number of states on the RHS of the rule is fewer than 4.

```
SELECT r
FROM Rulebase r
WHERE SUP(r) > 50% AND CONF(r) > 70%
      AND SIZE_RHS(r) < 4
```

Using the set of rules in the Figure 7.1, this query results in a rule r_1.

Rule state conditions

The rule state conditions allow the user to specify that a rule's LHS/RHS must contain a given set of states, be a subset of a given set of states, or be equal to a set of given states. These conditions have the format:

```
states_funct(r) <set_op> const,
```

where *state_funct* is either the STATES_LHS or STATES_RHS defining the set of states on the LHS or RHS of the rule. The *set_op* is one of the set operators SUPERSET_OF, SUBSET_OF and EQUAL. The *const* is a set of states. The SUPERSET_OF, SUBSET_OF, and EQUAL operators, respectively, correspond to the subset, superset, and equality queries defined in Definition 6.1 (Chapter 6).

Query 7.2 Find rules from the rulebase whose the RHS of the rule contains a set {A, B, D}.

```
SELECT  r
FROM Rulebase r
WHERE STATES_RHS(r) SUPERSET_OF {A, B, D}
```

The above query results in rules r_1 and r_2 because the right-hand sides of r_1 and r_2, i.e. p_3 and p_4, respectively, contain a set {A, B, D}. When the *state_funct* is used, the LHS/RHS of the rule is considered as a set of states, in which the order of the states and the relationships between states in the LHS/RHS of the rules are ignored.

Rule format conditions

These conditions allow the user to specify that a rule's LHS/RHS must contain a temporal pattern, be contained in a temporal pattern, or be equal to a temporal pattern. They have the format:

```
L|RHS(r) <pat_op> const,
```

where L|RHS(r) is either the LHS or the RHS of the rule r, *pat_op* is one of the temporal pattern containment operators CONTAINS, CONTAINED_IN and EQUAL. The *const* is a temporal pattern.

Query 7.3 Find rules from the rulebase whose the RHS of the rule contains a pattern $\begin{array}{cc} A & B \\ \left[\begin{array}{c} = \\ * \end{array}\right. & \left.\begin{array}{c} b \\ = \end{array}\right] \end{array}$.

```
SELECT r
FROM Rulebase r
WHERE RHS(r) CONTAINS TO_PATTERN(<A, B>, <b>)
```

In this query, TO_PATTERN is a function to convert a pair containing a list of states and a list of relationships to a normalized temporal pattern. This function simplifies the writing of queries involving rule format conditions. In the query above, TO_PATTERN(⟨A, B⟩, ⟨b⟩) represents a pattern $\begin{array}{cc} A & B \\ \left[\begin{array}{c} = \\ * \end{array}\right. & \left.\begin{array}{c} b \\ = \end{array}\right] \end{array}$. This query produces a rule r_1 as a result.

A complex conditional expression can be defined in terms of these simple conditional expressions. Accordingly, the processing of complex conditional expressions relies on that of the simple ones. Generally, queries involving several conditional expressions have many evaluation options and finding an efficient query plan constitutes a significant challenge. This is a problem of query optimisation, which will be considered in the future work. This chapter focuses on the methods for evaluating the simple conditional expressions.

The query containing pruning conditions as in Query 7.1 can be processed more efficiently by using B+ trees as indexes on the supports or confidences of the rules in the rulebase. On the other hand, the query involving rule state conditions as shown in Query 7.2 contains comparisons between sets. For example, to solve Query 7.2, the set of states in the RHS of rules needs to be checked if it contains a constant set {A, B, C}. In order to speed up this process, inverted files or signature files described in Chapter 6 can be utilized to index the LHS and/or RHS of the rules according to the needs.

In contrast, the evaluation of queries involving rule format conditions as in Query 7.3 requires comparisons between temporal patterns on the LHS/RHS of the rules against a constant pattern. A Temporal pattern not only contains a set of states, but also a set of relationships between states corresponding to the ordering of states within the pattern. Therefore, inverted file or signature file access methods cannot be directly applied to the temporal patterns, because these set-oriented techniques do not consider the ordering of states.

Since B+ trees are straightforward and set-oriented methods of inverted files and signature files have been described in Chapter 6, the remainder of this chapter focuses on developing the index scheme to efficiently evaluate the containment predicates CONTAINS, CONTAINED_IN and EQUAL.

In order to simplify the discussion and without loss of generality, the problem of finding rules that satisfy the rule format conditions is reformulated as the problem of finding temporal patterns that satisfy the following queries.

Definition 7.1 (Content-based queries on temporal patterns) Let $\mathcal{D}$ be a temporal pattern database, and q be a query pattern. The three types of queries on temporal pattern are:

1. **Subpattern queries**: find those patterns in $\mathcal{D}$ that contain q, that is, all $p \in \mathcal{D}$ such that $p \sqsupseteq q$.
2. **Superpattern queries**: find those patterns in $\mathcal{D}$ that are contained in q, that is, all $p \in \mathcal{D}$ such that $p \sqsubseteq q$.
3. **Equality queries**: find those patterns in $\mathcal{D}$ equal to q, that is, all $p \in \mathcal{D}$ such that $p = q$.

In this new problem formulation, a database $\mathcal{D}$ represents a set of temporal patterns from LHS/RHS of the rules, and a query pattern q represents a constant pattern that comes after the containment operators. The subpattern, superpattern, and equality queries correspond to operators CONTAINS, CONTAINED_N, and EQUAL, respectively. In order to speed up the query evaluation, the signature files are used to index the target patterns stored in $\mathcal{D}$. Section 7.5 describes the method to construct signature files from this collection of temporal patterns. In Section 7.6, methods for evaluating the queries using the signature files are developed.

7.5 Constructing Signature Files

Signature files store the signatures of temporal patterns. The signature of a temporal pattern is created by converting the temporal pattern into an *equivalent set* from which the signature is then generated. The idea is based on the previous work (Morzy et al. 2001, Nanopoulos et al. 2003, Zakrzewicz 2001) in which the signature of a sequential pattern is generated by first converting the sequential pattern into its equivalent set (see Chapter 6.6).

7.5.1 Converting Temporal Patterns to Equivalent Sets

Two functions are required to create the equivalent set of a temporal pattern. The first function is used to map a set of states in the pattern into a set of integers, while the second one maps the relationships between states into integers. These two functions are defined in the following.

Definition 7.2 (State mapping) Given a set of states S, a *state mapping* function $f(x)$ is a function which transforms a state type $x \in S$ into an integer value, such that $f(x) \neq f(y)$ for $x \neq y$, where $x, y \in S$.

Let a set of states $S = \{A, B, C, D\}$. An example of a simple state mapping function $f(x)$ can be defined as $f(A) = 1$, $f(B) = 2$, $f(C) = 3$, and $f(D) = 4$. This function maps each state into a unique value.

Definition 7.3 (Relationship mapping) Given a set of states S and a set of relations Rel, a *relationship mapping* $g(x, y, \mathbf{r})$ is a function which transforms a relationship $(x\ \mathbf{r}\ y)$ into an integer value, where $x, y \in S$, and $\mathbf{r} \in Rel$.

It is desirable to have a unique mapping for each relationship. However, designing such mapping is not trivial. Therefore, it is sufficient to have a function g such that $g(x, y, \mathbf{r}) \neq g(y, x, \mathbf{r})$ for any $x, y \in S$, $\mathbf{r} \in Rel$. Consider a function $g(x, y, \mathbf{r}) = h(\mathbf{r}) \cdot f(x) + f(y)$, where f is a state mapping function, and h is a function that maps $\mathbf{r} \in Rel$ into an integer. Let define $h(\mathbf{r})$ as: $h(=) = N$, $h(c) = 2N$, $h(fi) = 3N$, $h(s) = 4N$, $h(o) = 5N$, $h(m) = 6N$, and $h(b) = 7N$, where N is the number of states in S. Using this definition of h, g will have the property that $g(x, y, \mathbf{r}) \neq g(y, x, \mathbf{r})$ for any $x, y \in S$, $\mathbf{r} \in Rel$. For example, if $S = \{A, B, C, D\}$ and f is defined above, then $g(A, B, b) = (28 \times 1) + 2 = 30$, and $g(B, A, b) = (28 \times 2) + 1 = 57$, which results in $g(A, B, b) \neq g(B, A, b)$.

Having defined mapping functions f and g, the equivalent set of a temporal pattern can be defined using these two functions.

Definition 7.4 (Equivalent set) Given a temporal pattern p of size k, $S_p = \langle s_1, \ldots, s_k \rangle$ is the list of states in p, and M_p is a $k \times k$ matrix whose element $M_p[i, j]$ denotes the relationship between states s_i and s_j in S_p. The equivalent set of p, $E(p)$, is defined as:

$$E(p) = \left(\bigcup_{i=1}^{k} \{f(s_i)\} \right) \bigcup \left(\bigcup_{i=1}^{k-1} \bigcup_{j=i+1}^{k} \{g(s_i, s_j, \mathbf{r})\} \right)$$

where $\mathbf{r} = M_p[i, j]$.

For example, using the mapping functions f and g in the previous example, the equivalent set of patterns p_1 and p_3 (Figure 7.1) can be computed as follows:

$$\begin{aligned} E(p_1) &= \{(f(A)\} \cup \{f(B)\} \cup \{(g(A, B, b)\} = \{1, 2, 30\} \\ E(p_3) &= \{(f(A)\} \cup \{f(B)\} \cup \{f(D)\} \cup \{(g(A, B, b)\} \cup \\ &\quad \{(g(A, D, b)\} \cup \{(g(B, D, m)\} \\ &= \{1, 2, 4, 30, 32, 52\} \end{aligned}$$

Table 7.1: Equivalent sets and signatures of temporal patterns

Pattern	Equivalent set	Signature
p_1	$\{1, 2, 30\}$	0100 0110
p_2	$\{1, 2, 22\}$	0100 0110
p_3	$\{1, 2, 4, 30, 32, 52\}$	0101 0111
p_4	$\{1, 2, 3, 4, 22, 31, 32, 59, 60, 28\}$	1101 1111

Equivalent sets of the other temporal patterns are shown in the second column of Table 7.1.

It can be observed that if a temporal pattern is a subpattern of another pattern then the equivalent set of the first pattern is a subset of the second pattern's equivalent set. For example, p_1 is a subpattern of p_3, therefore, $E(p_1) \subseteq E(p_3)$ (Table 7.1). This property is formalized in the following.

Property 7.1 Given two temporal patterns p, q, and the corresponding equivalent sets $E(p)$ and $E(q)$, the following properties hold for any two temporal patterns and their equivalent sets:

1. $p \sqsupseteq q \rightarrow E(p) \supseteq E(q)$
2. $p \sqsubseteq q \rightarrow E(p) \subseteq E(q)$
3. $p = q \rightarrow E(p) = E(q)$

7.5.2 Converting Equivalent Sets to Signatures

Using the superimposed coding method, the signature of an equivalent set E is an F-bit binary number created by bit-wise union (OR) of all element signatures in E. Each element signature has F-bit length and m-bits are set to '1'. For example, given $F = 8$ and $m = 1$, the signature of element $e \in E$ is an 8-bit binary number that can be computed by a hash function $hash(e) = 2^{(e \bmod F)}$. For the set $E(p_3) = \{1, 2, 4, 30, 32, 52\}$, its element signatures are $hash(1) = 00000010$, $hash(2) = 00000100$, $hash(4) = 00010000$, $hash(30) = 01000000$, $hash(32) =$

00000001, and $hash(52) = 00010000$. The signature of $E(p_3)$ is computed using the bit-wise union of all these element signatures, and the resulting signature is '01010111'. Using the same method, the signatures of the other temporal patterns are shown in the third column of Table 7.1.

Property 7.2 Given two equivalent sets $E(p)$ and $E(q)$, and their corresponding signatures sig_p and sig_q, the signatures of equivalent sets have the following properties:

1. $E(p) \supseteq E(q) \rightarrow sig_p \wedge sig_q = sig_q$
2. $E(p) \subseteq E(q) \rightarrow sig_p \wedge sig_q = sig_p$
3. $E(p) = E(q) \rightarrow sig_p = sig_q$

Combining Properties 7.1 and 7.2, the relations between temporal patterns and their signatures are expressed in the following properties.

Property 7.3 Given two equivalent sets p and q, and their corresponding signatures sig_p and sig_q, these signatures have the following properties:

1. $p \sqsupseteq q \rightarrow sig_p \wedge sig_q = sig_q$
2. $p \sqsubseteq q \rightarrow sig_p \wedge sig_q = sig_p$
3. $p = q \rightarrow sig_p = sig_q$

As an example, consider temporal patterns p_1 and p_3, where $p_1 \sqsubseteq p_3$. It can be seen from Table 7.1 that $sig_{p_1} \wedge sig_{p_3} = 01000110 \wedge 01010111 = 01000110$, which is the value of sig_{p_1}.

Using these methods, the signature file of temporal patterns in database $\mathcal{D}$ can be created as follows. For each temporal pattern $p \in \mathcal{D}$, its equivalent set $E(p)$ is calculated, then its signature E_p is generated. This signature, together with the temporal pattern identifier (pid), is inserted into the signature file. The actual insertion depends on the signature file organization. For example, for SSF the signature is appended to the end of the file, while for BSSF each signature bit is appended to the end of the corresponding bit-slice file. Only the signatures are stored in the signature file, while the equivalent sets are only to facilitate the computation of signatures. This procedure is outlined in Algorithm 7.1.

Input: A database $\mathcal{D}$ of temporal patterns
Output: SignatureFile

```
1: for each $p \in \mathcal{D}$ do
2:     $E(p) = \text{Equivalent\_Set}(p)$
3:     $sig_p = \text{Signature}(E(p))$
4:     Insert $\langle sig_p, pid_p \rangle$ into SignatureFile
5: end for
6: return SignatureFile
```

Algorithm 7.1: Constructing a signature file of temporal patterns

7.6 Processing of Queries using Signature Files

This section describes the processing of subpattern, superpattern, and equality queries using signature files. In addition, it defines a similarity measure of temporal patterns that can be used to limit the query result and to cluster the rules.

Similar to the processing of set-based queries with signature files, the processing of temporal pattern queries also consists of two steps, filtering and false drop resolution steps. The process is depicted in Figure 7.4 (adopted from Zezula *et al.* (1991)).

7.6.1 Subpattern Queries

Given a temporal pattern database $\mathcal{D}$ and a query pattern q, the algorithm for evaluating subpattern queries is called *evaluateSubPattern*($\mathcal{D}$, q), which finds temporal patterns in $\mathcal{D}$ that contain q. If the signatures are stored in SSF, *evaluateSubPattern*($\mathcal{D}$, q) is presented in Algorithm 7.2. The equivalent set, $E(q)$, of q is first calculated, and then a query signature sig_q is formed. Each target signature sig_p in SSF is then examined against the query signature sig_q. If the target signature satisfies the search condition $sig_p \wedge sig_q = sig_q$ (the first

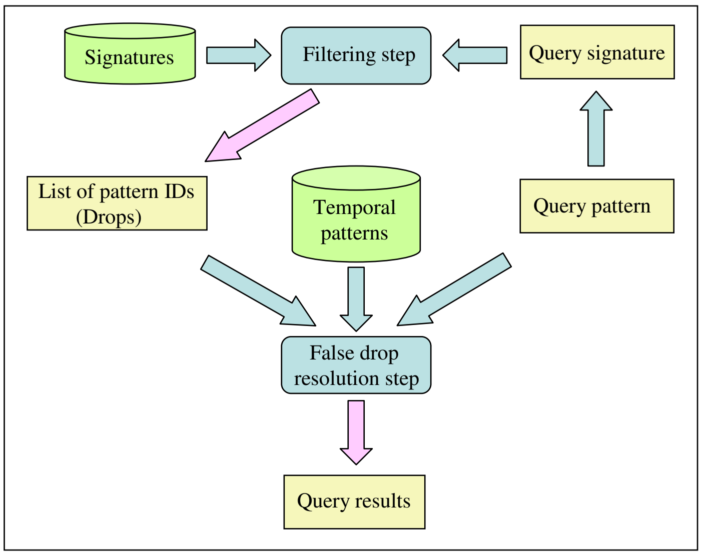


Figure 7.4: Processing temporal pattern query using signature files

property in Property 7.3), the corresponding temporal pattern becomes a drop and its identifier is added to the pattern-id (or PID)-list. Then, during *false drop verification*, each drop is checked to determine if it actually satisfies the query condition.

On the other hand, if the signatures are stored in BSSF, *evaluateSubPattern*($\mathcal{D}$, q) proceeds in a slightly different way, and is shown in Algorithm 7.3. The search condition $sig_p \wedge sig_q = sig_q$ cannot be used, since the target signature sig_p is scattered across bit-slice files. Instead, the bit-slices corresponding to the bit positions set to '1' in sig_q are retrieved, and then a bit-wise intersect (bit-wise AND) operation is performed on the retrieved bit-slices. The corresponding temporal pattern becomes a drop if the resulting bit entry is equal to '1', and its identifier is added to the PID-list. The false drop resolution step is the same as in SSF.

7.6.2 Superpattern Queries

Let *evaluateSuperPattern*($\mathcal{D}$, q) be the algorithm for evaluating superpattern queries, that is, for finding temporal patterns in $\mathcal{D}$ that are contained in q. If SSF

```
Input: Temporal pattern database $\mathcal{D}$, a query pattern $q$
Output: AnswerSet
 1: $E(q)$ = Equivalent_Set($q$)
 2: $sig_q$ = Signature($E(q)$)
 3: for each $\langle sig_p, pid_p \rangle \in SSF$ do
 4:    if $sig_p \wedge sig_q = sig_q$ then
 5:       Add $pid_p$ into PID-list
 6:    end if
 7: end for
 8: for each $pid_p$ in PID-list do
 9:    Retrieve $p$ from $\mathcal{D}$
10:    if $p \sqsupseteq q$ then
11:       Add $p$ into AnswerSet
12:    end if
13: end for
14: return AnswerSet
```

Algorithm 7.2: Pseudo code of evaluateSubPattern using SSF

```
Input: Temporal pattern database $\mathcal{D}$, a query pattern $q$
Output: AnswerSet
 1: $E(q)$ = Equivalent_Set($q$)
 2: $sig_q$ = Signature($E(q)$)
 3: Retrieve the bit-slices corresponding to the bit position set to '1' in $sig_q$
 4: Perform bit-wise intersect operation on the retrieved bit-slices
 5: for each entry where '1' is set in the resulting intersect bit slice do
 6:    Add the corresponding $pid_p$ into PID-list
 7: end for
 8: for each $pid_p$ in PID-list do
 9:    Retrieve $p$ from $\mathcal{D}$
10:    if $p \sqsupseteq q$ then
11:       Add $p$ into AnswerSet
12:    end if
13: end for
14: return AnswerSet
```

Algorithm 7.3: Pseudo code of evaluateSubPattern using BSSF

is used, the algorithm is similar to *evaluateSubPattern* in Algorithm 7.2 except the search condition $sig_p \wedge sig_q = sig_q$ (line 4) is replaced with $sig_p \wedge sig_q = sig_p$, and the query condition $p \sqsupseteq q$ (line 10) is replaced with $p \sqsubseteq q$.

If BSSF is used, the *evaluateSubPattern* retrieves bit-slices corresponding to the bit positions set to '0' in sig_q, and performs bit-wise union (bit-wise OR) operation on them. The corresponding temporal pattern becomes a drop if the resulting bit entry equals '0'. Each drop is then validated with respect to the query condition $p \sqsubseteq q$. The pseudo code of *evaluateSuperPattern* on BSSF is shown in Algorithm 7.4.

```
Input: Temporal pattern database $\mathcal{D}$, a query pattern $q$
Output: AnswerSet
 1: $E(q)$ = Equivalent_Set($q$)
 2: $sig_q$ = Signature($E(q)$)
 3: Retrieve the bit-slices corresponding to the bit position set to '0' in $sig_q$
 4: Perform bit-wise union operation on the retrieved bit-slices
 5: for each entry where '0' is set in the resulting union bit slice do
 6:    add the corresponding $pid_p$ into PID-list
 7: end for
 8: for each $pid_p$ in PID-list do
 9:    Retrieve $p$ from $\mathcal{D}$
10:    if $p \sqsubseteq q$ then
11:       Add $p$ into AnswerSet
12:    end if
13: end for
14: return AnswerSet
```

Algorithm 7.4: Pseudo code of evaluateSuperPattern using BSSF

7.6.3 Equality Queries

Let *evaluateEquality*$(\mathcal{D}, q)$ be the algorithm for processing equality queries. Using SSF, the algorithm follows the Algorithm 7.2, except that the search condition $sig_p \wedge sig_q = sig_q$ is replaced with $sig_p = sig_q$, and the query condition $p \sqsupseteq q$ is replaced with $p = q$.

When BSSF is used, the algorithm requires accesses to all bit-slice files, not only part of them. In order to decide if $sig_p = sig_q$, each bit of sig_q must be compared with corresponding bit of sig_p which is stored in different bit-slice files. This is only possible by accessing all bit-slice files.

7.6.4 Temporal Pattern Similarity

This subsection defines similarity measure for the normalized temporal patterns. To my knowledge, no similarity measures have been defined for the patterns. In order to define the similarity measure of the patterns, three properties must be considered:

1. Temporal patterns are variable-length objects that cannot be represented in a k-dimensional metric space,
2. Each pattern contains a list of states, and
3. Each pattern contains a set of state relationships.

The similarity measure is defined based on the measure known in the literature as the *Jaccard coefficient* of two sets, which expresses the fraction of elements of common to both sets. The Jaccard coefficient is not a metric[1]. Nevertheless, a distance function can be defined in terms of the similarity as *d(A,B)* = 1 - *sim(A,B)*, and it is easy to show that such a distance function is indeed a metric.

Given two temporal pattern α and β, let S_α and S_β denote a set of states in α and β, respectively. The similarity between patterns α and β is defined in Equation (7.1).

$$sim(\alpha,\beta) = \frac{|S_c| + |R_c|}{\sqrt{(N^s_\alpha + N^r_\alpha) \times \left(N^s_\beta + N^r_\alpha\right)}} \tag{7.1}$$

In the equation (7.1), $|S_c| = |S_\alpha \cap S_\beta|$ is the number of common states in α and β, and $|R_c|$ represents the number of common relationships. N^s_α represents

[1]metric is a function *m(.,.)* which is non-negative, symmetry, *m(x,y)* = 0 *iff* $x = y$, and satisfies the triangle inequality.

the number of states (size) of α, and N^r_α represents the number of relationships in α. For a temporal pattern α of size n, $N^s_\alpha = n$ and $N^r_\alpha = \frac{n(n-1)}{2}$. The value of $sim(\alpha, \beta)$ will be 1 if $\alpha = \beta$, but it will be 0 if they do not have common states.

As an example, consider temporal patterns p_2 and p_4 in Figure 7.1. The sets $S_{p_2} = \{A, B\}$ and $S_{p_4} = \{A, B, C, D\}$, so $|S_{p_2} \cap S_{p_4}|$ (the number of common states) $= 2$. The patterns only have 1 common relationship, that is, the relationship *(A before B)*. The value of $N^s_{p_2} = 2$, $N^r_{p_2} = 1$, $N^s_{p_4} = 4$, and $N^r_{p_4} = 6$. Therefore, the similarity between patterns p_2 and p_4 can be computed as:

$$sim(p_2, p_4) = \frac{2+1}{\sqrt{(2+1) \times (4+6)}} \approx 0.548$$

By using Equation (7.1), the similarity matrix of four temporal patterns in Figure 7.1 is:

$$SIM = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \begin{array}{c} \begin{array}{cccc} p_1 & p_2 & p_3 & p_4 \end{array} \\ \begin{bmatrix} 1 & 0.667 & 0.707 & 0.365 \\ 0.667 & 1 & 0.471 & 0.548 \\ 0.707 & 0.471 & 1 & 0.516 \\ 0.365 & 0.548 & 0.516 & 1 \end{bmatrix} \end{array}$$

Let $d(x, y)$ be a distance between temporal patterns x and y defined as $d(x, y) = 1 - sim(x, y)$. The distance function d is a metric, because it has the following properties:

1. Non-negative: $d(x, y) \geq 0$.
2. Symmetry: $d(x, y) = d(y, x)$.
3. Identity: $d(x, y) = 0$ iff $x = y$.
4. Triangle inequality: $d(x, y) \leq d(x, z) + d(z, y)$.

This measure can used to limit the number of rules resulted from the query processing. For example, given that a subpattern query generates n temporal patterns containing the query pattern q, the *k-nearest subpattern query* can choose

Table 7.2: Parameters

Symbol	Definition
F	Size of signature (in bits)
m	Weight of a signature element
N	Number of states
$\|D_s\|$	Size of sequence database
$\|C\|$	Average size of sequences
$\|D\|$	Size of temporal pattern database
$\|T\|$	Average size of temporal patterns
Q	Size of a query pattern

k of n temporal patterns that are most similar to q, where $k < n$. In addition, it can also be utilised as a distance function in the temporal rule clustering.

7.7 Experiments

To assess the performance of the proposed methods, the *evaluateSubPattern*, *evaluateSuperPattern*, and *evaluateEquality* were implemented on SSF and BSSF signature files. In addition, sequential versions of the methods (SEQ) were also implemented as baseline methods, which process queries by sequentially retrieving the target pattern (without using index) from the database and comparing it against the query pattern. All programs are written in Java Language. The experiments were conducted on synthetic datasets on an 2.4GHz Athlon PC with 512MB of RAM running Windows 2000 Professional.

The following subsections show the performance of *evaluateSubPattern* and *evaluateSuperPattern* for processing subpattern and superpattern queries, respectively. Update cost is not considered as it is assumed that the index is only created once after frequent patterns have been generated by a data mining process. Experimental parameters are listed in Table 7.2.

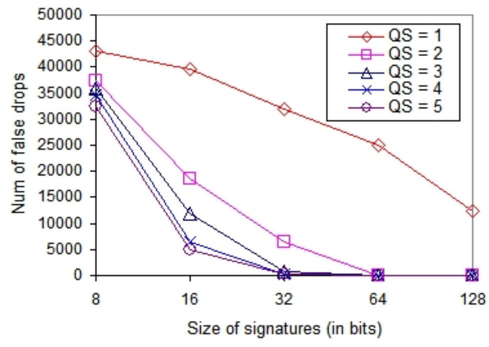
The temporal pattern database was generated using ARMADA from an interval sequence database D_s containing 10,000 interval sequences ($|D_s|$), with average length of 10 ($|C|$), and 100 different types of state (N). Using the minimum support of 0.08% and maximum gap of 200, ARMADA generated a set of 106,409 frequent temporal patterns. A temporal pattern database D is populated from this set of temporal patterns. First, the size of temporal pattern t was determined randomly from a Poisson distribution with mean equal to $|T|$. Then, a temporal pattern of size t is randomly picked from the set of frequent temporal patterns, and added it to the database D.

To guarantee that the evaluated queries do not return an empty result set, query patterns were generated as follows. For subpattern queries, a temporal pattern of size 5 with the highest support in D was selected, then the states from the pattern starting from the last state were individually removed, resulting in a set of five queries. A similar method is performed for superpattern queries by selecting a temporal pattern of size 10 to generate a further set of five queries.

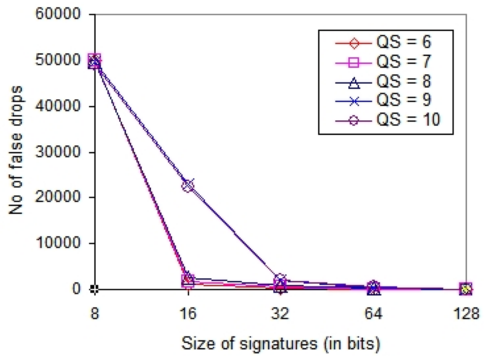
7.7.1 Effect of Signature Size on the Number of False Drops

This experiment observed how the size of the signature affects the number of false drops in *evaluateSubPattern* and *evaluateSuperPattern*. It also determined the optimal parameters for each query type in the experimental environment, particularly the values of F and m. As can be seen from Equation (6.3) of Chapter 6, the false drop probability depends on F, m, and the cardinalities of the query set and target set. The value of m was set to 1 and the value of F increased until no further performance improvement could be perceived. The size of the database $|D| = 50,000$, the average size of temporal pattern $|T| = 5$, and the number of states $N = 100$. The size of signature F was varied from 8 to 128 bits. The number of false drops was measured.

Figures 7.5(a) and 7.5(b) show the number of false drops for *evaluateSubPattern* and *evaluateSuperPattern*, respectively. The number of false drops is similar for both SSF and BSSF, since it does not depend on the signature file structures.



(a) evaluateSubPattern



(b) evaluateSuperPattern

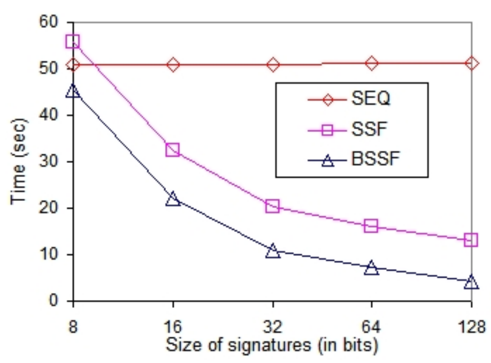
Figure 7.5: Effect of signature size on the number of false drops

As can be seen, the number of false drops consistently decreases as the size of the signature increases and the number of false drops is also influenced by the size of the query pattern. For *evaluateSubPattern* (Figure 7.5(a)), the larger the query pattern, the lower the number of false drops. Conversely, the larger the query pattern, the higher the number of false drops for *evaluateSuperPattern* (Figure 7.5(b)). The best recorded performance improvement were achieved with a signature size between 16 and 32 bits, at which point the number of false drops decreases significantly.

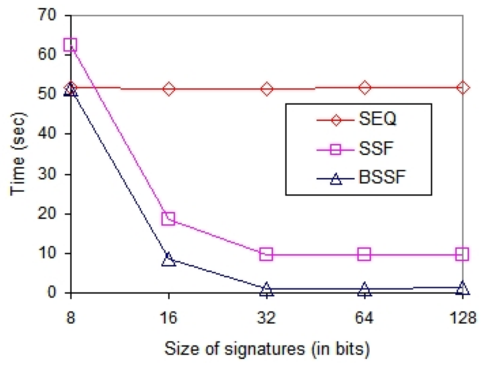
7.7.2 Effect of Signature Size on Query Processing Time

This experiment used the above dataset to compare the relative performance of the *evaluateSubPattern* and *evaluateSuperPattern* on SEQ, SSF, and BSSF. Each method was run on each of the queries used in the previous experiment. Figure 7.6(a) shows the total time required by *evaluateSubPattern*, while Figure 7.6(b) shows the total time required by *evaluateSuperPattern*.

The figures show that the query processing times of both methods on SSF and BSSF are proportional with the number of false drops from previous experiments. Both methods gain the best performance improvement when the size of signature is between 16 and 32. Both methods perform better on BSSF. For small values of

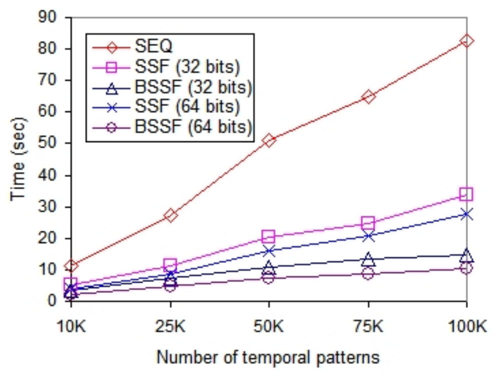


(a) evaluateSubPattern

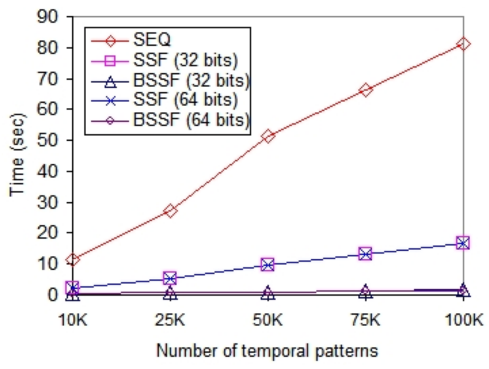


(b) evaluateSuperPattern

Figure 7.6: Effect of signature size on query processing time



(a) evaluateSubPattern



(b) evaluateSuperpattern

Figure 7.7: Effect of database size on query processing time

F, the query processing times of both methods are almost similar on SEQ, SSF, and BSSF. This is because when $F \leq 8$ the number of false drops becomes so high that when SSF and BSSF are used, the algorithms have to retrieve almost all patterns in the database during the verification step. Finally, both methods show a marked improvement on SSF and BSSF over SEQ.

7.7.3 Effect of Database Size on the Query Processing Time

In order to observe how the methods scale with respect to the database size, five datasets were generated in which $|T| = 5$, and $N = 100$. The size of database $|D|$ was varied from 10,000 to 100,000. Both methods were run on each dataset using two values of F (32 and 64 bits). Figure 7.7(a) shows the total time required by *evaluateSubPattern* on SEQ, SSF 32 bits, SSF 64 bits, BSSF 32 bits, and BSSF 64 bits to process five queries. Figure 7.7(b) shows the total time required by *evaluateSuperPattern* to process five queries.

In general, the processing times are proportional to the database size. Both methods remain the slowest on SEQ, but show the fastest or the best scaling behaviour on BSSF.

7.8 Summary

This chapter has presented the retrieval system, which facilitates the selection of interesting rules in the post-processing of discovered richer temporal association rules. The general framework of the post-processing model is described. In order to provide flexible accesses to the rulebase, a query language TAR-QL is proposed for expressing queries on richer temporal association rules. This chapter has developed and implemented low-level techniques for evaluating queries involving rule format conditions. These techniques utilize signature files to improves the performance of the query evaluation. The experiment results show that those that are based on BSSF perform better than the ones that are based on SSF, but both are better than sequential methods.

This chapter does not discuss the presentation of the rules after they have been retrieved. In the post-processing of discovered rules, the presentation of the rules is as important as the rule retrieval. Appropriate rule presentation methods can provide a better understanding of the rules so that the finding of interesting rules can be as easy as possible. Various methods have been proposed for visualising association rules generated by data mining algorithms (Hofmann, Siebes & Wilhelm 2000, Ong, Ong & Lim 2002, Wong, Whitney & Thomas 1999) The application of these methods to richer temporal association rules needs a further investigation, which is the subject of this thesis's future work.

Chapter 8

Conclusions

This thesis has studied the discovery of temporal patterns and rules from sequential data. Numerous previous studies have concentrated on the discovery of patterns in event sequence data, such as the discovery of temporal association rules, sequential patterns, episodes, and periodic patterns. In contrast, despite its promising applications, the pattern discovery from interval sequence data has received little attention, and many issues have not been fully explored. In this thesis, three main issues related to the pattern discovery from interval sequence data have been addressed. The first issue is the problem of defining structure for interesting patterns in interval sequence data. The second is the need for the efficient algorithms for the discovery of patterns in interval sequence data, while the third is the analysis of the discovered patterns to identify potentially interesting patterns.

The results that have been achieved so far include a problem formulation for the discovery of richer temporal association rules, a novel algorithm for discovering the rules, a high-level language for specifying retrieval tasks concerning the discovered rules, and low-level methods for evaluating queries involving rule format conditions. Moreover, this thesis has also made a contribution in the area of pattern discovery from event sequence data, by proposing a new type of rules called inter-transaction relative temporal association rules.

In this chapter, Section 8.1 provides a discussion of the major contributions of this thesis and differences to other work. Section 8.2 provides a discussion of several future research directions for extending this thesis.

8.1 Contributions

Discovery of Temporal Rules

In order to address the problem of defining pattern structure, this thesis has formulated the problem of discovering richer temporal association rules from interval sequence databases. The richer temporal association rules are formulated using the pattern formulation proposed by Höppner (2001). However, our work is different from the work of Höppner, because the richer temporal association rules are mined from a set of interval sequences, instead of a long sequence of intervals.

This thesis deals with the second issue by developing an efficient algorithm, ARMADA, for discovering richer temporal association rules. ARMADA does not require candidate generation, rather it utilizes a simple index to grow longer temporal patterns from the shorter frequent ones. In the process of growing the patterns, ARMADA only considers those client sequences indicated by a current index set, instead of searching on every client sequence in the database. It is true that extra storage is needed to store the index set, in addition to the memory allocated for the database. However, the size of the index set reduces as the prefix pattern to create the index set increases. Moreover, the proposed algorithm requires at most two database scans. When the database is too large to fit into memory, the algorithm divides the database into several partitions and mines each partition. A second pass of the database is then required to validate the true patterns in the database. Additionally, this thesis introduces a *maximum gap* time constraint that can be used to remove insignificant patterns, which in turn can reduce the number of frequent patterns and richer temporal association rules generated by the algorithm.

Although the discussion of richer temporal association rules has been done in the context of interval sequence data from the medical domain, the proposed concept is quite general and is applicable to the interval-based data from other domains. However, the usefulness of richer temporal association rules for representing temporal knowledge requires further investigation, by mining the rules from real datasets.

In the area of pattern discovery from event sequence data, this thesis has introduced a new type of rule called *inter-transaction relative temporal association rules*. In order to discover the rules, a set of frequent relative itemsets are first generated using the algorithm similar to the AprioriAll algorithm. Some modifications to the AprioriAll algorithm have been introduced to improve the performance of the algorithm. However, this work also shows that basing the algorithm on more efficient algorithms, such as the ideas contained in the FP-Growth algorithm (Han, Pei & Yin 2000) or other ideas of improving Apriori-based algorithms proposed by Bodon (2003) could be used instead.

Retrieval of Discovered Temporal Rules

In order to deal with the post-processing problem of richer temporal association rules, this thesis has proposed a retrieval system that can be utilised for facilitating the selection of interesting rules during the post-processing of a large set of discovered richer temporal association rules. As the first step toward realisation of a such retrieval system, the general framework of the post-processing model in which the retrieval system will be employed has been described. The post-processing framework puts the emphasis on the rule presentation step of the KDD process, where interesting rules iteratively can be found, without repeating the rule discovery step that is more time consuming. However, it should be noted that the rule discovery step can still be performed if it is considered necessary.

In addition, this thesis has proposed a query language TAR-QL for specifying the criteria of the rules to be retrieved. TAR-QL has been based on the subset of Rule-QL (Tuzhilin & Liu 2002), extended by adding several additional operators and functions to deal with richer temporal association rules. Furthermore, this thesis has developed low-level methods for evaluating queries involving rule format conditions. In order to improve the performance of the methods, the indexing technique based on signature files has been proposed. Although, the use of signature files as access methods in set and sequence retrieval is not new (see Chapter 6), this thesis has shown that signature files can be extended to support the retrieval of richer temporal association rules.

Finally, this thesis has developed similarity measure for temporal patterns, which is defined based on the *Jaccard coefficient* of two sets. Although the proposed measure itself is not a metric, the distance function derived from it is. This measure can be used to limit the number of rules resulted from the query processing or to cluster the rules based on the similarity of their antecedents or consequents.

8.2 Future Research Directions

The work presented in this thesis points to several directions for future research. One of the directions to be undertaken by the author includes an application to real-world problem domains and enhancements to the interface to facilitate the discovery of temporal patterns and rules. Ideally, a temporal mining algorithms should understand all types of relationship and convention, thus other work that could be considered is the link between relative relationships and the use of either accepted calendars (for example, the work of Hamilton and Randall (2000) and others) and/or mixing references to relative time with those of absolute time.

Another direction is to make the retrieval system more functional in supporting the post-processing of discovered richer temporal association rules. Two areas of extension present themselves. First, the development of TAR-QL parser for syntactic and semantic analysis of the queries. The parser is also used for building the parse tree for the query and calling the appropriate query processing methods. Second, the development of rule presentation component for presenting the rules resulting from the query. The rule presentation should be done in such a way that finding interesting rules is as easy as possible, for example, through rule visualisation. Visualisation has been an important part of data mining because human have remarkable abilities to spot hidden patterns (Shneiderman 1996). Although several methods have proposed for visualising association rules, there has not been a lot of research effort in the data mining community targeted to visualization of temporal patterns or rules.

Another possible direction is to extend TAR-QL, and hence the retrieval system, for querying multiple rulebases. Currently, the retrieval system is designed for the retrieval of rules from one set of discovered rules. When the system has the capability to query multiple rulebases, it can be used for analysing the behavior of the rules over a number of time periods (Liu et al. 2001, Zhao & Liu 2001, Baron & Spiliopoulou 2001). For example, assuming that the discovered rules for each month are stored in a separate rulebase, one type of query is to find the rules that are stable over several months. Another example is to find rules whose confidences and/or support are 'growing' or 'diminishing'. These types of queries are important as data mining is increasingly used in the production mode (Tuzhilin & Liu 2002).

Appendix A

Publications Resulting from This Thesis

The following publications have resulted from material presented within this thesis. Publications 1 relates to material presented in Chapters 2.1 and initial work of Chapter 3. Publication 2 contains early work of Chapter 5. Publication 3 is an an extended version of publication 2 and contains most of the material presented in Chapter 5.

1. Winarko, E. and Roddick, J.F. (2003). Relative Temporal Association Rule Mining. Proc. of the 2nd Australasian Data Mining Workshop (ADM'03), Canberra. Simoff, S. J., Williams, G. J. and Hegland, M., Eds., University of Technology, Sydney. 121-142.

2. Winarko, E. and Roddick, J.F. (2005). Discovering Richer Temporal Association Rules from Interval-based Data. Proc. of the 7th International Conference on Data Warehousing and Knowledge Discovery - DaWaK'05, Copenhagen, Denmark. Lecture Notes in Computer Science, 3589. Tjoa, A. M. and Trujillo, J., Eds., Springer. 315-325[1].

[1]Selected as one of the best papers

3. Winarko, E. and Roddick, J.F. (2007). ARMADA - An Algorithm for Discovering Richer Relative Temporal Association Rules from Interval-based Data. Data and Knowledge Engineering 63. 76-90.

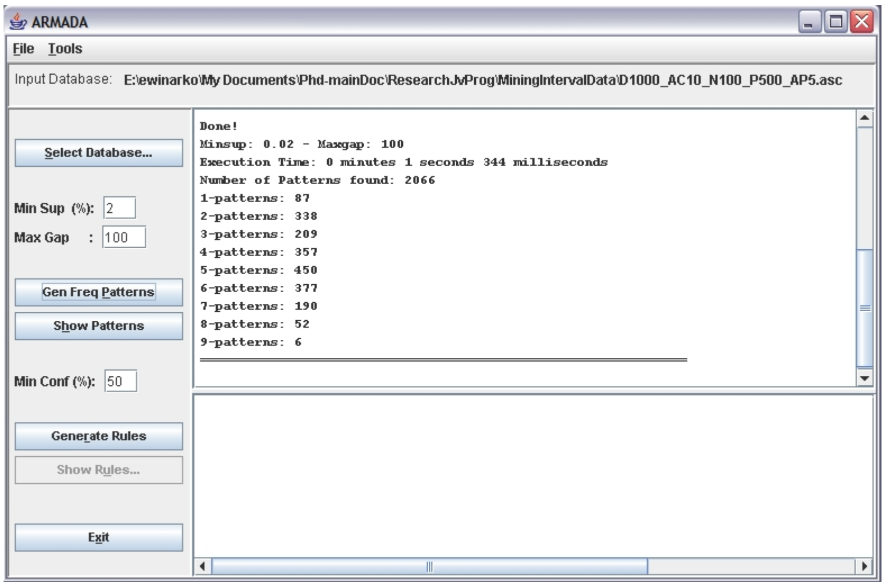
Appendix B

Rule Discovery System

B.1 ARMADA

In order to facilitate the mining of temporal patterns and richer temporal association rules, a simple interface to the ARMADA has been developed. A brief description of the interface and the way it can be used is given below. The interface is shown in Figure B.1 and contains two main areas:

- The left panel contains the controls for selecting input databases, setting the minimum support, maximum gap and minimum confidence, generating and displaying temporal patterns and richer temporal association rules.
- The right panel contains two areas for displaying the generated temporal patterns (top area) and temporal rules (lower area). Initially, the generated temporal patterns are stored in the file and only their summary is displayed. If the user wants to browse the patterns, she/he has to use the 'Show Patterns' control to display the patterns from the file. This is also the case for the temporal rules.



Input Database: E:\ewinarko\My Documents\Phd-mainDoc\Research.JvProg\MiningIntervalData\D1000_AC10_N100_P500_AP5.asc

Min Sup (%): 2

Max Gap : 100

Min Conf (%): 50

```
Done!
Minsup: 0.02 - Maxgap: 100
Execution Time: 0 minutes 1 seconds 344 milliseconds
Number of Patterns found: 2066
1-patterns: 87
2-patterns: 338
3-patterns: 209
4-patterns: 357
5-patterns: 450
6-patterns: 377
7-patterns: 190
8-patterns: 52
9-patterns: 6
```

Figure B.1: Screen shot of the ARMADA interface

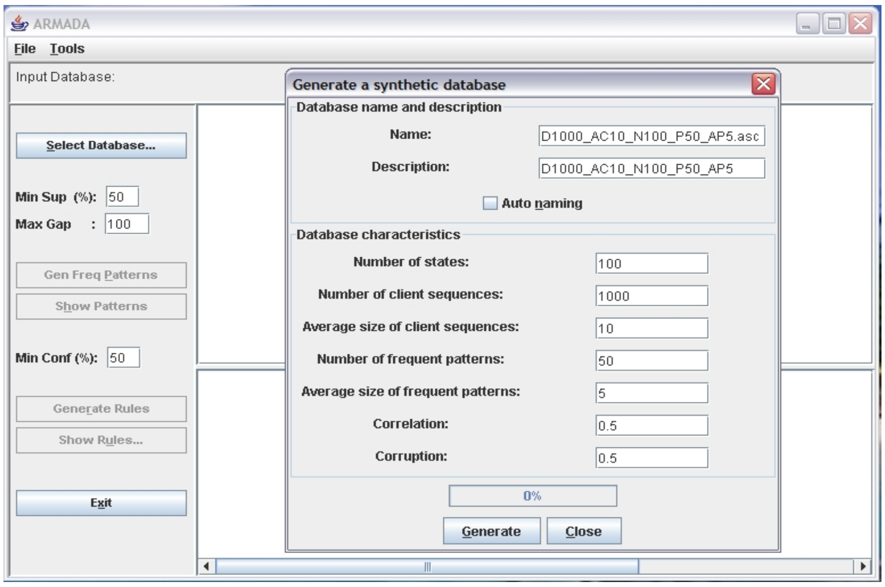
B.2 Interval Data Generator

This sections describes in more detail the synthetic data generation program to generate interval-based datasets used in the experiments in Chapter 5. The generator requires five main parameters shown in table B.1. The other two parameters, the correlation and the corruption, are normally always set to 0.5. The correlation value is the mean correlation between the frequent patterns, and the corruption value is the mean of the corruption coefficient indicating how much a frequent pattern will be corrupted before being used. The generator user interface is shown in Figure B.2.

$\|D\|$	Number client sequences
$\|C\|$	Average size of client sequences
$\|P\|$	Average size of potentially frequent patterns
N_P	Number of potentially frequent patterns
N	Number of states

Table B.1: Parameters

In generating the dataset, the generator first creates a random pool of potentially frequent patterns that will be used in the generation of client sequences.



Input Database:

Min Sup (%): 50
Max Gap : 100
Min Conf (%): 50

Database name and description

Name: D1000_AC10_N100_P50_AP5.asc
Description: D1000_AC10_N100_P50_AP5

Database characteristics

Number of states: 100
Number of client sequences: 1000
Average size of client sequences: 10
Number of frequent patterns: 50
Average size of frequent patterns: 5
Correlation: 0.5
Corruption: 0.5

Figure B.2: Screen shot of the data generator interface

The number of potentially frequent patterns is N_P. A frequent pattern is generated by first picking the size of the pattern (the number of states in the pattern) from Poisson distribution with mean μ equal to $|P|$. The type of states is chosen randomly (from N state types). Then, in order to form a pattern, temporal relations between consecutive states are determined randomly. Assuming that the normalized temporal patterns is used, the temporal relations are chosen from the set $\{$*before, meets, overlaps, is-finished-by, contains, starts, equal*$\}$. Each state in the pattern is then assigned an interval value according to its temporal relation with the state that comes before it. The interval value of the first state in the pattern is chosen randomly. If the pattern contains two similar consecutive states, their temporal relation is always set to *before*. To generate a client sequence, the generator first determines the size of the client sequence. The size is picked from a Poisson distribution with mean μ equal to $|C|$. Each client sequence is assigned a series of potentially frequent patterns.

The database that contains generated client sequences is stored in an ASCII file. The file is structured into two parts. The first part contains a list of state types, each preceded by their coding number starting from 1 and on a separate line, as follows:

```
1 S1
2 S2
3 S3
```

The second part is opened by a BEGIN_DATA line and closed by an END_DATA line. Each line between these two lines represents a transaction and contains a list of four integers representing client-id, start time, end time, state code number. Several transactions with the same value of client-id form a client sequence. The following example shows the database of two client sequences containing five transactions. The first three transactions form a sequence of client-id 1, and the last two transactions form a sequence of client-id 2. The first transaction of a client-id 1 has a state S2 holds during a period of time $[3, 7)$. Similarly, the last transaction of a client-id 2 has a state S3 holds during a period of time $[11, 14)$.

```
1 S1
2 S2
3 S3
BEGIN_DATA
1 3  7  2
1 7  10 1
1 12 16 3
2 6  9  2
2 11 14 3
END_DATA
```

Appendix C

TAR-QL Query Language

This appendix describes the specification of TAR-QL query language.

```
SELECT [FUNCT]r
FROM Rulebase r
[WHERE {pruning_condition|rule_state_condition|
       rule_format_condition}
       [{AND|OR} {pruning_condition|rule_state_condition|
       rule_format_condition}] ...
]
[GROUP BY {LHS|RHS|SUPPORT|CONFIDENDE}]
[HAVING condition]
[ORDER BY {LSIZE|RSIZE|SUPPORT|CONFIDENCE} [DESC|ASC]]
```

In the above description, the FROM clause specifies the rulebase over which the queries are performed, r is the rule variable defined over the rulebase, and the SELECT clause selects the variable that satisfies the condition of the WHERE clause. The optional FUNCT specifies aggregation operations over the resulting set of rules.

Aggregate Functions

A number of aggregate functions exist: COUNT, MAX, MIN, and AVG. These functions can be used in the SELECT clause or in a HAVING clause which will be discussed later. The description of the functions is as follows.

- COUNT: returns the number of rules in the resulting set of rules,
- AVG_CONF, AVG_SUP, AVG_LSIZE, AVG_RSIZE: return the average levels of confidence, support, and the size of rules' LHS/RHS in the resulting set,
- MAX_CONF, MAX_SUP, MAX_LSIZE, MAX_RSIZE: return the maximum values of confidence, support, and the size of rules' LHS/RHS in the resulting set,
- MIN_CONF, MIN_SUP, MIN_LSIZE, MIN_RSIZE: return the minimum values of confidence, support, and the size of rules' LHS/RHS in the answer set.

Conditional Expressions

Three types of conditions after the WHERE clause are: *pruning conditions*, *rule state conditions*, and *rule format conditions*.

Pruning conditions

The pruning conditions are used to specify queries involving rule attributes, such as support, confidence, the size of rules, and the name of states. They have the following format:

```
Rule_funct(r) <rel_op> const
```

The *Rule_funct* is one of the following defined functions:

- CONF(r): returns the confidence value of the rule r,

- SUP(r): returns the support value of the rule r,
- SIZE_LHS(r): returns the number of states of the rule r left-hand side,
- SIZE_RHS(r): returns the size of the rule r right-hand side,
- STATE_LHS(r, n): returns the n-th state of the rule r left-hand side,
- STATE_RHS(r, n): returns the n-th state of the rule r right-hand side.

The *rel_op* is one of the relational operators $\leq$, $\geq$, $<$, $>$, and $=$. The *const* is a constant.

Query C.1 Find rules whose support is greater than 50% and confidence is greater than 70%.

```
SELECT r
FROM Rulebase r
WHERE SUP(r) > 50% AND CONF(r) > 70%
```

Query C.2 Find rules whose number of states on the LHS of the rules is fewer than 4 and second state on the RHS of the rules is '*B*'.

```
SELECT r
FROM Rulebase r
WHERE SIZE_LHS(r) < 4
        AND STATE_RHS(r, 2) > 'B'
```

Rule state conditions

The rule state conditions allow the user to specify that the rule's LHS/RHS (left-hand side/right-hand side) must contain a certain set of states, be a subset of a given set of states, or be equal to a set of given states. These conditions have the format:

```
states_funct <set_op> const,
```

The *states_funct* is one of the following defined functions:

- STATES_LHS(r): returns the set of states of the rule r left-hand side,
- STATES_RHS(r): returns the set of states of the rule r right-hand side.

The *set_op* is one of the set operators SUPERSET_OF, SUBSET_OF and EQUAL. The *const* is a set of states.

Query C.3 Find rules whose the RHS of the rule contains a set {A, B, D}.

```
SELECT  r
FROM Rulebase r
WHERE STATES_RHS(r) SUPERSET_OF {A, B, D}
```

Rule format conditions

The rule format condition allows the user to specify that the rule's LHS/RHS must contain a temporal pattern, be contained in a temporal pattern, or be equal to a temporal pattern. It has the format:

```
pattern_funct <pat_op> const
```

The *pattern_funct* is one of the following two defined functions:

- LHS(r): returns the left-hand side of the rule r,
- RHS(r): returns the right-hand side of the rule r.

The *pat_op* is one of the temporal pattern containment operators CONTAINS, CONTAINED_IN and EQUAL. The *const* is a temporal pattern constant.

In order to simplify the writing of temporal patterns in the query, a function TO_PATTERN is introduced to convert a pair containing a list of states and a list of relationships to a normalized temporal pattern. As an example, TO_PATTERN(⟨A, C, B, D⟩, ⟨o, b, m, b, b, o⟩) represents a pattern

$$\begin{array}{cccc} A & C & B & D \\ \left[\begin{array}{cccc} = & o & b & b \\ * & = & m & b \\ * & * & = & o \\ * & * & * & = \end{array}\right] \end{array}.$$

Query C.4 Find rules whose the RHS of the rule contains a pattern $\begin{array}{ccc} A & B & D \\ \left[\begin{array}{ccc} = & o & b \\ * & = & m \\ * & * & = \end{array}\right] \end{array}$.

```
SELECT r
FROM Rulebase r
WHERE RHS(r) CONTAINS TO_PATTERN(<A, B, D>, <o, b, m>)
```

Grouping and Ordering Rules

The GROUP BY clause can be used to group the resulting rules based on various grouping criteria. After the rules have been grouped, aggregate functions can be applied to the individual group. Four keywords that can be used with the GROUP BY clause are LHS, RHS, SUPPORT, and CONFIDENDE, for grouping rules based on the structure of LHS/RHS of the rules, support, and confidence.

Query C.5 Find average confidence of rules that have similar structure on their left-hand side.

```
SELECT AVG_CONF(r)
FROM Rulebase r
GROUP BY LHS
```

Sometimes we want to retrieve the values of aggregate functions only for groups that satisfy certain conditions. HAVING clause is used to eliminate all the groups for which the conditional expression of the HAVING clause does not evaluate to TRUE.

Query C.6 For each group of rules on which there are more than four rules, find average confidence of rules in the group.

```
SELECT AVG_CONF(r)
FROM Rulebase r
GROUP BY LHS
HAVING COUNT > 4
```

The ORDER clause specifies an order for displaying the resulting set. The ORDER BY LSIZE, ORDER BY RSIZE, ORDER BY SUPPORT, and ORDER BY CONFIDENCE are used to order rules based on the sizes of LHS and RHS of the rules, support, and confidence, respectively. Furthermore, the resulting rules may be sorted in ascending (ASC) or descending (DESC) order.

Bibliography

Agrawal, R., Imielinski, T. & Swami, A. (1993), Mining association rules between sets of items in large databases, *in* 'Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'93)', pp. 207–216.

Agrawal, R. & Srikant, R. (1994), Fast algorithms for mining association rules, *in* 'Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)', pp. 487–499.

Agrawal, R. & Srikant, R. (1995), Mining sequential patterns, *in* P. S. Yu & A. S. P. Chen, eds, 'Proceedings of the 11th International Conference on Data Engineering (ICDE'95)', IEEE Computer Society Press, Taipei, Taiwan, pp. 3–14.

Aiello, M., Monz, C., Todoran, L. & Worring, M. (2002), 'Document understanding for a broad class of documents', *International Journal on Document Analysis and Recognition* **5**(1), 1–16.

Ale, J. M. & Rossi, G. H. (2000), An approach to discovering temporal association rules, *in* 'Proceedings of the 2000 ACM Symposium on Applied Computing', pp. 294–300.

Allen, J. F. (1983), 'Maintaining knowledge about temporal intervals', *Communications of the ACM* **26**(11), 832–843.

André-Jönsson, H. & Badal, D. Z. (1997), Using signature files for querying time-series data, *in* H. J. Komorowski & J. M. Zytkow, eds, 'Proceedings of Principles of Data Mining and Knowledge Discovery, First European Symposium', Vol. 1263 of *LNCS*, Springer, Trondheim, Norway, pp. 211–220.

Antunes, C. M. & Oliveira, A. L. (2001), Temporal data mining: An overview, *in* 'Proceedings of the KDD'01 Workshop on Temporal Data Mining', San Francisco, USA, pp. 1–13.

Antunes, C. & Oliveira, A. L. (2003), Generalization of pattern-growth methods for sequential pattern mining with gap constraints, *in* P. Perner & A. Rosenfeld, eds, 'Proceedings of the 3rd International Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM'03)', Vol. 2734 of *LNCS*, Springer, Leipzig, Germany, pp. 239–251.

Antunes, C. & Oliveira, A. L. (2004), Sequential pattern mining algorithms: Trade-offs between speed and memory, *in* 'Proceedings of the 2nd International Workshop on Mining Graphs, Trees and Sequences (MGTS'04)', Pisa, Italy.

Aref, W. G., Elfeky, M. G. & Elmagarmid, A. K. (2004), 'Incremental, online, and merge mining of partial periodic patterns in time-series databases', *IEEE Transactions on Knowledge and Data Engineering* **16**(3), 332–342.

Ayres, J., Flannick, J., Gehrke, J. & Yiu, T. (2002), Sequential pattern mining using a bitmap representation, *in* 'Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02)', ACM Press, Edmonton, Alberta, Canada, pp. 429–435.

Baron, S. & Spiliopoulou, M. (2001), Monitoring change in mining results, *in* Y. Kambayashi, W. Winiwarter & M. Arikawa, eds, 'Proceedings of the 3rd International Conference on Data Warehousing and Knowledge Discovery (DaWak'01)', Vol. 2114 of *LNCS*, Springer, Munich, Germany, pp. 51–60.

Bayardo, R. J. (1998), Efficiently mining long patterns from databases, *in* 'Proceedings of the ACM SIGMOD International Conference on the Management of Data (SIGMOD'98)', ACM Press, Seattle, WA, USA, pp. 85–93.

Berberidis, C., Vlahavas, I. P., Aref, W. G., Atallah, M. J. & Elmagarmid, A. K. (2002), On the discovery of weak periodicities in large time series, *in* T. Elomaa, H. Mannila & H. Toivonen, eds, 'Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery in Databases (PKDD'02)', Vol. 2431 of *LNCS*, Springer, Helsinki, Finland, pp. 51–61.

Bodon, F. (2003), A fast apriori implementation, *in* 'Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations', Melbourne, Florida, USA.

Böhlen, M. H., Busatto, R. & Jensen, C. S. (1998), Point-versus interval-based temporal data models., *in* 'Proceedings of the 14th International Conference on Data Engineering (ICDE'98)', IEEE Computer Society Press, Orlando, Florida, USA, pp. 192–200.

Botta, M., Boulicaut, J.-F., Masson, C. & Meo, R. (2004), Query languages supporting descriptive rule mining: A comparative study, *in* R. Meo, P. L. Lanzi & M. Klemettinen, eds, 'Database Support for Data Mining Applications', Vol. 2682 of *LNCS*, Springer, pp. 24–51.

Carterette, B. & Can, F. (2005), 'Comparing inverted files and signature files for searching a large lexicon', *Information Processing and Management* **41**(3), 613–633.

Casas-Garriga, G. (2003), Discovering unbounded episodes in sequential data, *in* N. Lavrac, D. Gamberger, H. Blockeel & L. Todorovski, eds, 'Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'03)', Vol. 2838 of *LNCS*, Springer, Cavtat-Dubrovnik, Croatia, pp. 83–94.

Ceglar, A. & Roddick, J. F. (2006), 'Association mining', *ACM Computing Surveys* **38**(2).

Chandra, R., Segev, A. & Stonebraker, M. (1994), Implementing calendars and temporal rules in next generation databases, *in* 'Proceedings of the 10th International Conference on Data Engineering (ICDE'94)', Houston, TX, USA, pp. 264–273.

Chen, X. & Petrounias, I. (1998), A framework for temporal data mining, *in* 'Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA'98)', Vienna, Austria, pp. 796–805.

Chen, X. & Petrounias, I. (2000), An integrated query and mining system for temporal association rules, *in* Y. Kambayashi, M. K. Mohania & A. M. Tjoa, eds, 'Proceedings of the 2nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK'00)', London, UK, pp. 327–336.

Chen, Y. (2004), 'Building signature trees into OODBs', *Journal of Information Science and Engineering* **20**, 275–304.

Comer, D. (1979), 'The ubiquitous b-tree', *Computing Surveys* **11**(2), 121–137.

Das, G. & Gunopulos, D. (2003), 'Time series similarity and indexing', *Invited Chapter in Handbook on Data Mining* .

Das, G., Lin, K.-I., Mannila, H., Renganathan, G. & Smyth, P. (1998), Rule discovery from time series, *in* 'Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining', New York City, New York, USA, pp. 16–22.

Deppisch, U. (1986), S-tree: A dynamic balanced signature index for office retrieval, *in* 'Proceedings of the 1986 ACM Conference on Research and Development in Information Retrieval', pp. 77–87.

Dietterich, T. G. & Michalski, R. S. (1985), 'Discovering patterns in sequences of events', *Artificial Intelligent* **25**, 187–232.

Drakengren, T. & Jonsson, P. (1997), Towards a complete classification of tractability in Allen's algebra, *in* 'Proceedings of the International Joint Conference on Artificial Intelligence', pp. 1466–1475.

Elfeky, M. G., Aref, W. G. & Elmagarmid, A. K. (2004), Using convolution to mine obscure periodic patterns in one pass, *in* E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Böhm & E. Ferrari, eds, 'Proceedings of the 9th International Conference on Extending Database Technology (EDBT'04)', Vol. 2992 of *LNCS*, Springer, Heraklion, Crete, Greece, pp. 605–620.

Fagin, R., Nievergelt, J. & Strong, H. (1979), 'Extendible hashing: A fast access method for dynamic files', *ACM Transactions on Database Systems* **4**(3), 315–344.

Faloutsos, C. & Christodoulakis, S. (1987), 'Description and performance analysis of signature file methods for office filing', *ACM Transactions on Information Systems* **5**(3), 237–257.

Fayyad, U. M., Piatetsky-Shapiro, G. & Smyth, P. (1996), 'The KDD process for extracting useful knowledge from volumes of data', *Communication of the ACM* **39**(11), 27–34.

Freksa, C. (1992), 'Temporal reasoning based on semi-intervals', *Artificial Intelligence* **54**(1), 199–227.

Garofalakis, M. N., Rastogi, R. & Shim, K. (1999), SPIRIT: Sequential pattern mining with regular expression constraints, *in* M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik & M. L. Brodie, eds, 'Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)', Morgan Kaufmann, Edinburgh, Scotland, UK, pp. 223–234.

Goutte, C., Toft, P. & Rostrup, E. (1999), 'On clustering fMRI time series', *Neuroimage* **9**(3), 298–310.

Guesgen, H. (1989), Spatial reasoning based on allen's temporal logic, Technical Report TR-89-049, International Computer Science Institute, Berkeley, CA.

Guimarães, G. & Ultsch, A. (1999), A method for temporal knowledge conversion, *in* 'Proceedings of the 3rd International Conference in Intelligent Data Analysis (IDA'99)', Springer, pp. 369–380.

Guttman, A. (1984), R-Trees: A dynamic index structure for spatial searching, *in* 'Proceedings of the ACM SIGMOD International Conference on Management of Data', Boston, USA, pp. 47–57.

Hamilton, H. J. & Randall, D. J. (2000), Data mining with calendar attributes, *in* J. F. Roddick & K. Hornsby, eds, 'Proceedings of the International Workshop on Temporal, Spatial and Spatio-Temporal Data Mining (TSDM'00)', Vol. 2007 of *LNAI*, Springer, Lyon, France.

Han, J., Chiang, J. Y., Chee, S., Chen, J., Chen, Q., Cheng, S., Gong, W., Kamber, M., Koperski, K., Liu, G., Lu, Y., Stefanovic, N., Winstone, L., Xia, B. B., Zaiane, O. R., Zhang, S. & Zhu, H. (1996), DBMiner: a system for data mining in relational databases and data warehouses, *in* 'Proceedings of the International Conference on Knowledge Discovery in Databases and Data Mining (KDD'96)'.

Han, J., Dong, G. & Yin, Y. (1999), Efficient mining of partial periodic patterns in time series database, *in* 'Proceedings of the 15th International Conference on Data Engineering (ICDE'99)', IEEE Computer Society Press, Sydney, Australia, pp. 106–115.

Han, J., Gong, W. & Yin, Y. (1998), Mining segment-wise periodic patterns in time-related databases, *in* 'Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining', AAAI Press, pp. 214–218.

Han, J. & Kamber, M. (2001), *Data Mining : Concepts and Techniques*, Academic Press, San Diego.

Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U. & Hsu, M.-C. (2000), FreeSpan: Frequent pattern-projected sequential pattern mining, *in* 'Proceedings of the 2000 International Conference on Knowledge Discovery and Data Mining (KDD'00)', Boston, MA, USA, pp. 20–23.

Han, J., Pei, J. & Yin, Y. (2000), Mining frequent patterns without candidate generation, *in* W. Chen, J. Naughton & P. A. Bernstein, eds, 'Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'00)', pp. 1–12.

Han, J., Wang, J., Lu, Y. & Tzvetkov, P. (2002), Mining top-k frequent closed pattern without minimum support, *in* 'Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)', Maebashi, Japan, pp. 211–218.

Harms, S. K. & Deogun, J. (2004), 'Sequential association rule mining with time lags', *Journal of Intelligent Information Systems, Special issue on Data Mining* **22**(1), 7–22.

Harms, S. K., Deogun, J. S., Saquer, J. & Tadesse, T. (2001), Discovering representative episodal association rules from event sequences using frequent closed episode sets and event constraints, *in* N. Cercone, T. Y. Lin & X. Wu, eds, 'Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM'01)', IEEE Computer Society, San Jose, California, USA, pp. 603–606.

Harms, S. K., Deogun, J. S. & Tadesse, T. (2002), Discovering sequential association rules with constraints and time lags in multiple sequences, *in* M.-S. Hacid, Z. W. Ras, D. A. Zighed & Y. Kodratoff, eds, 'Proceedings of the 13th International Symposium on Foundations of Intelligent Systems (ISMIS'02)', Vol. 2366 of *LNCS*, Springer, Lyon, France, pp. 432–441.

Helmer, S. & Moerkotte, G. (1999), 'A performance study of four index structures for set-valued attributes of low cardinality', *Reihe Informatik 2, University of Mannheim* .

Hofmann, H., Siebes, A. P. & Wilhelm, A. F. (2000), Visualizing association rules with interactive mosaic plots, *in* 'Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00)', Boston, MA, USA, pp. 227–235.

Höppner, F. (2001), Learning temporal rules from state sequence, *in* 'Proceedings of IJCAI'01 Workshop on Learning from Temporal and Spatial Data', Seattle, USA, pp. 25–31.

Höppner, F. (2002), Time series abstraction methods - a survey, *in* 'Proceedings GI Jahrestagung Informatik, Workshop on Knowledge Discovery in Databases', Lecture Notes in Informatics, Dortmund, Germany, pp. 777–786.

Höppner, F. (2003), 'Knowledge discovery from sequential data'. PhD Dissertation, Technical University Braunschweig, Germany.

Huang, K.-Y. & Chang, C.-H. (2004*a*), Asynchronous periodic patterns mining in temporal databases, *in* 'Proceedings of the IASTED International Conference on Databases and Applications, part of the 22nd Multi-Conference on Applied Informatics', IASTED/ACTA Press, Innsbruck, Austria, pp. 43–48.

Huang, K.-Y. & Chang, C.-H. (2004*b*), Mining periodic patterns in sequence data, *in* Y. Kambayashi, M. K. Mohania & W. Wöß, eds, 'Proceedings of the 6th International Conference on Data Warehousing and Knowledge Discovery (DaWaK'04)', Vol. 3181 of *LNCS*, Springer, Zaragoza, Spain, pp. 401–410.

Imielinski, T. & Mannila, H. (1996), 'A database perspective on knowledge discovery', *Communication of the ACM* **39**(11), 58–64.

Imielinski, T. & Virmani, A. (1998), Association rules... and what's next? towards second generation data mining systems, *in* W. Litwin, T. Morzy & G. Vossen, eds, 'Proceedings of the 2nd East European Symposium on Advances in Databases and Information Systems (ADBIS'98)', LNCS, Springer, Poznan, Poland, pp. 6–25.

Imielinski, T. & Virmani, A. (1999), 'MSQL: A query language for database mining', *Data Mining and Knowledge Discovery* **3**(4), 373–408.

Ishikawa, Y., Kitagawa, H. & Ohbo, N. (1993), Evaluation of signature files as set access facilities in OODBs, *in* P. Buneman & S. Jajodia, eds, 'Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'93)', ACM Press, pp. 247–256.

Joshi, M. V., Karypis, G. & Kumar, V. (2001), Universal formulation of sequential patterns, *in* 'Proceedings of the KDD'01 Workshop on Temporal Data Mining', San Francisco, USA.

Kam, P.-S. & Fu, A. W.-C. (2000), Discovering temporal patterns for interval-based events, *in* Y. Kambayashi, M. K. Mohania & A. M. Tjoa, eds, 'Proceedings of the 2nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK'00)', Vol. 1874 of *LNCS*, Springer, London, UK, pp. 317–326.

Kao, B., Zhang, M., Yip, C. L., Cheung, D. W. & Fayyad, U. M. (2005), 'Efficient algorithms for mining and incremental update of maximal frequent sequences', *Data Mining and Knowledge Discovery* **10**(2), 87–116.

Karimi, K. & Hamilton, H. J. (2000), Finding temporal relations: Causal bayesian network vs. C4.5, *in* 'Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems', Charlotte, NC, USA, pp. 266–273.

Keogh, E., Lin, J. & Truppel, W. (2003), Clustering of time series subsequences is meaningless: Implications for past and future research, *in* 'Proceedings of the 2003 IEEE International Conference on Data Mining (ICDM'03)', Melbourne, FL, pp. 115–122.

Kitagawa, H. & Fukushima, K. (1996), Composite bit-sliced signature file: An efficient access method for set-valued object retrieval, *in* 'Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications (CODAS'96)', Kyoto, Japan, pp. 542–549.

Kitagawa, H., Fukushima, Y., Ishikawa, Y. & Ohbo, N. (1993), Estimation of false drops in set-valued object retrieval with signature files, *in* D. B. Lomet, ed., 'Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO'93)', Vol. 730 of *LNCS*, Springer, pp. 146–163.

Kitagawa, H., Watanabe, N. & Ishikawa, Y. (1996), Design and evaluation of signature file organization incorporating vertical and horizontal decomposition schemes, *in* R. Wagner & H. Thoma, eds, 'Proceedings of the 7th International Conference on Database and Expert Systems Applications (DEXA'96)', Vol. 1134 of *LNCS*, Springer, Zurich, Switzerland, pp. 875–888.

Klemettinen, M., Mannila, H., Ronkainen, P., Toivonen, H. & Verkamo, A. (1994), Finding interesting rules from large sets of discovered association rules, *in* N. Adam, B. Bhargava & Y. Yesha, eds, 'Proceedings of the 3rd International Conference on Information and Knowledge Management', ACM Press, Gaithersburg, Maryland, pp. 401–407.

Klemettinen, M., Mannila, H. & Toivonen, H. (1996), Interactive exploration of discovered knowledge: A methodology for interaction, and usability studies, Technical Report C-1996-3, Department of Computer Science, University of Helsinki, Finland.

Koskela, T., Lehtokangas, M., Saarinem, J. & Kaski, K. (1996), Time series prediction with multilayer perceptron, FIR, and Elman neural networks, *in* 'Proceedings of World Congress on Neural Network', pp. 491–496.

Koundourakis, G. & Theodoulidis, B. (2002), Association rules and evolution in time, *in* 'Proceedings of Methods and Applications of Artificial Intelligence, Second Hellenic Conference on AI, SETN 2002', Thessaloniki, Greece, pp. 261–272.

Kouris, I. N., Makris, C. H. & Tsakalidis, A. K. (2004), 'Using information retrieval techniques for supporting data mining', *Data and Knowledge Engineering* **52**(2005), 353–383.

Krokhin, A. A., Jeavons, P. & Jonsson, P. (2003), 'Reasoning about temporal relations: The tractable subalgebras of Allen's interval algebra', *Journal of the ACM* **50**(5), 591–640.

Kundu, A., He, Y. & Bahl, P. (1988), Word recognition and word hypothesis generation for handwritten script: A Hidden Markov Model based approach, *in* 'Proceedings of 1988 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'88)', pp. 457–462.

Laxman, S., S., S. P. & Unnikrishnan, K. P. (2004), Fast algorithms for frequent episode discovery in event sequences, *in* 'Proceedings of the 3rd Workshop on Mining Temporal and Sequential Data', Seattle, WA.

Laxman, S. & Sastry, P. S. (2006), 'A survey of temporal data mining', *Sãdhanã* **31**, 173–198.

Leban, B., McDonald, D. & Forster, D. (1986), A representation for collections of temporal intervals, *in* 'Proceedings of the AAAI-1986 5th International Conference on Artificial Intelligence', pp. 367–371.

Lee, C.-H., Lin, C.-R. & Chen, M.-S. (2001), On mining general temporal association rules in a publication database, *in* 'Proceedings of the 1st IEEE International Conference on Data Mining'.

Lee, D. L. & Leng, C.-W. (1989), 'Partitioned signature files: Design issues and performance evaluation', *ACM Transactions on Information Systems* **7**(2), 158–180.

Lee, D. L. & Leng, C.-W. (1990), A partitioned signature file structure for multiattribute and text retrieval, *in* 'Proceedings of the 6th International Conference on Data Engineering (ICDE'90)', Los Angeles, California, pp. 389–397.

Lee, J. W., Lee, Y. J., Kim, H. K., Hwang, B. H. & Ryu, K. H. (2002), Discovering temporal relation rules from interval data, *in* 'Proceedings of the 1st EuroAsian Conference on Advance in Information and Communication Technology', Shiraz, Iran.

Lee, W.-C. & Lee, D. L. (1992), Signature file methods for indexing object-oriented database systems, *in* 'Proceedings of the 2nd International Computer Science Conference', Hongkong, pp. 616–622.

Lee, W.-C. & Lee, D. L. (1999), 'Signature caching techniques for information filtering in mobile environments', *Wireless Networks* **5**(1), 57–67.

Leleu, M., Rigotti, C., Boulicaut, J.-F. & Euvrard, G. (2003*a*), Constraint-based mining of sequential patterns over datasets with consecutive repetitions, *in* N. Lavrac, D. Gamberger, H. Blockeel & L. Todorovski, eds, 'Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'03)', Vol. 2838 of *LNCS*, Springer, Cavtat-Dubrovnik, Croatia, pp. 303–314.

Leleu, M., Rigotti, C., Boulicaut, J.-F. & Euvrard, G. (2003*b*), Go-spade: Mining sequential patterns over datasets with consecutive repetitions., *in* P. Perner & A. Rosenfeld, eds, 'Proceedings of the 3rd International Conference on

Machine Learning and Data Mining in Pattern Recognition (MLDM'03)', Vol. 2734 of *LNCS*, Springer, Leipzig, Germany, pp. 293–306.

Lent, B., Swami, A. N. & Widom, J. (1997), Clustering association rules., *in* W. A. Gray & P.-Å. Larson, eds, 'Proceedings of the 13th International Conference on Data Engineering (ICDE'97)', IEEE Computer Society, pp. 220–231.

Li, Y., Ning, P., Wang, X. S. & Jajodia, S. (2001), Discovering calendar-based temporal association rules, *in* 'Proceedings of the 8th International Symposium on Temporal Representation and Reasoning', pp. 111–118.

Liao, T. W. (2005), 'Clustering of time series data - a survey', *Pattern Recognition* **38**(11), 1857–1874.

Lin, M.-Y. & Lee, S.-Y. (2002), Fast discovery of sequential patterns by memory indexing, *in* 'Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery (DaWaK'02)', Aix-en-Provence, France, pp. 150–160.

Lin, M.-Y. & Lee, S.-Y. (2003), Improving the efficiency of interactive sequential pattern mining by incremental pattern discovery, *in* 'Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)', Hawaii, USA.

Lin, M.-Y., Lee, S.-Y. & Wang, S.-S. (2002), DELISP: Efficient discovery of generalized sequential patterns by delimited pattern-growth technology, *in* 'Proceedings of the 6th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'02)', Taipei, Taiwan, pp. 189–209.

Lin, W., Orgun, M. A. & Williams, G. J. (2002), An overview of temporal data mining, *in* S. J. Simoff, G. J. Williams & M. Hegland, eds, 'Proceedings of the 1st Australasian Data Mining Workshop (ADM'02)', Sydney, Australia, pp. 83–89.

Lin, Z. & Faloutsos, C. (1992), 'Frame-sliced signature files', *IEEE Transactions on Knowledge and Data Engineering* **4**(3), 281–289.

Litwin, W. (1980), Linear Hashing: a new tool for files and table addressing, *in* 'Proceedings of the 6th Conference on Very Large Databases', Montreal, Canada, pp. 212–223.

Liu, B., Hsu, W. & Ma, Y. (1999), Pruning and summarizing the discovered associations, *in* 'Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'99)', San Diego, CA, USA, pp. 125–134.

Liu, B., Hu, M. & Hsu, W. (2000), Multi-level organization and summarization of the discovered rules, *in* 'Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00)', ACM Press, New York, NY, USA.

Liu, B., Ma, Y. & Lee, R. (2001), Analyzing the interestingness of association rules from the temporal dimension, *in* 'Proceedings of IEEE International Conference on Data Mining (ICDM'01)', Silicon Valley, USA, pp. 377–384.

Lu, H., Feng, L. & Han, J. (2000), 'Beyond intratransaction association analysis: mining multidimensional intertransaction association rules', *ACM Transactions on Information Systems* **18**(4), 423 – 454.

Lu, H., Han, J. & Feng, L. (1998), Stock movement prediction and n-dimensional inter-transaction association rules, *in* 'Proceedings of the SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'98)', Seattle, Washington, pp. 12:1–12:7.

Mannila, H. & Toivonen, H. (1996), Discovering generalised episodes using minimal occurrences, *in* E. Simoudis, J. Han & U. Fayyad, eds, 'Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)', AAAI Press, Portland, Oregon, pp. 146–151.

Mannila, H., Toivonen, H. & Verkamo, A. (1997), 'Discovery of frequent episodes in event sequences', *Data Mining and Knowledge Discovery* **1**(3), 259 – 289.

Mannila, H., Toivonen, H. & Verkamo, A. I. (1995), Discovering frequent episodes in sequences, *in* U. M. Fayyad & R. Uthurusamy, eds, 'Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining (KDD'95)', AAAI Press, Montreal, Canada, pp. 210–215.

Masseglia, F., Cathala, F. & Poncelet, P. (1998), The PSP approach for mining sequential patterns, *in* 'Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery in Databases (PKDD'98)', Nantes, France, pp. 176–184.

Masseglia, F., Poncelet, P. & Teisseire, M. (2003), 'Incremental mining of sequential patterns in large databases', *Data and Knowledge Engineering* **46**(1), 97–121.

Méger, N. & Rigotti, C. (2004), Constraint-based mining of episode rules and optimal window sizes, *in* J.-F. Boulicaut, F. Esposito, F. Giannotti & D. Pedreschi, eds, 'Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'04)', Vol. 3202 of *LNCS*, Springer, Pisa, Italy, pp. 313–324.

Meo, R., Psaila, G. & Ceri, S. (1996), A new SQL-like operator for mining association rules, *in* T. M. Vijayaraman, A. P. Buchmann, C. Mohan & N. L. Sarda, eds, 'Proceedings of the 22th International Conference on Very Large Data Bases (VLDB'96)', pp. 122–133.

Moffat, A. & Zobel, J. (1996), 'Self-indexing inverted files for fast text retrieval', *ACM Transactions on Information Systems* **14**(4), 349–379.

Mooney, C. H. & Roddick, J. F. (2004), Mining relationships between interacting episodes, *in* M. W. Berry, U. Dayal, C. Kamath & D. Skillicorn, eds, 'Proceedings of the 4th SIAM International Conference on Data Mining (SDM'04)', Lake Buena Vista, Florida, USA.

Mörchen, F. (2006), A better tool than Allen's relations for expressing temporal knowledge in interval data, *in* 'Proceedings of the Workshop on Theory and Practice of Temporal Data Mining (TPTDM'06)', Philadelphia, PA, USA, pp. 25–34.

Mörchen, F. & Ultsch, A. (2004), Mining hierarchical temporal patterns in multivariate time series, *in* S. Biundo, T. W. Frühwirth & G. Palm, eds, 'Proceedings of the 27th Annual German Conference in Artificial Intelligence', Springer, pp. 127–140.

Mörchen, F., Ultsch, A. & Hoos, O. (2004), Discovering interpretable muscle activation patterns with the temporal data mining method, *in* J.-F. Boulicaut, F. Esposito, F. Giannotti & D. Pedreschi, eds, 'Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'04)', Vol. 3202 of *LNCS*, Springer, Pisa, Italy, pp. 512–514.

Morzy, T., Wojciechowski, M. & Zakrzewicz, M. (2001), Optimizing pattern queries for web access logs, *in* 'Proceedings of the 5th East European Conference on Advances in Databases and Information Systems (ADBIS'01)', Vol. 2151 of *LNCS*, Springer, pp. 141–154.

Morzy, T. & Zakrzewicz, M. (1998), Group bitmap index: A structure for association rules retrieval, *in* 'Proceedings of the 4th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'98)', pp. 284–288.

Nalwa, V. S. (1997), 'Automatic on-line signature verification', *Proceedings of the IEEE* **85**, 215–239.

Nanopoulos, A., Zakrzewicz, M., Morzy, T. & Manolopoulos, Y. (2003), 'Efficient storage and querying of sequential patterns in database systems', *Information and Software Technology* **45**, 23–34.

Nebel, B. & Bürckert, H.-J. (1995), 'Reasoning about temporal relations: a maximal tractable subclass of allen's interval algebra', *Journal of the ACM* **42**(1), 43–66.

Netz, A., Chaudhuri, S., Fayyad, U. M. & Bernhardt, J. (2001), Integrating data mining with SQL databases: OLE DB for data mining, *in* 'Proceedings of the 17th International Conference on Data Engineering (ICDE'01)', pp. 379–387.

Nørvåg, K. (1999), Efficient use of signatures in object-oriented database systems, *in* 'Proceedings of the 3rd East European Conference on Advances in Databases and Information Systems (ADBIS'99)', Vol. 1691 of *LNCS*, Springer, Maribor, Slovenia, pp. 367–381.

Ong, K. H., Ong, K. L. & Lim, E. P. (2002), Crystalclear: Active visualization of association rules, *in* 'Proceedings of the International Workshop on Active Mining (AM'02), in conjunctio with IEEE International Conference on Data Mining', Maebashi City, Japan.

Ozden, B., Ramaswamy, S. & Silberschatz, A. (1998), Cyclic association rules, *in* 'Proceedings of the 14th International Conference on Data Engineering (ICDE'98)', IEEE Computer Society Press, Orlando, Florida, USA, pp. 412–421.

Padmanabhan, B. & Tuzhilin, A. (1996), Pattern discovery in temporal databases: A temporal logic approach, *in* E. Simoudis, J. Han & U. Fayyad, eds, 'Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)', AAAI Press, Portland, Oregon, pp. 351–354.

Padmanabhan, B. & Tuzhilin, A. (1998), A belief-driven method for discovering unexpected patterns, *in* 'Proceedings of the 4st International Conference on Knowledge Discovery and Data Mining (KDD'98)', pp. 94–100.

Papapetrou, P. (2006), 'Constraint-based mining of frequent arrangements of temporal intervals'. Master Thesis, Department of Computer Science, Boston University, USA.

Papapetrou, P., Kollios, G., Sclaroff, S. & Gunopulos, D. (2005), Discovering frequent arrangements of temporal intervals, *in* 'Proceedings of the 5th IEEE International Conference on Data Mining (ICDM'05)', IEEE Computer Society Press, Houston, Texas, USA, pp. 354–361.

Parthasarathy, S., Zaki, M. J., Ogihara, M. & Dwarkadas, S. (1999), Incremental and interactive sequence mining, *in* 'Proceedings of the International Conference on Information and Knowledge Management (CIKM'99)', ACM, Kansas City, Missouri, USA, pp. 251–258.

Pasquier, N., Bastide, Y., Taouil, R. & Lakhal, L. (1999), Discovering frequent closed itemset for association rules, *in* 'Proceedings of the 7th International Conference on Database Theory (ICDT'99)', Jerusalem, Israel, pp. 398–416.

Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U. & Hsu, M.-C. (2001), PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth, *in* 'Proceedings of the 17th International Conference on Data Engineering (ICDE'01)', IEEE Computer Society Press, Heidelberg, Germany, pp. 215–224.

Rabitti, F. & Zezula, P. (1990), A dynamic signature technique for multimedia databases, *in* J.-L. Vidick, ed., 'Proceedings of the 13th International Conference on Research and Development in Information Retrieval (SIGIR'90)', ACM, Brussels, Belgium, pp. 193–210.

Rainsford, C. P. & Roddick, J. F. (1999), Adding temporal semantics to association rules, *in* J. M. Zytkow & J. Rauch, eds, 'Proceedings of the 3rd European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'99)', pp. 504–509.

Rainsford, C. P. & Roddick, J. F. (2000), Visualisation of temporal interval association rules, *in* 'Proceedings of 2nd International Conference on Intelligent Data Engineering and Automated Learning (IDEAL'00)', Vol. 1983 of *LNCS*, Springer, Shatin, N.T., Hong Kong, pp. 91–96.

Ramaswamy, S., Mahajan, S. & Silberschatz, A. (1998), On the discovery of interesting patterns in association rules, *in* 'Proceedings of the 24th International Conference on Very Large Data Bases', pp. 368–379.

Ramsay, J. O. & Silverman, B. W. (1997), *Functional Data Analysis*, Springer Series in Statistics. Springer.

Robert, C. (1979), Partial match retrieval via the method of the superimposed codes, *in* 'Proceedings of the IEEE', Vol. 67, No. 12, pp. 1624–1642.

Roddick, J. F. & Mooney, C. H. (2005), 'Linear temporal sequences and their interpretation using midpoint relationships', *IEEE Transactions on Knowledge and Data Engineering* **17**(1), 133–135.

Roddick, J. F. & Spiliopoulou, M. (2002), 'A survey of temporal knowledge discovery paradigms and methods', *IEEE Transactions on Knowledge and Data Engineering* **14**(4), 750–767.

Shneiderman, B. (1996), The eyes have it: A task by data type taxonomy for information visualization, *in* 'Proceedings of the 1996 IEEE Symposium on Visual Languages', IEEE Press, p. 336.

Silberschatz, A. & Tuzhilin, A. (1995), On subjective measure of interestingness in knowledge discovery, *in* U. M. Fayyad & R. Uthurusamy, eds, 'Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining (KDD'95)', Montréal, Québec, Canada, pp. 275–281.

Srikant, R. & Agrawal, R. (1996), Mining sequential patterns: Generalizations and performance improvements, *in* 'Proceedings of the 5th International Conference on Extending Database Technology', Avignon, France, pp. 3–17.

Toivonen, H., Klemettinen, M., Ronkainen, P., Hatonen, K. & Mannila, H. (1995), Pruning and grouping of discovered association rules, *in* 'ECML-95 Workshop on Statistics, Machine Learning, and Knowledge Discovery in Databases', Heraklion, Greece, pp. 47–52.

Tousidou, E., Bozanis, P. & Manolopoulos, Y. (2002), 'Signature-based structures for objects with set-valued attributes', *Information Systems* **27**(2), 93–121.

Tousidou, E., Nanopoulos, A. & Manolopoulos, Y. (2000), 'Improved methods for signature-tree construction', *The Computer Journal* **43**(4), 301–314.

Tronícek, Z. (2001), Episode matching, *in* A. Amir & G. M. Landau, eds, 'Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)', Vol. 2089 of *LNCS*, Springer, Jerusalem, Israel, pp. 143–146.

Tung, A. K., Lu, H., Han, J. & Feng, L. (1999), Breaking the barrier of transactions: Mining inter-transaction association rules, *in* 'Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'99)', San Diego, CA, USA, pp. 297–301.

Tuzhilin, A. & Adomavicius, G. (2002), Handling very large numbers of association rules in the analysis of microarray data, *in* 'Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02)', ACM Press, New York, NY, USA, pp. 396–404.

Tuzhilin, A. & Liu, B. (2002), Querying multiple sets of discovered rules, *in* 'Proceedings of the 8th ACM SIGKDD International Conference on Knowledge

Discovery and Data Mining (KDD'02)', ACM Press, New York, NY, USA, pp. 52–60.

Tzvetkov, P., Yan, X. & Han, J. (2003), TSP: Mining top-k closed sequential patterns, *in* 'Proceedings of the 2003 IEEE International Conference on Data Mining (ICDM'03)', Melbourne, FL.

Ultsch, A. (2004), Unification-based temporal grammar, Technical Report 37, Computer Science Department, Philipps-University Marburg, Germany.

Vilain, M. B. (1982), A system for reasoning about time, *in* 'Proceedings of the 2nd National Conference on Artificial Intelligence (AAAI'82)', AAAI Press / MIT Press, pp. 197–201.

Villafane, R., Hua, K. A., Tran, D. & Maulik, B. (2000), 'Knowledge discovery from series of interval events', *Journal of Intelligent Information Systems* **15**(1), 71–89.

Wang, J. & Han, J. (2004), BIDE: Efficient mining of frequent closed sequences, *in* 'Proceedings of the 20th International Conference on Data Engineering (ICDE'04)', IEEE Computer Society, Boston, MA, USA, pp. 79–90.

Winarko, E. & Roddick, J. F. (2003), Relative temporal association rule mining, *in* S. J. Simoff, G. J. Williams & M. Hegland, eds, 'Proceedings of the 2nd Australasian Data Mining Workshop (ADM'03)', pp. 121–142.

Winarko, E. & Roddick, J. F. (2005), Discovering richer temporal association rules from interval-based data, *in* A. M. Tjoa & J. Trujillo, eds, 'Proceedings of the 7th International Conference on Data Warehousing and Knowledge Discovery (DaWaK'05)', Vol. 3589 of *LNCS*, Springer, Copenhagen, Denmark, pp. 315–325.

Winarko, E. & Roddick, J. F. (2007), 'ARMADA - an algorithm for discovering richer relative temporal association rules from interval-based data', *Data and Knowledge Engineering* **63**, 76–90.

Wong, P. C., Whitney, P. & Thomas, J. (1999), Visualizing association rules for text mining, *in* 'Proceedings of IEEE Symposium on Information Visualization', IEEE Computer Society Press, Los Alamitos, CA, pp. 120–123.

Yamato, J., Ohya, J. & Ishii, K. (1992), Recognizing human action in time-sequential images using Hidden Markov Model, *in* 'Proceedings of 1992 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'92)', Champaign, IL, pp. 379–385.

Yan, X., Han, J. & Afshar, R. (2003), CloSpan: mining closed sequential patterns in large datasets, *in* 'Proceedings of 2003 SIAM International Conference on Data Mining (SDM'03)', San Fransisco, CA, pp. 167–177.

Yang, J., Wang, W. & Yu, P. S. (2000), Mining asynchronous periodic patterns in time series data, *in* 'Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00)', ACM Press, pp. 275–279.

Yang, J., Wang, W. & Yu, P. S. (2001), Infominer: Mining surprising periodic patterns, *in* 'Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'01)', ACM Press, pp. 395–400.

Yang, J., Wang, W. & Yu, P. S. (2002), Infominer+: Mining partial periodic patterns with gap penalties, *in* 'Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)', IEEE Computer Society Press, Maebashi City, Japan, pp. 725–728.

Yang, J., Wang, W. & Yu, P. S. (2003), STAMP: On discovery of statistically important pattern repeats in long sequential data, *in* D. Barbará & C. Kamath, eds, 'Proceedings of the 3rd SIAM International Conference on Data Mining (SDM'03)', SIAM, San Francisco, CA, USA.

Yang, J., Wang, W. & Yu, P. S. (2004), 'Discovering high-order periodic patterns', *Knowledge and Information Systems* **6**(3), 243–268.

Yang, Z. & Kitsuregawa, M. (2005), LAPIN-SPAM: An improved algorithm for mining sequential pattern, *in* 'ICDE Workshops', p. 1222.

Zaki, M. J. (1998), Efficient enumeration of frequent sequences, *in* 'Proceedings of the 7th International Conference on Information and Knowledge Management (CIKM'98)', Washington DC, USA, pp. 68–75.

Zaki, M. J. (2000), Sequence mining in categorical domains: Incorporating constraints, *in* A. Agah, J. Callan & E. Rundensteiner, eds, 'Proceedings of the 9th International Conference on Information and Knowledge Management (CIKM'00)', ACM Press, McLean, VA, USA, pp. 422–429.

Zaki, M. J. (2001), 'SPADE: An efficient algorithm for mining frequent sequences', *Machine Learning Journal* **42**(1/2), 31–60.

Zakrzewicz, M. (2001), Sequential index structure for content-based retrieval, *in* D. W.-L. Cheung, G. J. Williams & Q. Li, eds, 'Proceedings of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'01)', Vol. 2035 of *LNCS*, Springer, Hongkong, China.

Zezula, P., Rabitti, F. & Tiberio, P. (1991), 'Dynamic partitioning of signature files', *ACM Transactions on Information Systems* **9**(4), 336–367.

Zezula, P. & Tiberio, P. (1990), Selecting signature files for specific applications, Technical Report 74, CIOC-NCR, Bologna.

Zhang, M., Kao, B., Yip, C. & Cheung, D. (2001), A GSP-based efficient algorithm for mining frequent sequences, *in* 'Proceedings of the International Conference on Artificial Intelligence (ACAI'01)', Las Vegas, Nevada, USA, pp. 497–503.

Zhao, K. & Liu, B. (2001), Visual analysis of the behavior of discovered rules, *in* ‘Workshop Notes in ACM SIGMOD 2001 Workshop on Visual Data Mining’, San Fransisco, CA, USA.

Zimbrão, G., de Souza, J. M., de Almeida, V. T. & da Silva, W. A. (2002), An algorithm to discover calendar-based temporal association rules with item’s lifespan restriction, *in* ‘Proceedings of the 2nd Workshop on Temporal Data Mining’.

Zobel, J., Moffat, A. & Ramamohanarao, K. (1998), ‘Inverted files versus signature files for text indexing’, *ACM Transactions on Database Systems* **23**(4), 453–490.