EFFICIENT MAIN MEMORY ALGORITHMS FOR SIGNIFICANT INTERVAL AND FREQUENT EPISODE DISCOVERY

by

SAGAR HASMUKH SAVLA

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2006

To my parents and friends who have always encouraged me to strive for the best.

ACKNOWLEDGEMENTS

I would like to thank Dr. Sharma Chakravarthy for giving me an opportunity to work on this exciting project. Dr. Sharma's guidance, advice, support and encouragement have been invaluable in the completion of this work. Sincere thanks are also due to Dr. Mohan Kumar and Dr. Leonidas Fegaras for serving on my committee.

I would like to thank all my colleagues from ITLAB (Aditya Telang, Arvind Venkatachalam, Balakumar Kendai, Hari Hara Subramanian, Jayakrishna, Dr. Raman Adaikkalavan, Ruchika Marda, Roochi Mishra and Swapnil Sharma) for their invaluable help and advice during the course of my thesis. I would also like to acknowledge Ambika Srinivasan, Sunit Shrestha and Dhawal Bhatia for the previous work which they have done on this topic and would like to mention that some parts in this thesis have been adapted from their thesis. Last but not the least, I would like to thank my parents Mr. Hasmukh Savla and Mrs. Bharti Savla, my sister Shveta and my friends Ankur, Prachi, Bhushan, Sarika, Bhavik and others for their constant love and support. Without their encouragement and endurance, this work would not have been possible. A special thanks to Sarika Kurmi for helping proof read my thesis and improve the writing quality.

This work was also supported, in part, by the National Science Foundation (NSF) (grants ITR 0121297, IIS-0326505, EIA-0216500 and IIS 0534611) and the Computer Science and Engineering Department at The University of Texas at Arlington.

November 1, 2006

ABSTRACT

EFFICIENT MAIN MEMORY ALGORITHMS FOR SIGNIFICANT INTERVAL AND FREQUENT EPISODE DISCOVERY

Publication No. _____

SAGAR HASMUKH SAVLA, M.S.

The University of Texas at Arlington, 2006

Supervising Professor: Sharma Chakravarthy

There is a considerable research on sequential mining of time-series data. Sensor-based applications such as MavHome require prediction of events for automating the environment using time-series data collected over a period of time. In these applications, it is important to predict tight and accurate intervals of interest to effectively automate the application. Also, detection of frequent patterns is needed for the automation of sequence of happenings. Although, there is a considerable body of work on sequential mining of transactional data, most of them deal with time point data and make several iterations over the entire data set for discovering frequently occurring patterns.

An alternative approach consisting of three phases has been proposed for detecting significant intervals and frequent episodes. In the first phase, time-series data is folded over a periodicity (day, week, etc.) using which intervals are formed. The advantage of this approach is that the data set is compressed substantially thereby reducing the size of input used and hence the computation. Also, each event/device can be processed individually allowing for parallel computation of individual events. Significant intervals

that satisfy the criteria of minimum confidence and maximum interval-length specified by the user are discovered from this compressed interval data.

In this thesis, we present a new single pass main memory algorithm (OnePass-SI algorithm) for detecting significant intervals. Unlike its counterparts, OnePass-SI algorithm does not follow the classic apriori style to discover significant intervals. While analyzing the OnePass-SI algorithm, we shall discuss its characteristics, complexity, scalability issues and its advantages over other algorithms. We shall also compare the performance of our algorithm with previously developed SID and SQL-based algorithms.

For the second phase, we propose a frequent episode discovery (FED) algorithm (OnePass-FED algorithm). The OnePass-FED algorithm proposed in this thesis is a main memory algorithm. The OnePass-FED algorithm works on the significant intervals discovered in the first phase to discover interesting episodes in a single pass as compared to the apriori class of algorithms. This approach is significantly more efficient and scales well as compared to traditional mining algorithms. Extensive experimental analysis establishes its efficiency and scalability. We also compare the performance of this algorithm with our previously developed SQL-based Hybrid Apriori algorithm.

The third and final phase is frequent episode validation phase. This phase validates the frequent episodes generated by the second phase if they are to be interpreted on a finer granularity (example, week using daily periodicity). For the third phase, we present a BatchValidation algorithm. The steps carried out for frequent episode validation are redesigned in the BatchValidation algorithm to give a much improved performance. Experiments carried out to compare the performance of the BatchValidation algorithm with previously developed Naive algorithm showed that, the BatchValidation algorithm is significantly faster.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Why data mining?

With the advent of automated data collection tools, our ability to generate and collect data has increased rapidly in the last few decades. The last few decades has seen a dramatic increase in the amount of information or data being stored in electronic format. This accumulation of data has taken place at an explosive rate. The amount of information in the world is always increasing and the size and number of databases are increasing even faster. The increase in use of electronic data gathering devices such as point-of-sale or remote sensing devices has also contributed to this explosion of available data. Data storage has become easier because of the availability of large amount of low cost computing power.

Having concentrated so much attention on the accumulation of data the problem is what to do with this valuable resource? Database management systems give access to stored data. But, this is only a small part of what can be gained from the data. Traditional on-line transaction processing systems, or OLTP's, are good at putting data into databases quickly, safely and efficiently but are not good at delivering meaningful analysis in return. Analyzing data can provide further knowledge about an application. Hence, this explosive growth in data has opened the possibility of extracting useful information and knowledge from the data. 'On-Line Analytical Processing (OLAP) provide analysis techniques with functionalities such as summarization, consolidation, and aggregation, as well as ability to view information from different angles. Although OLAP tools support multidimensional analysis and decision making, additional data analysis tools are

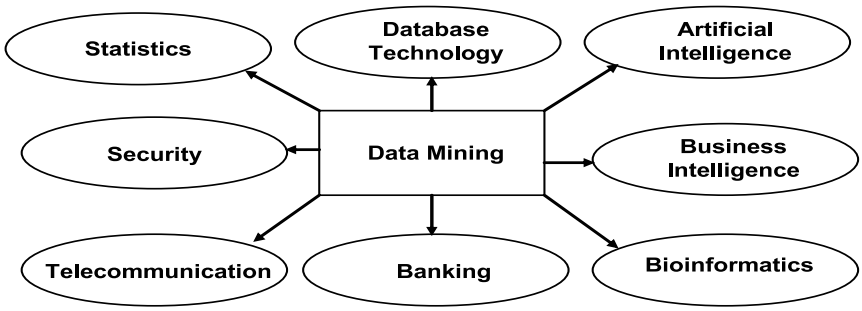

Figure 1.1 Areas where data mining is applied

required for in-depth analysis' [1]. This is where data mining has obvious benefits for any domain.

1.2 What is data mining?

Some of the most popular definitions of data mining are:

'The nontrivial extraction of implicit, previously unknown, and potentially useful information from data' [2].

'The science of extracting useful information from large data sets or databases' [3].

'Data Mining refers to the extracting or mining knowledge from large amounts of data' [1].

The goal of mining is to extract nuggets of knowledge from large amounts of data. Data mining involves the process of analyzing data to show patterns or relationships, by sifting through large amounts of data, and picking out pieces of correlated information or patterns that occur, such as picking out statistically useful and significant information from raw data. Data mining has now grown to an interdisciplinary field. Figure 1.1 lists some of the fields where data mining is been widely used nowadays.

A simple example of data mining is its use in a retail sales department. If a store tracks the purchases of a customer and notices that a customer buys a lot of silk shirts,

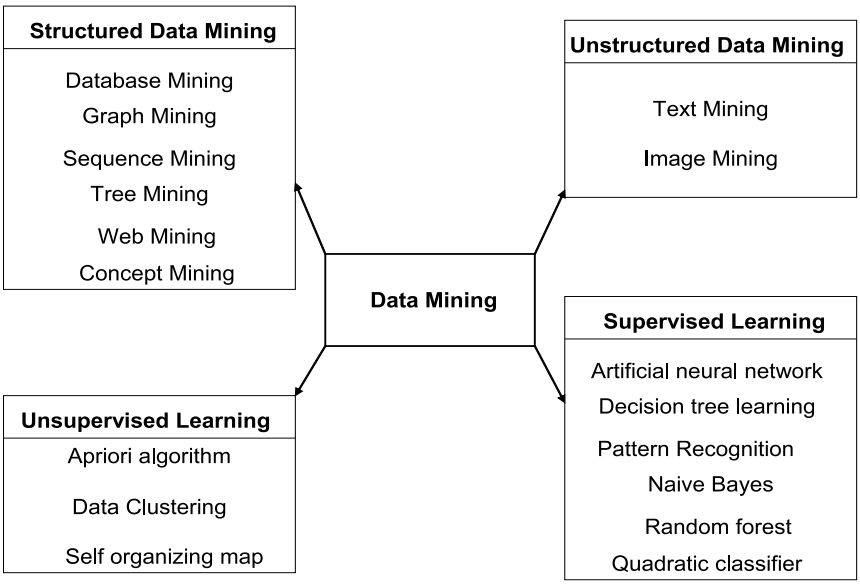

Figure 1.2 Categories in data mining

the data mining system will make a correlation between that customer and silk shirts. The sales department will look at that information and may begin direct mail marketing of silk shirts to that customer, or it may alternatively attempt to get the customer to buy a wider range of products. In this case, the data mining system used by the retail store discovered new information about the customer that was previously unknown to the company. Another widely used (though hypothetical) example is that of a very large North American chain of supermarkets. Through intensive analysis of the transactions and the goods bought over a period of time, analysts found that beer and diapers were often bought together. Though explaining this interrelation might be difficult, taking advantage of it, on the other hand, should not be hard (example, placing the high-profit diapers next to the high-profit beers). This technique is often referred to as 'Market Basket Analysis'.

The examples in the above paragraph describe a very simple use of statistical analysis using data mining techniques. Other data mining approaches include classification

and prediction using machine learning techniques [4], cluster analysis in large databases [5], time series analysis [6], sequential pattern mining [7], cyclic association rules [8], Web usage mining [9], etc. The growth in the field of data mining has caused formation of many categories and sub-categories within data mining. Some of these categories and sub-categories are seen in the Figure 1.2

1.3 Time-series data mining

Growing interest in the field of data mining over the past few decades has resulted in a number of algorithms for processing various kinds of data including time-series data. In general, time-series data can be defined as an ordered sequence of values of a variable at aperiodic time intervals. Time-series data analysis is used in a variety of data-centric applications such as economic forecasting, sales forecasting, budgetary analysis, stock market analysis, inventory studies, census analysis and so forth. Time-series data is known for its huge volume. Hence, discovering useful nuggets of information from them is a challenging task.

A considerable amount of work [10] has been done to process and mine through the large collections of time-series data sets via sequential mining. Most of the existing and traditional sequential mining techniques [11, 12] use data in its original form as time points. On the other hand, in real-world applications (example, predicting automating device in a smart home), we are interested in identifying intervals with a high degree of certainty in which an event occurs. For example, it is useful to extract intervals of high activity from telephone logs to understand the network usage. Similarly, for smart home [13] class of applications, it is useful to consider periods of high activity of the devices rather than their actual usage at various points in time, to infer the usage patterns of each device as well as interactions between different devices. Another characteristic of these traditional sequential mining algorithms is that they process the entire data set several

times. For time-series data, the sheer size of the data set makes running an algorithm that makes several passes on the entire data set time consuming. The efficiency of these sequential mining algorithms can be improved by reducing the data set without changing the outcome. The above characteristics, as well as the work in [14] makes three things very clear:

1. for many applications, it is important to interpret the occurrence of events in intervals rather than at time points,
2. the efficiency of these sequential mining algorithms can be improved by reducing the data, and
3. reducing the number of passes and intermediate results generated, will further improve the efficiency of these algorithms

1.4 Problem Domain

The algorithms proposed in this thesis can be extended to other domains which generates time-series data. The motivation for this work came from the MavHome (Managing An Intelligent and Versatile Home) project which is a multi-disciplinary research project at the University of Texas at Arlington (UTA) focused on the creation of an intelligent and versatile home environment [13]. We assume a time-series data set (generated by some application). The aim is to automate various activities in this environment such as when to turn ON a light, when to turn OFF the TV/Audio system, etc., to maximize the inhabitant comfort (or power management, security monitoring, etc.). This requires prediction of the device usage patterns using the usage data.

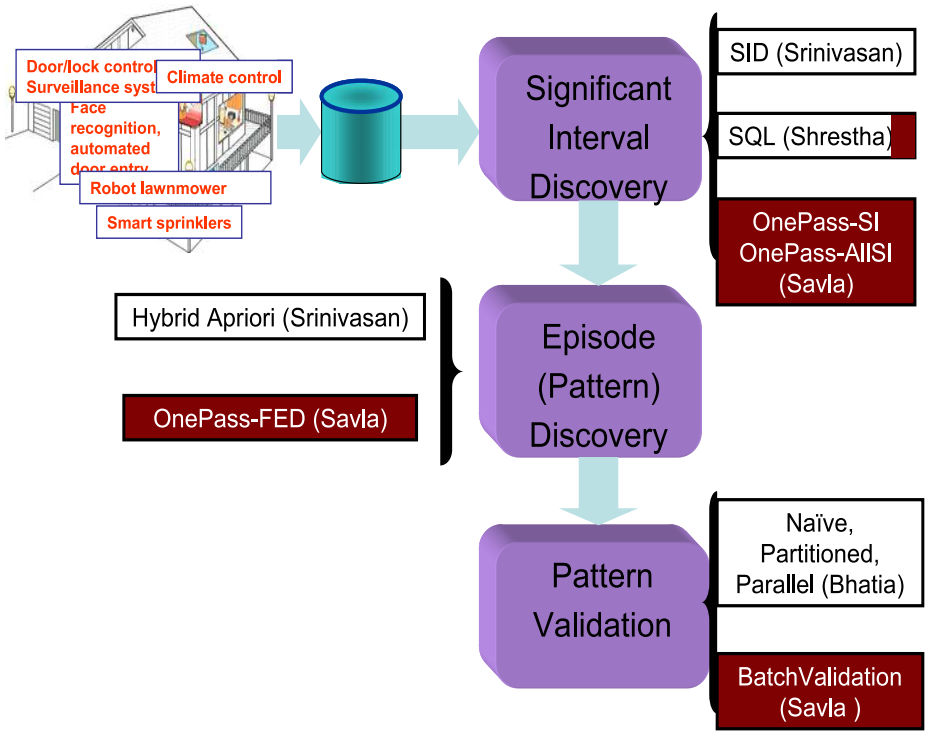

Figure 1.3 Different Phases of solution to smart home problem

1.5 Focus of the Thesis

Given the task of discovering significant intervals and frequent patterns/episodes[1], we divide our task into three phases:

1. identifying the best significant intervals (tightest intervals) based on characteristics provided by the user,
2. discovering interactions between events, to identify frequently occurring patterns/episodes of different sizes and strengths, and
3. validating the frequent episodes/patterns detected (if need be)

This thesis has contributions in each of the above three phases. Each of these phases have work done by Srinivasan [15], Shrestha [16] and Bhatia [17] for their respective thesis. The aim of this thesis is: i) to provide an efficient main memory algorithm for significant

[1]Please note that patterns and episodes are used interchangeably in the remainder of the thesis.

interval discovery, ii) to provide an efficient main memory algorithm for frequent episode discovery, iii) to redesign the frequent episode validation phase, and iv) to extend and improvise the performance of the previous works. A summary of the contributions of this thesis is shown in the Figure 1.3. The contributions of this thesis in each phase is highlighted in Figure 1.3 as brown shaded boxes. Each contribution is explained in detail in later chapters.

The organization of the rest of the thesis is as follows: Review of related work in this field is done in Chapter 2. The solution to the first phase of the problem is discussed in Chapter 3. The solution to the second phase of the problem is discussed in Chapter 4. The solution to the third and final phase of the MavHome problem is discussed in Chapter 5. Finally, the conclusions derived from this thesis and some of the potential future work are described in Chapter 6. The experimental results for each phase are seen in their respective chapters.

CHAPTER 2

RELATED WORK

Traditional algorithms to discover frequent episodes [11, 12, 18, 19, 20] operate on time stamped data. To the best of our knowledge, there is no other interval-based sequential mining algorithm. We provide a survey of approaches found in the literature in the following sections.

2.1 GSP

The GSP (Generalized Sequential Patterns) [12] is designed for transactional data where each sequence is a list of transactions ordered by transaction time and each transaction is a set of items. It extends their own previous work [7] by enabling specification of the maximum time difference between the earliest and latest event in an element as well as the minimum and maximum gaps between adjacent elements of the sequential patterns. Timing constraints such as maximum span, event-set window size, maximum gap, and minimum gap are applied in this approach. The algorithm finds all sequences that satisfy these constraints and whose support is greater than the user-specified minimum. The support counting method used is COBJ (One occurrence per object). The algorithm defines the notion of anti-monotonicity in which a sub sequence of a contiguous sequence may or may not be valid. The sequence c is a subsequence of s if any of the following holds:

1. c is derived from s by dropping an event from its first or last event-set,
2. c is derived from s by dropping an event from any of its event-sets that have at least two elements, and

3. c is a contiguous subsequence of c‘, that is a contiguous subsequence of s,

This algorithm consists of two phases: i) the first phase scans the database to identify all the frequent items of size one, and ii) the second phase is an iterative phase that scans the database to discover frequent sequences of the possible sizes. The second phase consists of the candidate generations and pruning steps wherein sequences of greater length are identified; sequences that are not frequent are pruned out from further iterations. The iterative phase is computationally intensive. Therefore, optimizations such as hash tree data structures and transformation of the data into a vertical format are proposed in this paper. The algorithm terminates when no more sequences are found.

2.2 WINEPI and MINEPI

The authors in this paper [11] concentrate on sequences of events with an associated time of occurrence that can describe the behavior and action of users or systems in several domains such as smart home environments, telecommunications systems, web usage and text mining. The algorithms are designed for discovering serial, parallel or composite sequences that represent a frequent episode. A frequent episode is defined as a collection of events that occur within the given time interval (window) in a given partial order. Based on the ordering of events in an episode, it is classified as a serial episode or a parallel episode. Unlike parallel episodes, serial episodes require a temporal order of events. Composite sequences are generated from the combination of parallel and serial sequences.

The authors propose two approaches, WINEPI and MINEPI to discover the frequent episodes in a given input sequence. In WINEPI, events of the sequences must be close to each other. The closeness is determined by the window parameter. A time window is slid over the input data and the sequences within the window are considered. Thus, the window is defined as a slice of an event sequence and an event sequence is then

considered as sequences of overlapping windows. The number of windows is determined by the width of the window. The number of windows in which an episode occurs is the support of the episode. If this support is greater than the minimum support threshold specified, the episode is detected as a frequent episode. The algorithm finds all sequences that satisfy the time constraints ms and whose support exceeds a user defined minimum support (min_sup), counted with the CWIN method - one occurrence per span window. The ms time constraint specifies the maximum allowed time difference between latest and earliest occurrences of events in the entire sequence. This algorithm makes multiple passes over the data. The first pass determines the support for all individual events. In other words, for each event the number of windows containing the event is counted. Each subsequent pass k starts with generating the k-event long candidate sequences Ck from the set of frequent sequences of length k-1 found in the previous pass. This approach is based on the subset property of the apriori principle that states that a sequence cannot be frequent unless its subsequences are also frequent. The algorithm terminates when no frequent sequences are generated at the end of the pass. For parallel episodes, WINEPI uses set of counters and sequence length for support counting; a finite state automaton is used for discovering the serial episodes.

MINEPI, an alternate approach to discovering frequent sequences is a method based on minimal occurrences of the frequent sequences. In this approach, the exact occurrences of the sequences are considered. A minimal occurrence of a sequence is determined as having an occurrence in a window w = [ts, te], but not in any of its sub windows. For each frequent sequence s, the locations of their minimal occurrences are stored, resulting in a set of minimal occurrences denoted by mo(s) = [ts, te] | [ts, te] is a minimal window in which s occurs. The support for a sequence is determined by the number of its minimal occurrences |mo(s)|. The approach defines rules of the form: s‘[w1]-> s[w2], where s‘ is a subsequence of s and w1 and w2 are windows. The interpretation of the

rule is that if s‘ has a minimal occurrence at interval [ts, te] which is shorter than w1, then s occurs within interval [ts, te‘] which is shorter than w2. The approach is similar to the universal formulation with w2 corresponding to ms and an additional constraint w1 for subsequence length, with CWINMIN as the support counting technique. The confidence and frequency of the discovered rules with a large number of window widths are obtained in a single run. MINEPI uses the same algorithm for candidate generation as WINEPI with a different support counting technique. In the first round of the main algorithm, mo(s) is computed for all sequences of length one. In the subsequent rounds, the minimal occurrences of s are located by first selecting its two suitable subsequences s1 and s2 and then performing a temporal join on their minimal occurrences. Frequent rules and patterns can be enumerated by looking at all the frequent sequences and then its subsequences. For the above algorithm, window is an extremely essential parameter since only a window's worth of sequences is discovered. Moreover, the data structures used for this algorithm can exceed the size of the database in the initial passes. But the strength of MINEPI lies in detection of episode rules without looking at the data again. The episode rule determines the connection between two sets of events as it consists of two different time bounds. This is possible since MINEPI maintains intermediate data structure for each frequent episode discovered. Making a single pass over this data structures can help in determining the sub episodes and the confidence of the episode rule. A sub graph of a frequent episode is considered as a sub episode of the frequent episode. Confidence of an episode rule is a ratio of frequency of an episode to its sub episode.

2.3 ED

The algorithm Episode Discovery (ED) proposed in [18, 21] is a data mining algorithm that discovers behavioral patterns in time-ordered input sequence. The problem

domain in this approach is a smart home where patterns related to inhabitant device interactions and the ordering information is discovered. The patterns discovered are then used by intelligent agents to automate device interactions. This approach is based on the Minimum Description Length (MDL) Principle [22] and discovers multiple characteristics of the patterns such as its frequency, periodicity, order and the length of a pattern. It uses compression ratio as the evaluation measure since greater compression ratio results in a shorter description length.

Given an input stream S the ED algorithm carries out a sequence of steps to discover the significant episodes. First, the algorithm generates maximal episodes, P_i, from the input sequence, S, by incrementally processing each event occurrence. An episode window maintains the occurrences and is pruned when an occurrence is outside of the allowable window time frame due to the addition of a new event occurrence. The window contents prior to pruning are maximal for that particular window instance, and are used to generate a maximal episode. Next, the algorithm constructs an initial collection of itemsets, one for each maximal episode. Additional itemsets are generated so that the episode subsets of the maximal episodes can be evaluated for significance. To avoid generating the power set of each maximal episode itemset, ED algorithm prunes the complete set of potential itemsets in a tractable manner, while ensuring itemsets leading to significant episodes are retained. Once the itemsets have been generated, a significant episode candidate C_n is created for each itemset (I_n). Next, each maximal episode is compared with the I_n of each candidate. If the maximal episode contains all of I_n, then the maximal episode is added to the episode set of the candidate as an occurrence of the itemset. ED evaluates the candidates by making use of the MDL principle, which targets patterns that can be used to minimize the description length of a database by replacing each instance of the pattern with a pointer to the pattern definition. Once the candidates have been evaluated, ED greedily identifies the significant episodes as those

meeting a user-configurable minimum significance (compression) value. After selecting a candidate, the algorithm marks the events that represent instances of the candidate's pattern, which allows ED to construct a filtered input stream. These steps are repeated until all candidates have been processed.

2.4 Significant Interval Discovery Approaches

The approach that uses significant intervals for time-series data was proposed by Srinivasan and Shrestha [15, 16, 23]. Srinivasan [15, 23] used a main memory apriori style algorithm for discovering significant intervals. An SQL-based approach was also developed for significant interval discovery (SID) by Shrestha [16]. In [15, 23], the authors propose a suite of apriori style, level-wise main-memory algorithms. In [16], the author proposes a level-wise SQL-based approach. Both the algorithms work on folded data rather than the original data. However, these approaches make multiple passes over compressed data as they follow the classic apriori class of algorithms. Our algorithm OnePass-SI, is a main-memory approach which discovers significant intervals in a single pass only.

2.5 Frequent Episode Discovery

The approach that discovers frequent episodes from significant intervals in time-series data was proposed by Srinivasan [15]. In [15], the author proposes an apriori style, level-wise SQL-based algorithm (Hybrid Apriori) to discover frequent episodes. However, this approach makes multiple iterations as it follows the classic apriori class of algorithms. Our algorithm OnePass-FED, is a main-memory approach which discovers frequent episodes in a single pass only.

CHAPTER 3

SIGNIFICANT INTERVAL DISCOVERY

This chapter describes a new algorithm for discovering significant intervals in time-series data which is the first phase of the overall problem.

3.1 Significant interval representation

Traditional time-series data can be represented with an event timestamp model. An event e (for example Light ON, Fan OFF, etc.) is associated with a sequence of timestamps $\{T_1, T_2, \cdots, T_n\}$ that describes its occurrences over a period of time. The notion of periodicity (such as daily, weekly, monthly, etc.) is used to delimit the scope of information we are interested in mining. For example, if we want to automate the devices using only 24 hour period(without considering the differences between days of the week) then daily periodicity is used. If we are interested in differentiating automation between weekdays and weekends, for example, then weekly periodicity is used (as there will be loss of information in daily periodicity). Given a periodicity, we group the event occurrences using that periodicity. For each event, the number of occurrences at each time point can be obtained by grouping on the timestamp and event (if multiple events are together). We term the number of occurrences of each event as event count (ec). Thus the time-series data can be represented as <e $\{T_1, O_1\}$, $\{T_2, O_2\}$, $\{T_3, O_3\}$, $\cdots$, $\{T_n, O_n\}$>, where T_i represents the timestamp associated with the event e and O_i represents the number of occurrences. O_i is referred as the event count of the event e at T_i.

An interval is sequence of time points with a clearly defined start and end point. A point is treated as an interval with one time point. As an interval has a number of

time points, the event count of an interval is the sum of the event counts at each point in that interval. A point and an interval are also associated with a *confidence*. *Confidence* of an interval is the percentage of the total event count over the period of data collection in terms of periodicity (number of days or weeks).

We define a *Window* as a time interval w[T_s, T_e] where (T_s <= T_e), T_s is the start time and T_e is the end time of the interval. An interval associated with an event is represented as A 6 tuple – [T_s, T_e, ec, l, d, c] – where ec represents total event count of the interval, l denotes the length of the interval (T_e-T_s+1), d indicates the density (computed as ec/l), and c represents confidence of the interval (computed as ec/N*100). N is the number of units of time-series data in days, weeks, months, etc.

3.1.1 Significant Interval

Given a time sequence T, periodicity, confidence `min-conf` and interval length `max-len`, we define an interval w[T_s, T_e] as a *significant interval* in T if:

1. confidence (c) of w, >= `min-conf`,
2. length (l) of w, <= `max-len`, and
3. there is no window w‘=[T_s‘, T_e‘] in T such that T_s‘>=T_s and T_e‘=<T_e and conditions (1) and (2) hold. In other words, a significant interval cannot be embedded in another significant interval. Essentially, we are looking for smallest or tightest intervals that satisfy the confidence and length given by a user. Our aim is to find all significant intervals that are present.

3.1.2 Valid Significant Interval

Significant intervals can be of unit size, overlapping or disjoint. All valid intervals should be discovered by a significant interval discovery (SID) algorithm. Figure 3.1 shows all possible valid significant intervals.

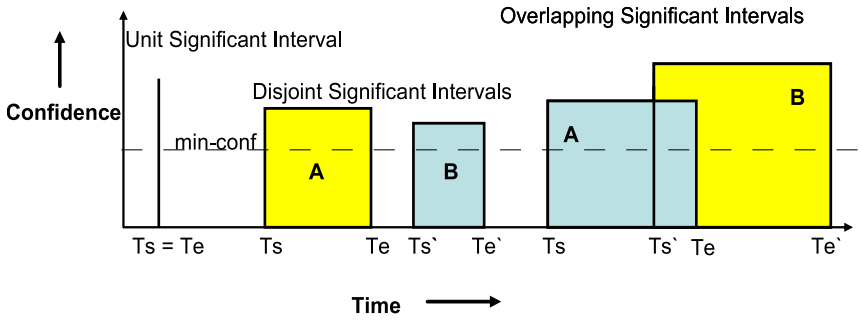

Figure 3.1 Valid Significant Intervals

1. **Unit Significant Interval:** It is a significant interval with the same start and end time; that is, $T_s = T_e$.
2. **Disjoint Significant Intervals:** They are defined as two significant intervals (that is, w$[T_s, T_e]$ and w‘$[T_s‘, T_e‘]$), which do not overlap (that is, $T_s‘ >= T_e$ and $T_e‘$ is not in the interval $[T_s, T_e]$ or $T_e‘> T_s$ and $T_s‘$ is not in the interval $[T_s, T_e]$).
3. **Overlapping Significant Intervals:** They are defined as two significant intervals w$[T_s, T_e]$ and w‘$[T_s‘, T_e‘]$ if ($T_s <= T_e‘ < T_e$ and $T_s‘ < T_s$) or ($T_s < T_s‘ <= T_e$ and $T_e‘ > T_e$).

3.1.3 Invalid Significant Interval

An invalid significant interval is identified by the third condition of the definition for a significant interval; that is, a significant interval cannot have an embedded valid significant interval (including a unit significant interval). Figure 3.2 gives an example of an invalid significant interval. A is not a significant interval as it contains another significant interval B. Hence, only B is a significant interval. In other words, B is the tightest significant interval. For example, consider a significant interval A obtained by combining the time points 10:10 and 10:20 which have event count of 2 and 10 respectively. Another significant interval B is obtained due to the time point 10:20 (that is, it is a unit

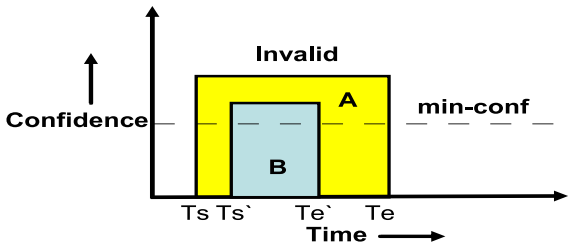

Figure 3.2 Invalid Significant Interval

length significant interval). As we can see, A forms a significant interval due to a major contribution of event count from time point 10:20. A would not have been a significant interval had it not been for time point 10:20. B also forms a significant interval due to the event count from time point 10:20. We are interested in discovering only the tightest significant intervals. Hence, we conclude that, A is not a valid significant interval and B is the only valid tightest significant interval.

3.2 OnePass-SI Algorithm

In this section, we present the OnePass-SI algorithm that discovers all significant intervals with a single pass over a time-series data set. The significant interval discovery algorithm has 2 phases: preprocessing and significant interval discovery.

3.2.1 Preprocessing

1. **Folding:** In the preprocessing phase, the input data set is combined to a periodicity of interest (example, daily or weekly). This process is called as **folding**. In our data ets from MavHome, the database contains the device name, its status (ON/OFF), and timestamp of the state change (and this combination is considered as a unique event). The tables in Figure 3.3 illustrate the information about Lamp1, that is, the time points at which it was turned 'ON' and the number of times (over the

Table 1 Input Dataset

Device	Status	TimeStamp
Lamp1	ON	8/10/2005 1:10
Lamp1	ON	8/11/2005 1:15
Lamp1	ON	8/11/2005 1:50
Lamp1	ON	8/12/2005 1:20
Lamp1	ON	8/13/2005 1:20
Lamp1	ON	8/14/2005 1:15
Lamp1	ON	8/15/2005 1:20
Lamp1	ON	8/15/2005 1:50
Lamp1	ON	8/16/2005 1:15
Lamp1	ON	8/16/2005 1:50

Table 2 Input Dataset after folding

Device	Status	Time Of Occurrence	Event Count
Lamp1	**ON**	**1:10**	**1**
Lamp1	**ON**	**1:15**	**3**
Lamp1	**ON**	**1:20**	**3**
Lamp1	**ON**	**1:50**	**3**

Figure 3.3 Input data set and corresponding folded data

entire data set) it was turned 'ON' at that time point. For example, Lamp1 has been turned 'ON' at 1:15 three times (on three different days) in the data set (of six days).

2. **Time Wrapping:** The compression of data due to folding should occur without the loss of any information of interest. As we are looking for intervals, they can span the periodicity boundary. For example, some intervals might span two days or weeks. Consider a situation where an inhabitant of the Smart Home operates his TV between 11:30 p.m. to 12:30 a.m. every night. Then, the interval 11:30 p.m. to 12:30 a.m. becomes a significant interval for automation. However, folding of data on a daily basis will not lead to identification of such interval and some interesting intervals that span two days or weeks may not be discovered at all. This problem is overcome in our approach by means of `time wrapping`. Time wrapping allows the formation of intervals that spans two days, two weeks, two months, etc. For example, for daily periodicity, all time points at the beginning of the day up to the `max-len` are added at the end with a different (next) day value. If `max-len` is specified as 30 minutes, then time points, from 00:00 through 00:30

a.m. are added at the end and can participate in the intervals spanning two days. For weekly periodicity, Sunday's values are padded with early Monday values. For other weekdays, time wrapping is already present as we need Tuesday's value for Monday and so on. Only for Sunday we need to time wrap Monday's value using the next week notation.

3.2.2 Interval Formation

Once the time wrapping is done, the OnePass-SI algorithm is applied to discover significant intervals. In this subsection, we briefly describe the motivation behind developing the algorithm, its description, provide the pseudo code, and explain the working of the algorithm with an example. An analysis of the algorithm is given in the next sub-section.

1. **Motivation:** The current significant interval discovery algorithms follow the classic apriori style, that is, the next level significant intervals are obtained by joining/combining two copies of the previous level significant intervals. This method, though effective, leads to the formation of a large number of intermediate intervals. If one could avoid the generation of intermediate intervals and still generate all the significant intervals correctly, the number of passes as well as memory usage can be reduced significantly. With this in mind, we wanted to develop a main memory algorithm which forms significant intervals directly and is memory efficient as well. An algorithm which merely forms significant intervals in a single pass was not the only motivation. The algorithm should also be scalable. That is, its memory usage should be independent of the size of the data set and the number of distinct events present in the data set. In summary, the motivations for developing the OnePass-SI algorithm were: i) need for non-apriori style algorithm that is scalable, ii) need for

a faster main memory algorithm, and iii) an algorithm which produces the same output as the other current significant interval discovery algorithms.

The basic idea behind the algorithm is to traverse the data set once using two pointers, one to indicate the beginning of a significant interval and the other to indicate the end of the significant interval. This will identify all the significant intervals in a single pass. Of course, some tests for unit significant intervals and embedded significant intervals need to be done along the way. By the time the first pointer reaches the end of the data set, we would have identified all the significant intervals.

2. **Description:** To begin traversing the folded data, we start from a the first time point. Then we traverse forward to combine with the adjacent time point to see if it forms a significant interval. This forward traversing continues until a significant interval is discovered or a stop condition is reached. When a significant interval is discovered, it is stored. Based on the definition of significant interval (in Section 3.1), a significant interval is represented using a start time, an end time, an event count, density, interval length, and confidence. Hence for storing this significant interval, we need to keep track of the start and end time of the interval. In other words, first we need to keep a track of the point where we started to combine the time points and second we need to keep a track of the end point where the significant interval was discovered. Hence, we use two pointers in the OnePass-SI algorithm. The first pointer is used to keep track of the start of the significant interval. The second pointer traverses the folded data in ascending order of time, starting from the position of the first pointer and the second pointer moves ahead combining the individual time points until it finds a significant interval or it has reached the maximum interval-length specified by the user or there are no more time points to be considered.

3. **Pseudo Code for OnePass-SI algorithm:** The pseudo code for the interval formation (OnePass-SI algorithm) is shown in Algorithm 1.

Algorithm 1 OnePass-SI algorithm

```
1:  Start with the first time point obtained after folding.
2:  while all time points not considered do
3:      if significant interval formed then
4:          if previously discovered significant interval subsumes current significant interval then
5:              Remove previously discovered significant interval.
6:          end if
7:          Add current significant interval.
8:          Start next interval formation from time point where current significant interval formation had started.
9:      else
10:         Combine with adjacent time point to form a bigger interval.
11:         if interval bigger than user specified maximum interval length then
12:             Restart with next time point following the time point where current interval formation began.
13:         end if
14:     end if

15: end while
```

4. **Description of the Pseudo-code:** The significant interval discovery starts from the first time point in the folded data as shown at line 1. The while loop at line 2 iterates until all the time points have been considered as starting point to form a significant interval. Lines 3 to 8 are executed when a significant interval is discovered. When a significant interval is discovered, we need to check if it is subsumed by the previously discovered significant interval. This is checked in the if condition at line 4 which if true removes previously discovered significant interval as seen in line 5. Lines 7 and 8 store the significant interval and arrange the two pointers to start discovery of next significant interval, respectively. Lines 9 to 14 are executed if moving the second pointer does not cause formation of a significant interval. Line 10 causes the second pointer to move to adjacent time point and

form a bigger interval. Line 11 checks whether the user specified maximum interval length has reached or not. If line 11, evaluates to true then the first pointer moves to adjacent time point to start formation of interval from that point.

5. **Example:** Consider a 15 day data set which gives a folded table as given in Figure 3.3. For 40% minimum confidence (`min-conf`), maximum interval length (`max-len`) of 20 minutes and `Daily` periodicity, the algorithm performs the following steps:
   (a) The algorithm starts significant interval discovery at time point 1:10 and combines it with time point 1:15 and computes a confidence of 26.67%. Since the required 40% of confidence is not achieved by combining these two time points, the next time point 1:20 is combined with this time interval. This leads to the discovery of a significant interval (say A) with start time of 1:10, end time of 1:20 and confidence of 46.67%.
   (b) Next, the algorithm starts significant interval discovery at time point 1:15 and combines it with time point 1:20 leading to the discovery of a significant interval (say B) with start time of 1:15, end time of 1:20, and confidence of 40%. Since the previous significant interval A subsumes the significant interval B, A is removed from the significant interval list.
   (c) Next, the algorithm starts significant interval discovery at time point 1:20 and combines it with time point 1:50 to give a confidence of 40% with interval length of 31 minutes. Even though this interval satisfies the constraint of `min-conf`, it is not a significant interval because it does not satisfy the `max-len` constraint.
   (d) Next, the algorithm starts significant interval discovery at time point 1:50 which cannot be further combined with any more time points. Since all time

points in the data set have been considered there are no more time points and the algorithm stops.

3.2.3 Analysis of OnePass-SI algorithm

1. **Complexity:** To calculate the complexity of the algorithm, we have to consider the iterations carried out by the two loops of the algorithm which represent the outer and inner pointers. The outer loop visits the entire folded data set at least once whereas the number of data points visited by the inner loop depends purely on the characteristics of the data, `max-len` and `min-conf` values. Hence, the complexity of the algorithm depends on these two loops. Since the number of data points visited by the inner loop is not constant, we shall discuss the complexity of the algorithm using the worst case and best case scenarios.

   In the best case, the complexity of the algorithm will be $\Omega(n)$ where n is number of data points obtained after folding. In this case, the inner loop will visit just one data point. This will be the case in two situations: i) all significant intervals discovered are of unit length or, ii) no significant intervals are discovered as all intervals violate the `max-len` and/or `min-conf` constraints.

   In the worst case, the complexity of the algorithm will be $O(n^2)$. In this case, the inner loop will visit all the data points in the folded data for each outer loop iteration. However, the worst case situation is a rare occurrence, because the `max-len` will restrict the forward traversal of the inner loop as not all data points will fall within the `max-len`. Furthermore, due to the sparse characteristics of the data set, it is really very rare that for all inner iterations, the entire data set is traversed each time. Hence, the algorithm runs between $O(n)$ and $O(n^2)$ complexity with the complexity being closer to $O(n)$. Note that n is dependent on periodicity and is a fixed value for a given periodicity. Even if the data is retrieved from a

secondary storage (e.g., file or disk), the number of I/O's are bounded by n (or less assuming that a page will fit more than 1 record). Furthermore, once the data is retrieved from a data storage device, it is cached in main memory. Since the points considered are adjacent to each other, only one retrieval of data from data storage is needed. Hence, the maximum number of I/O's (disk acesses) needed is O(m) where m is the number of pages needed to hold n data points.

2. **Scalability:** OnePass-SI algorithm is a main memory approach. Hence, memory usage and its relationship with input data size is critical for the scalability of the algorithm. The size of the raw data set is not a problem for OnePass-SI algorithm as the memory need does not increase proportionally. The preprocessing step of **folding** (the preprocessing step introduced at the start) significantly reduces the amount of data in a data set. The maximum number of time points which are possible after folding is known. For example, for `daily` periodicity and `minutes` granularity, the maximum number of time points possible are 24*60 = 1440. Similarly, maximum number of time points for `daily/seconds` granularity is 86400, `weekly/minutes` granularity is 10080, `weekly/seconds` granularity is 604800. These are the maximum number of time points which are possible after folding. Furthermore, the actual number of time points obtained from real-world data sets is likely to be much lesser than these numbers. At run time, it is not necessary to hold the entire data set in memory. Only `max-len` data set needs to be held in memory which is much smaller than the folded data set. Hence, memory scalability is not an issue in OnePass-SI algorithm.

   If the data is collected for a number of devices, OnePass-SI algorithm can be applied to each device independently. The resulting significant intervals for each device can be combined and sorted on the start time of the interval detecting frequent events/patterns.

3. **Adapting OnePass-SI algorithm to other domains:** OnePass-SI algorithm has been developed for time-series data. Instead of binary values associated with each event (example, ON/OFF), it is possible to have a numerical or categorical value. Examples of such data include: i) number of people accessing a particular site at a particular interval of the day, ii) number of cars passing through a sensor in a minute interval over weekdays, weekends, and so on. In such situations, minimal to the algorithm might be required. With numerical or categorical value, folding needs to take the value into consideration. Values at a time point can be aggregated in different ways during folding based on the domain semantics (example, average, minimum, maximum). For example, if traffic data is been analyzed average, minimum, and maximum may provide different ways of looking at the congestion at a particular point or interval with respect to capacity.

   Another possible variant is the way in which the confidence of an interval is calculated. Currently, we take the percentage of the event count of the interval with respect to the number of units of the data set (days, weeks, etc.) and is computed as ec/N*100. Another possible way of defining confidence would be to take a percentage of the event count with respect to the number of units of the data set as well as the length of the interval and can be computed using the formula ec*100/(N*l). With these changes, the definition of a significant interval will have to be modified, the process of merging/formation of a significant interval would have to be redefined. As seen from the above discussions, the OnePass-SI algorithm can be modified in different ways to suite different application domains.

4. **OnePass-AllSI Algorithm:** With the OnePass-SI algorithm, significant intervals are discovered for a given confidence and maximum length. If we want to discover significant intervals for another value of maximum length, we need to execute the algorithm again using the new maximum length. As prediction and mining algo-

rithms involve exploration of the data set for different input parameters, it is useful to compute as much as possible in one execution if it can be re-used for other parameter values. OnePass-AllSI algorithm accomplishes that with respect to the maximum length parameter. Again the basic idea is to compute significant intervals without using the max-len parameter. This will identify significant intervals of all sizes with slightly larger computation effort. Once all significant intervals are identified, we can extract significant intervals that are smaller than a specified max-len without much effort.

The pseudo code for the OnePass-AllSI algorithm is shown in Algorithm 2.

Algorithm 2 OnePass-AllSI algorithm

```
1:  Start with the first time point obtained after folding.
2:  while all time points not considered do
3:      if significant interval formed then
4:          if previously discovered significant interval subsumes current significant interval then
5:              Remove previously discovered significant interval.
6:          end if
7:          Add current significant interval.
8:          Start next interval formation from time point where current significant interval formation had started.
9:      else
10:         Combine with adjacent time point to form a bigger interval.
11:         if last time point reached then
12:             Restart with next time point following the time point where current interval formation began.
13:         end if
14:     end if
15: end while
```

Description of the Pseudo-code: The significant interval discovery starts from the first time point in the folded data as shown at line 1. The while loop at line 2 iterates until all the time points have been considered as starting point to form a significant interval. Lines 3 to 8 are executed when a significant interval

is discovered. When a significant interval is discovered, we need to check if it is subsumed by the previously discovered significant interval. This is checked in the if condition at line 4 which if true removes previously discovered significant interval as seen in line 5. Lines 7 and 8 store the significant interval and arrange the two pointers to start discovery of next significant interval, respectively. Lines 9 to 14 are executed if moving the second pointer does not cause formation of a significant interval. Line 10 causes the second pointer to move to adjacent time point and form a bigger interval. Line 11 checks whether the last time point is reached. If line 11 evaluates to true then, the first pointer moves to adjacent time point to start formation of interval from that point.

Example: Consider a 15 day data set which gives a folded table as given in Figure 3.3. For 40% minimum confidence (`min-conf`), and `Daily` periodicity, the algorithm performs the following steps:

(a) The algorithm starts significant interval discovery at time point 1:10 and combines it with time point 1:15 to give a confidence of 26.67%. Since the required 40% of confidence is not achieved by combining these two time points, the next time point 1:20 is combined to this time interval. This leads to the discovery of a significant interval (say A) with start time of 1:10, end time of 1:20 and confidence of 46.67%.

(b) Next, the algorithm starts significant interval discovery at time point 1:15 and combines it with time point 1:20 leading to the discovery of a significant interval (say B) with start time of 1:15, end time of 1:20 and confidence of 40%. Since the previous significant interval A is subsuming significant interval B, A has to be removed from the significant interval list.

(c) Next, the algorithm starts significant interval discovery at time point 1:20 and combines it with time point 1:50 to give a confidence of 40% with interval

length of 31 minutes. As seen previously, when we used OnePass-SI algorithm for significant interval discovery, even though this interval satisfied the constraint of `min-conf`, it was not discovered as a significant interval because it did not satisfy the constraint of `max-len`. But now since we are using the OnePass-AllSI algorithm for significant interval discovery, this interval is classified as a significant interval with start time of 1:20, end time of 1:50 and confidence of 40%.

(d) Next, the algorithm starts significant interval discovery at time point 1:50 which cannot be further combined with any more time points. As there are no more time points, that is, all the time points have been considered the algorithm stops.

5. **Comparison with other approaches:** As [15, 16] discover significant intervals over time-series data, we shall give a comparison of our approach with them. Both use the following strategy: i) form intermediate intervals by combining with adjacent intervals/time points only, ii) from these intervals, remove the significant intervals, iii) use the remaining intermediate intervals to form larger intervals. In contrast, the OnePass-SI algorithm finds significant interval by combining the time points till it finds a significant interval, or a significant interval cannot be formed. For this, it uses two pointers, one to make sure that all the time points are considered as start point and second to combine with adjacent time points till a significant interval is not formed or an end is reached. There is no formation of intermediate intervals in OnePass-SI algorithm. Hence, it will take much less time to execute as compared to other main-memory and SQL counterparts. This is also evident by the experimental results.

3.3 Improvement to SQL-based Significant Interval Discovery Algorithm

In this section, we discuss the improvements which have been made to the SQL-based significant interval discovery algorithm proposed by Shrestha [16]. In the following subsections, we shall first describe the motivations that lead to the need for improving the original algorithm. Then, we shall briefly describe the improvements made to the algorithm.

3.3.1 Motivation

The SQL-based significant interval discovery algorithm uses the apriori style to obtain the next level significant interval by joining significant intervals from the previous level until no more significant intervals can be formed. The number of iterations taken by the algorithm is data dependent. While analyzing the execution of all the stages in the SQL-based significant discovery algorithm it was discovered that, the first level significant interval generation takes a disproportionate amount of time. The time taken by the first level significant interval generation is high (70% to 80%) as compared to the combined time taken by the rest of the program. Hence, optimizing the first level significant interval generation was very important to improve the performance of this SQL algorithm.

3.3.2 Improvements

1. **Analysis:** On analyzing the first level interval generation the following reasons were identified as the reason why it takes such a huge amount of time.
   (a) even though we reduce the amount of data to be processed by folding, the amount of data used to generate the first level significant intervals is large when compared to later levels,

(b) the first level significant intervals are generated by joining three tables, two of which contain the folded data and the third which is created from the folded data with additional information which will be used when joining the three tables. This join turns out to be very expensive because, there is a large number of folded data and three joins are used.

2. **Possible Solutions:** After analysis, it is evident that the data size cannot be reduced further during folding. Hence reducing the number of joins is worth exploring.
3. **Proposed solution:** The number of joins in the algorithm goes to three because we are trying to merge the adjacent tuples to obtain the first level significant intervals. To do this a table which identifies all the distinct start points and their minimum time difference is created. This table is created by joining two folded data tables. All this could have been avoided if there were a mechanism in SQL to identify the rows of a table by a row number (as we assume folded data to be sorted on time). By using this number it would have been easier to find the adjacent row (tuple) to merge. This feature is available in the most popular RDBMS's but their usage and properties differ depending on the RDBMS. The figure 3.4 gives a summary of what can be done depending on the RDBMS used.
4. **Implementation Details:** We have implemented the solution for Oracle only. It can also be applied to other RDBMS. The folded and time wrapped data is stored in *countsuptemp* table. First, we create a table *newdata* with the following statement

   CREATE *table newdata as*

   SELECT *rownum as ID, txtdeviceid, txtstatus, dttime, recordcount*

   FROM *countsuptemp;*

RDBMS	Way to get Row Number
Oracle	Using **rownum** which automatically gives rownumber for a row.
SQL Server 2005	Using **ROW_NUMBER()** function.
SQL Server 2000	**Dynamically** obtaining row number.
DB2	Using **ROW_NUMBER()** function.

Figure 3.4 Methods of obtaining row number for different RDBMSs

This statement creates a table to store the folded data but with the added attribute *id* corresponding to the row number. Next, we obtain the first level intervals by doing a self join on this table. The SQL statement is given below

INSERT *into myfirstleveltemp*

*(****SELECT*** *t1.txtdeviceid, t1.txtstatus, t1.dttime, t2.dttime,*

*((t2.dttime-t1.dttime)*24*60)+1,*

t1.recordcount+t2.recordcount,

t1.recordcount, t2.recordcount,

round((t1.recordcount+t2.recordcount)/N, 3),

*round((t1.recordcount+t2.recordcount)/(((t2.dttime-t1.dttime)*24*60)+1),3)*

FROM *newdata t1, newdata t2*

WHERE *t2.id = (t1.id)+1*

AND *(((t2.dttime-t1.dttime)*24*60)+1) <= max-len*

AND *t1.txtdeviceid = t2.txtdeviceid*

AND *t1.txtstatus = t2.txtstatus);*

Data Set	No. of Days	No. of Tuples	No. of Events
1	100	27K	2
2	120	40K	3
3	150	54K	4
4	150	67K	5
5	180	80K	6
6	200	94K	7

Figure 3.5 Data sets used for Experiments

Note that in the above, adjacent time points are compared using id attribute. With the use of id attribute we avoid generating a table and the number of joins is further reduced to 2.

3.4 Experimental Results

In this section, we shall discuss the experiments performed and the conclusions which can be derived from them.

Data sets collected by the MavHome project were used for some experiments. Synthetic data sets were also generated to verify the correctness of the algorithm. The MavHome data set consists of timestamps along with device characteristics for all devices deployed in the MavPad. When a device changes its status, a record is inserted into the database with its associated timestamp.

A summary of the data sets used for the experiments is seen in Figure 3.5.

The algorithm was implemented using Java and the database used was Oracle 9i. The experiments were conducted on a Linux machine (running on dual processors with 2.4 GHz CPU speed and 2 GB memory) of the Distributed and Parallel Computing Cluster at UTA.

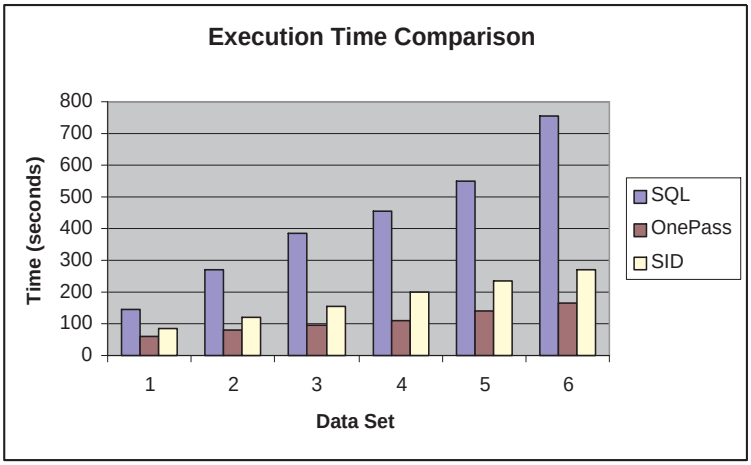

Figure 3.6 Experimental Results (Daily Periodicity)

Data Set	OnePass	SQL
	Time (seconds)	
1	153.55	6318.7
2	217.04	10517.1
3	309.89	12724.58
4	369.93	16130.21
5	440.92	19315.17
6	479.78	22665.57

Figure 3.7 Experimental Results (Weekly Periodicity)

3.4.1 Comparison of SID, SQL and OnePass-SI algorithms

Several experiments were conducted to compare the OnePass-SI algorithm with the SID [15] and SQL-based [16] approaches. The time taken by these algorithms formed the basis of comparison. The time taken was averaged over three runs for each data set.

Figure 3.6 and Figure 3.7 gives the comparison of the execution time for the different approaches. The parameters for the experiments were `min-conf`: 60% and `max-len`: 50 minutes. Experiments compared in figure 3.6 use `daily` periodicity and those shown in figure 3.7 use `weekly` periodicity. As seen from these figures, OnePass-SI algorithm takes

order of magnitude less time to execute AS compared to the SID main memory and SQL algorithms. For daily periodicity experiments shown in figure 3.6, the OnePass-SI algorithm shows a performance improvement varying from 20% to 50% over SID algorithm and 140% to 350% improvement over SQL algorithm depending on the data set. For weekly periodicity experiments seen in figure 3.7, the OnePass-SI algorithm shows a performance improvement of 4000% to 4600% over SQL algorithm depending on the data set. This is due to the fact that OnePass-SI algorithm uses a non-iterative approach as compared to SID and SQL algorithms. Several other experiments have been performed to observe the effect of `max-len` and `min-conf`. In general, OnePass-SI algorithm outperforms SID and SQL algorithms.

3.4.2 Effect of varying parameters on OnePass-SI algorithm

Effect of max-len: In these set of experiment we want to test the effect of the changing the `max-len` user parameter for the same `min-conf` and periodicity. The chart seen in figure 3.8 shows the effect of varying the `max-len` for a set of experiments keeping the `min-conf` = 60% and periodicity as `Daily`.

As seen from the figure 3.8 as the `max-len` increases the time taken to discover significant intervals also increase. This is because, as the `max-len` increases the number of time points which are checked for forming significant intervals also increases. The number of significant intervals discovered also increases which is seen in figure 3.9.

3.4.3 Comparison of OnePass-SI and OnePass-AllSI algorithms

Comparison of number of significant intervals: A set of experiments were carried out to compare the number of significant intervals discovered by both the OnePass-SI and OnePass-AllSI algorithms. The significant intervals discovered by the OnePass-AllSI algorithm are a superset of the significant intervals discovered by the OnePass-SI

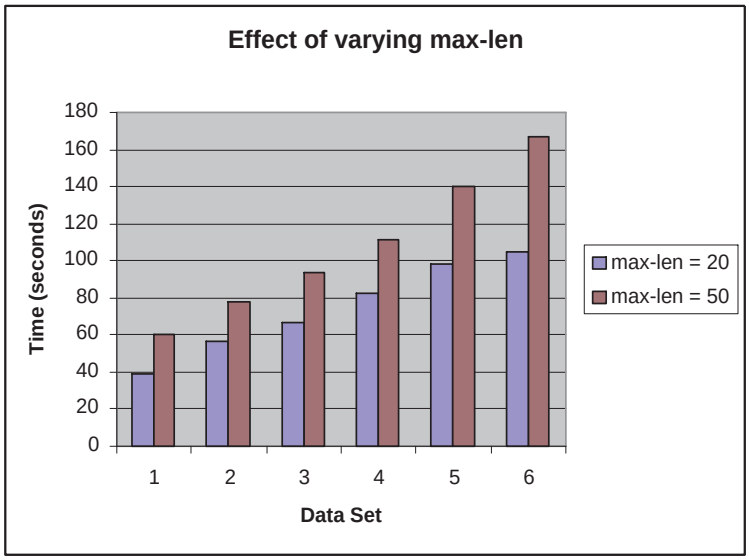

Figure 3.8 Experiments varying max-len

algorithm. This is because the OnePass-AllSI algorithm does not have a limitation on the `max-len`, a significant interval can have. The parameters for the experiments were `min-conf`: 60% and `max-len`: 20 minutes.

As we can see from Figure 3.10, the OnePass-AllSI algorithm finds additional significant intervals as compared to the OnePass-SI algorithm. Upon inspection of the significant intervals discovered by both the algorithms, it was found that the additional

Data Set	No. of Significant Intervals	
	max-len = 20	max-len = 50
1	766	1942
2	1149	2841
3	1532	3780
4	1915	4725
5	2298	5586
6	2681	6468

Figure 3.9 Comparison of significant intervals

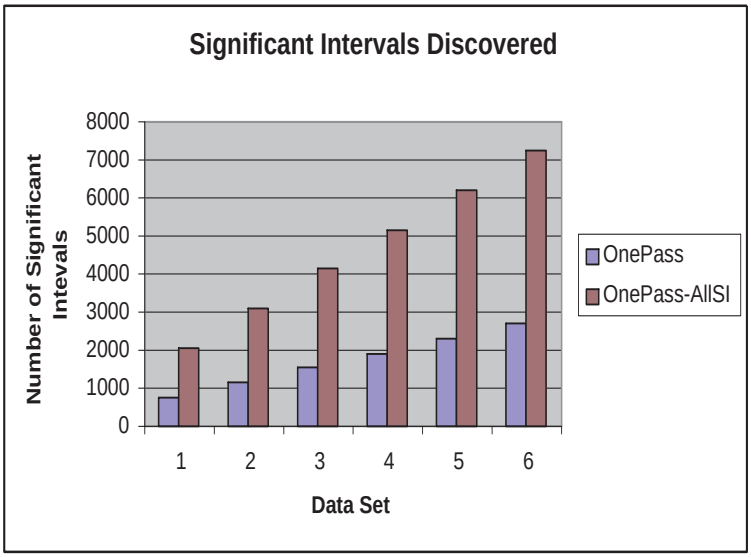

Figure 3.10 SID's discovered for OnePass-SI and OnePass-AllSI algorithms

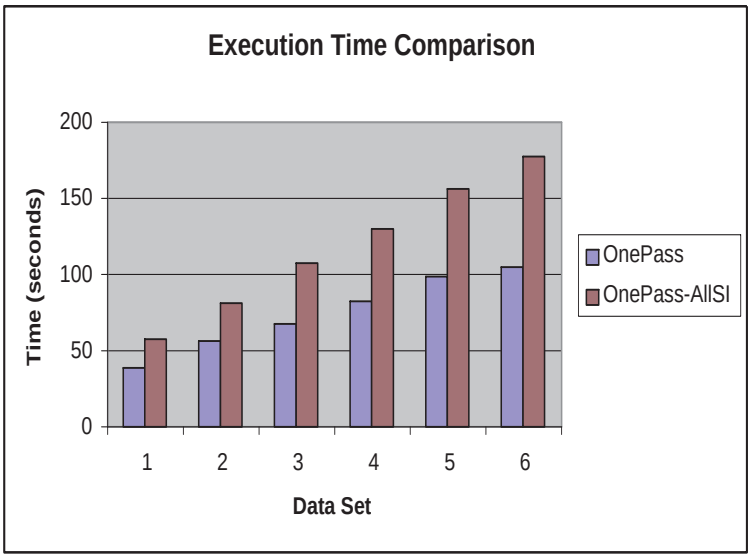

Figure 3.11 Time comparison for OnePass-SI and OnePass-AllSI algorithms

significant intervals discovered by the OnePass-AllSI algorithm were those which had interval length greater than 20 minutes.

Time comparison: As the OnePass-AllSI algorithm produces a superset of significant intervals as compared to its OnePass-SI algorithm counterpart, we wanted to check the extra time taken for finding all significant intervals of different lengths. A set of experiments were carried out to analyze the difference in the time taken by each of these algorithms.

As seen from Figure 3.11, the OnePass-AllSI algorithm takes 40% to 70% more time than the OnePass-SI algorithm. Even though OnePass-AllSI algorithm takes more time, it will be beneficial to use OnePass-AllSI algorithm if we want to explore the data set for different `max-len` values. The execution time difference is also dependent on `min-conf` selectivity. These significant intervals can be used when a user gives a different call for significant intervals of a different `max-len` but same `min-conf`. This will help reducing computational time and increase response time to user calls.

3.4.4 Comparison of SQL-based SID algorithms

In these set of experiments, the SQL-based significant interval discovery algorithms (that is, the older version designed and developed by [16] and the modified version discussed in this chapter) are executed to compare their performance.

Figure 3.12 gives a comparison of the time taken by SQL (original) algorithm versus SQL (modified) algorithm. As seen from the figure, the modified algorithm takes much lesser time as compared to the original algorithm. There is a performance improvement in between 60% - 80%.

Figure 3.12 Comparison of original and modified SQL SID algorithms

CHAPTER 4

FREQUENT EPISODE DISCOVERY

Frequent Episode Discovery (FED) is performed using the significant intervals discovered by the significant interval discovery algorithms discussed in the previous chapter. This chapter describes a new algorithm for frequent episode discovery in time-series data which is the second phase of the overall MavHome problem.

4.1 Sequential Mining

The sequential associations or sequential patterns can be represented as follows: When A occurs, B also occurs within a certain time. Few examples of sequential patterns of interest which can occur in MavHome are mentioned below:

'Every morning Roger turns on the light and the fan between 10 a.m. and 10:15 a.m.'

'Every evening between 6 p.m. and 6:30 p.m., Sam turns on the drawing room light and the television to watches CNN news.'

'Every Tuesdays and Saturdays, between 2 p.m. and 3 p.m., Judy turns on the laundry machine and the lights in the laundry room.'

From these examples, we can observe that the frequent episodes of interest relate to a group of devices with which a MavHome inhabitant interacts. Additionally, these frequent episodes occur during the same time interval with sufficient periodicity. In the next section, we shall describe OnePass-FED algorithm which is developed to discover such frequent episodes.

4.2 OnePass-FED

In this section, we present the OnePass-FED algorithm that discovers frequent episodes from significant intervals.

4.2.1 Motivation

The Hybrid Apriori [15] approach makes several passes over the data set. This generates intermediate results for each iteration of the algorithm. The number of passes is determined by the number of devices in the data set or the stopping conditions where no new frequent episodes are generated. Hybrid Apriori uses an SQL-based approach to discover frequent episodes. An SQL-based approach has additional overhead of creating and managing intermediate tables. Additionally, date/time comparison is inefficient as they are string operations. Another disadvantage of using SQL is that duplicate frequent episode elimination requires a set of SQL statements to be executed. The reason for this will be seen in a later discussion. Additionally, the I/O's to the database in an SQL-based approach are very time consuming.

Based on the above observations and to address the above limitations, we wanted to explore a main memory algorithm in which all frequent episodes can be generated in a single pass. Some of the issues which had to be dealt with while designing this algorithm were: i) using the standard and specialized data structures available in programming languages, ii) designing an optimized algorithm which uses these data structures efficiently, iii) avoid using the classic Apriori style, iv) taking scalability into account, and v) reduce time taken for duplicate elimination. A pictorial representation demonstrates the difference between OnePass-FED algorithm (Figure 4.2) and Hybrid Apriori algorithm (Figure 4.1).

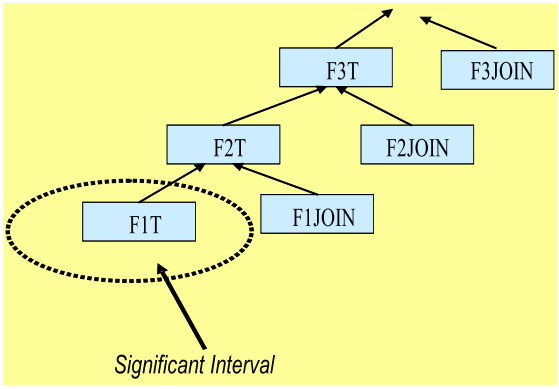

Figure 4.1 Hybrid Apriori

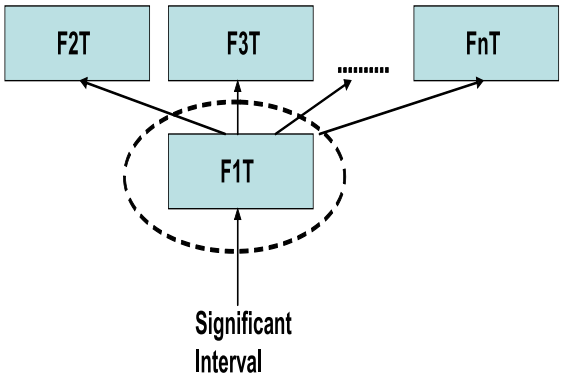

Figure 4.2 OnePass-FED

4.2.2 Description

We use the same concept of using two pointers as in the OnePass-SI algorithm. We union/append all the significant intervals of events that are relevant and have the same period. We also assume that the significant intervals (SIDs) have been discovered using the same periodicity. We then sort the significant intervals on their start time. Then the traversal of these significant intervals starts. Suppose at any point of time in the algorithm, frequent episode discovery from a particular significant interval has started. We then traverse forward and see if the adjacent significant interval can be combined to form a frequent episode. This forward traversing continues until the size of the frequent episode corresponds to the maximum distinct events or until we cannot

traverse any further. After this significant interval cannot be further used as base, the next frequent episode discovery has to start from the next significant interval where the previous episode discovery had started. For this, we need to keep track of the original start significant interval. Hence, we use two pointers in the OnePass-FED algorithm. The first pointer is used to keep track of the starting (base) significant interval used for frequent episode discovery. The second pointer traverses the significant intervals in ascending order, starting from the position of the first pointer and moves ahead discovering frequent episode until any of the previously mentioned stopping conditions occur.

4.2.3 Term Definitions

Here we shall describe some terms which are going to be used while describing the algorithm.

In the previous chapter, confidence of an interval had been defined for a single event or an episode. However, OnePass-FED algorithm detects frequent episodes. Hence, we need to define the confidence of a pattern using the confidence associated with the events forming the pattern. This is termed as `Pattern Confidence` (PC). The `Pattern Confidence` of an episode within an interval is defined as the minimum of the confidence of its constituent events. When several events occur with different confidence in an interval, we can only infer that all events occur with minimum confidence in that interval. Hence, the confidence of a pattern interval represents the minimum number of occurrence of an episode within an interval. With frequently occurring patterns, `Pattern Confidence` underestimates the actual probability of the events occurring together but retains its significance or order relative to the other patterns discovered.

Frequent episodes are discovered by combining/merging events. The parameters that define how this combination/merge takes place are `Sequential Window` and `Semantics`. `Sequential Window` defines the maximum allowed time within which events

in a frequent episode may start/end. The way in which this maximum allowed time is interpreted is defined by `Semantics`. There are two types of `Semantics` which a user may define: `Semantics-s` and `Semantics-e`. `Semantics-s` generates all possible combinations of events, which start within `Sequential Window` units of the first event. `Semantics-e`, on the other hand, generates combinations of events that start and end within the `Sequential Window` units of the first event. Most of the traditional sequential mining techniques deal with events that occur at a point and form all possible combination of events within an instance of a sliding window. Since points are replaced by intervals, the above two semantics need to be considered to form maximal sequences. The use of `Semantics-s` results in more sequences as compared with `Semantics-e` since events that occur with an interval greater than the `Sequential Window` will not participate in the generation of maximal sequences in `Semantics-e`.

The `Sequential Window` constraint of OnePass-FED algorithm automatically helps satisfy the subset property because of which the pruning based on the subset property is not explicitly performed. As an example, let A(1, 10), B(2, 5), C(7, 15), D(17, 25) be the significant intervals. The figures in the parenthesis indicate start time and end time of the significant intervals. Assuming a `Sequential Window` of 10 units and `Semantics-s`, the 2 event episodes discovered are AB(1, 10), AC(1, 15), BC(2, 15), CD(7, 25). The 3 event episode discovered is ABC (1, 15). A is combined with B because B started within 10 units of start of A. A is also combined with C because C started within 10 units of start of A. This automatically implies that B combines with C since B started after A.

4.2.4 Time Wrapping

The frequent episode discovery should occur without the loss of any information of interest. Some significant intervals might span two days or weeks (depending upon periodicity). Consider a situation where an inhabitant of the MavHome operates his

TV between 11:50 p.m. to 12:10 a.m. and his Radio between 12:05 a.m. to 12:20 a.m. every night. A frequent episode of TV and Radio should then be discovered in the time interval 11:50 p.m. to 12:20 a.m. for automation. However, using the significant intervals arranged in ascending order of time will not lead to frequent episode discovery of such episodes. Also, other interesting episodes that span two days or weeks may not be discovered at all. This problem is overcome in the algorithm by means of time wrapping. For example, for daily periodicity, all significant intervals that fall below the start of that day but lying below the `Sequential Window` are replicated and given a separate day value. If `Sequential Window` is specified as 30 minutes, then significant intervals, which have start time less than 00:30 can participate in the frequent episode discovery spanning across two days. For weekly periodicity, Sunday's significant intervals are padded with early Monday significant intervals. For other weekdays, time wrapping is already present. Only for Sunday we need to time wrap Monday's significant intervals using the next week notation.

4.2.5 Pseudo Code for OnePass-FED Algorithm

The Pseudo Code for the OnePass-FED algorithm is given in Algorithm 3.

4.2.6 Explanation of Pseudo Code for OnePass-FED Algorithm

The algorithm starts by obtaining the distinct events (n) from the significant intervals as seen in Line 1. These distinct events act as one of the stopping conditions for episode discovery which will be seen in the discussion to follow. In Line 2, the significant intervals are brought into memory ordered (ascending) by start time and end time. In Line 3, the significant intervals which start/end within `Sequential Window` range of the next day, week, month, etc. are added to the end of the data. Lines 4, 5 and 6 initialize the two pointers `start` and `next` to point to their respective positions. Line 7 initializes

Algorithm 3 OnePass-FED Algorithm

```
1:  Obtain distinct events (n).
2:  Obtain significant intervals arranged in increasing order of start time, end time.
3:  Add time wrapped data.
4:  Start with the first significant interval.
5:  start = next = first significant interval.
6:  next significant interval (that is, next++).
7:  frequent episode = event corresponding to the significant interval pointed by start.
8:  while all significant intervals are not considered by start pointer do
9:      if further progress not possible then
10:         start from next significant interval after current start(that is, start++)
11:         next = start
12:         frequent episode = event corresponding to the significant interval pointed by start.
13:     else
14:         if event pointed by next significant interval not present in frequent episode then
15:             if next significant interval forms a frequent episode then
16:                 Add event pointed by next significant interval to frequent episode.
17:                 Frequent episode discovered stored/displayed.
18:             end if
19:         end if
20:     end if
21:     next significant interval (that is, next++).
22: end while
```

the frequent episode to the event corresponding to the significant interval pointed by the start pointer. Line 8 is the while loop which terminates after all the significant intervals have been pointed by the start pointer once. Line 9 is an if condition which checks to see if further movement of the second pointer is possible or a new frequent episode discovery should be started. The conditions which prevent further movement of the second pointer are either of the following: i) the frequent episode size is equal to the maximum number of distinct events (n) obtained from Line 1 or ii) the maximum `Sequential Window` limit has been reached and hence it is no use moving the second pointer. Lines 10 through 12 are executed when the second pointer cannot be moved further and it is required to

Event	Start Time	End Time
LampON	15:00	15:15
FanON	15:10	15:20
TVON	16:00	16:10
LampON	16:00	16:10
FanON	16:03	16:12

Figure 4.3 Significant intervals discovered by SID

start a new frequent episode search. Line 10 moves the first pointer, that is, start by one significant interval. Lines 11 and 12 reinitialize next and frequent episode variables to their respective new values. Lines 13 through 20 are executed when the if condition at line 9 evaluates to false. Line 14 checks whether the event corresponding to the significant interval pointed by the `next` pointer is not present in the current frequent episode under consideration. Lines 15 through 18 are executed only when the if condition in Line 14 is true. Lines 16 and 17 add the event corresponding to the significant interval pointed by next to the current episode discovered and store the frequent episode and/or output the episode discovered if the `Sequential Window` constraint is satisfied.

4.2.7 Example

In this subsection, we shall explain the OnePass-FED algorithm with the help of an example. This will give a clearer view of the algorithm. OnePass-FED algorithm works on the significant intervals generated by the significant interval discovery algorithms. For this example, let us assume that the significant intervals have been generated using one of the significant interval discovery algorithms and are as seen in Figure 4.3.

For this example, we shall consider `Semantics-s` and `Sequential Window`-30 minutes. The OnePass-FED algorithm goes through the following steps:

1. The number of distinct events (n) is found as 3, that is, n = 3.

2. The OnePass-FED algorithm starts with first significant interval [LampON, 15:00, 15:15, 70], that is, start is pointing to this significant interval. The next pointer moves to significant interval [FanON, 15:10, 15:20, 80]. The start of this significant interval lies within `Sequential Window` units (that is, 30 minutes). Hence, these two significant intervals form a 2 event frequent episode [LampON FanON, 15:00, 15:20, 70]. The Pattern Confidence of the frequent episode is taken as the minimum of the two confidences which is 70 in this case. Now, the next pointer moves to significant interval [TVON, 16:00, 16:10, 75]. The `Sequential Window` constraint is violated. Hence, the next pointer will not move any further. None of the significant intervals after this significant interval will not satisfy the `Sequential Window` constraint. Hence, no more frequent episodes can be further formed with [LampON, 15:00, 15:15, 70] as the start (base) significant interval.
3. Now, the start pointer moves to the significant interval [FanON, 15:10, 15:20, 80]. The next pointer moves to significant interval [TVON, 16:00, 16:10, 75]. Again, the `Sequential Window` constraint is violated and no more frequent episode can be discovered with the current base significant interval.
4. Now, the start pointer moves to the significant interval [TVON, 16:00, 16:10, 75]. The next pointer moves to significant interval [LampON, 16:00, 16:10, 80]. Both these significant intervals combine to form a 2 event frequent episode [TVON LampON, 16:00, 16:10, 75]. The next pointer now moves to the significant interval [FanON, 16:03, 16:12, 70]. This combines with the current frequent episode to form a 3 event frequent episode [TVON LampON FanON, 16:00, 16:12, 70]. Now, as the value of n is 3, we can conclude that no more higher level frequent episode will be formed using the base significant interval [TVON, 16:00, 16:10, 75].

2 event frequent episodes				
Event1	Event2	Start Time	End Time	Confidence (%)
LampON	FanON	15:00	15:20	70
TVON	LampON	16:00	16:10	75
LampON	FanON	16:00	16:12	70

Figure 4.4 2 event frequent episodes

3 event frequent episode					
Event1	Event2	Event3	Start Time	End Time	Confidence (%)
TVON	LampON	FanON	16:00	16:12	70

Figure 4.5 3 event frequent episodes

5. Now, the start pointer moves to the significant interval [LampON, 16:00, 16:10, 80] and the process will continue. Finally, the frequent episodes detected when the algorithm stops are as seen in Figures 4.4 and 4.5.

4.3 Analysis of OnePass-FED algorithm

4.3.1 Complexity

To calculate the complexity of the algorithm, we have to consider the iterations carried out by the two loops of the algorithm which represent the outer and inner pointers. The outer loop visits the significant intervals at least once whereas the number of significant intervals visited by the inner loop depends purely on the characteristics of the data, `Semantics` type and `Sequential Window` value. Hence, the complexity of the algorithm depends on these two loops. Since the number of significant intervals visited by the inner loop is not constant, we shall discuss the complexity of the algorithm using the worst case and best case complexity scenarios.

In the best case, the complexity of the algorithm will be $\Omega(n)$ where n is total number of significant intervals of all events obtained after applying significant interval

discovery algorithm. In this case, the inner loop will visit just one significant interval. This case will take place when the number of distinct events is equal to 'one', that is, all the significant intervals are for a single event only. No frequent episode being discovered is another situation when the best case takes place. This happens when adjacent events violate the `Sequential Window` constraint, that is, no event Starts/Ends within `Sequential Window` units of the first event in the frequent episode.

In the worst case, the complexity of the algorithm will be $O(n^2)$. In this case, the inner loop will visit all the significant intervals for each outer loop iteration. However, the worst case situation is a rare occurrence, because the `Sequential Window` will restrict the forward traversing of the inner loop as not all significant intervals will fall within the `Sequential Window` stopping the inner loop from going any further. In addition, the number of distinct events also restricts the forward traversing of the inner loop, that is, the inner loop stops after the frequent episode size reaches the maximum size possible (which is the number of distinct events). Hence, it is very rare that for all inner iterations, the entire data set is traversed each time. Hence, in summary the algorithm has $O(n*m)$ complexity (n = number of total significant intervals; m = number of significant intervals visited by the inner loop). This complexity is more **closer** to the $O(n)$.

4.3.2 Scalability

OnePass-FED algorithm is a main memory approach. Hence, memory usage is critical for the scalability of the algorithm. As observed from the explanation of the working of the OnePass-FED algorithm, all the significant intervals need not be present in the main memory at the same time. Instead, only those significant intervals which lie `Sequential Window` units away from the first event in the episode, can be brought into the memory. However, this will require many probes to the database. In order to reduce the number of probes (that is, I/O) to the database, we can bring in a batch of

Event 1	Event 2	Start Time	End Time
Lamp ON	Fan ON	10:00 a.m.	10:10 a.m.
Fan ON	Lamp ON	10:00 a.m.	10:10 a.m.

Figure 4.6 Duplicates discovered by OnePass-FED

significant intervals which lie within `N * Sequential Window` of the first event of the frequent episode (N here is an integer which can be assigned a value between 1 and MaxN where, MaxN is an integer which makes `N * Sequential Window` value equal to the maximum time points possible based on the granularity and periodicity). The next batch of `N * Sequential Window` can be brought into the memory when the last significant interval of the current batch is encountered. Also, the value of N should be such that the number of probes to the database is small.

4.3.3 Duplicate Elimination

Duplicates are generated during the frequent episode discovery process. Examples of such duplicates are seen in Figure 4.6.

Both the frequent episodes in the table in Figure 4.6 convey the same information. It is only the order of event that has changed. We would like to eliminate such type of frequent episodes. The duplicate elimination can be done in two ways. First, since databases do not allow rearrangement of columns (only rows by using group by and order clauses), we transpose the rows of each frequent episode into columns, sort and reconstruct them to remove the duplicate frequent episodes (this is what is done in Hybrid Apriori algorithm). The second method is what OnePass-FED algorithm uses. We order the events of the frequent episode before inserting them to a database and then a GROUP BY is used to eliminate the duplicates. Hence, duplicate elimination in OnePass-FED algorithm is much more efficient as compared to Hybrid Apriori algorithm.

4.3.4 Specialized and Standard Data Structures

A very detailed explanation of the data structures which are used while implementing this algorithm is not given here. However, we would like to mention some of them. A user-defined data structure (class) is used for frequent episodes. The members of this data structure include, the events in the frequent episode, the start time and end time of the frequent episode and the `Pattern Confidence` of the frequent episode. Also, *Collections* like *Vector* and *ArrayList* are used while implementing the algorithm.

4.4 Experimental Analysis

In this section, we shall discuss the experiments performed and the conclusions which can be derived from them. A number of experiments were carried out to compare the performance of Hybrid Apriori and OnePass-FED frequent episode discovery (FED) algorithms. The significant intervals determined for each event form the input to OnePass-FED algorithm.

Data sets collected by the MavHome project are used for conducting the experiments. Synthetic data sets were also created to verify the correctness of the algorithm. These data sets consists of timestamps recorded by all the devices deployed in the MavHome. When a device changes its status, a record is inserted into the database with its associated timestamp.

The algorithm was implemented using Java and the database used was Oracle 9i. The experiments were conducted on a Linux machine (running on dual processors with 2.4 GHz CPU speed and 2 GB memory) of the Distributed and Parallel Computing Cluster at UTA.

Experiments were carried out for daily as well as weekly periodicity for both `Semantics-s` and `Semantics-e`. A summary of the data sets (including the significant

Data set	No. of Days	No. of Tuples	Significant Intervals (Daily)	Significant Intervals (Weekly)	Distinct Event
1	100	27K	1704	5280	2
2	120	40K	2379	6291	3
3	150	54K	2948	8388	4
4	150	67K	3670	10394	5

Figure 4.7 Data sets used for experiments

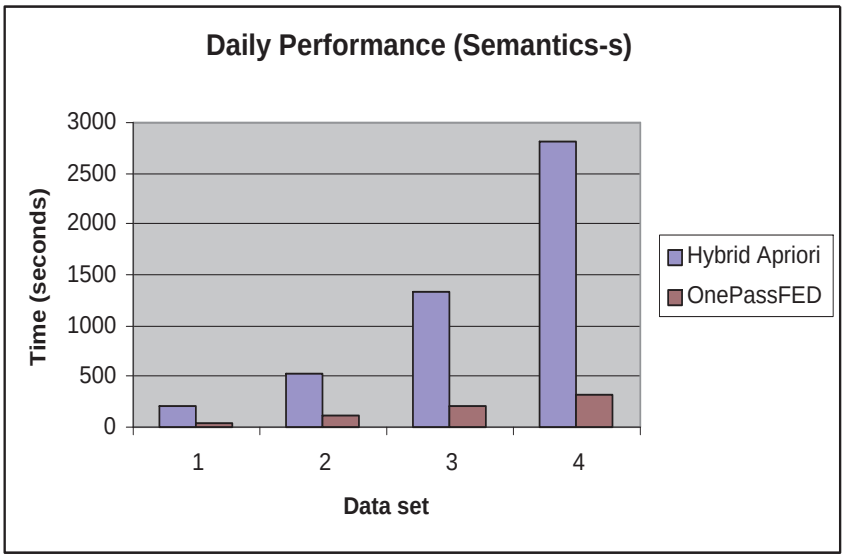

Figure 4.8 Summary of experiments for daily periodicity (Semantics-s)

intervals generated after significant interval discovery) which is used for frequent episode discovery experiments discussed in this thesis is shown in Figure 4.7.

Daily periodicity: The frequent episode discovery was carried out with `Sequential Window`-40.

Figure 4.8 gives a comparison of the time taken by Hybrid Apriori algorithm and OnePass-FED algorithm for `Semantics-s`. As seen from the figure OnePass-FED algorithm shows a performance improvement increasing from 350% to 793%. For further explanation of these results refer to section 4.4.1.

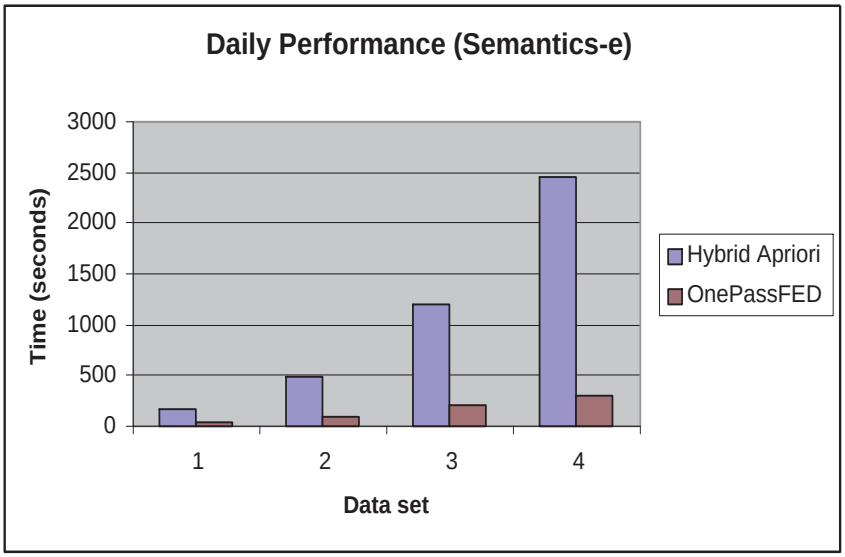

Figure 4.9 Summary of experiments for daily periodicity (Semantics-e)

Figure 4.9 gives a comparison of the time taken by Hybrid Apriori algorithm and OnePass-FED algorithm for `Semantics-e`. As seen from the figure OnePass-FED algorithm shows a performance improvement increasing from 305% to 735%. For further explanation of these results refer to section 4.4.1.

Figure 4.10 gives a comparison of the number of frequent episodes discovered by using `Semantics-s` and `Semantics-e` for the same data set. The figure shows that the number of frequent episodes discovered by `Semantics-s` is greater than the number of episodes discovered by `Semantics-e`. For further explanation of these results refer to section 4.4.1.

Weekly periodicity: The frequent episode discovery was carried out with `Sequential Window`-40.

Figure 4.11 gives a comparison of the time taken by Hybrid Apriori algorithm and OnePass-FED algorithm for `Semantics-s`. As seen from the figure OnePass-FED

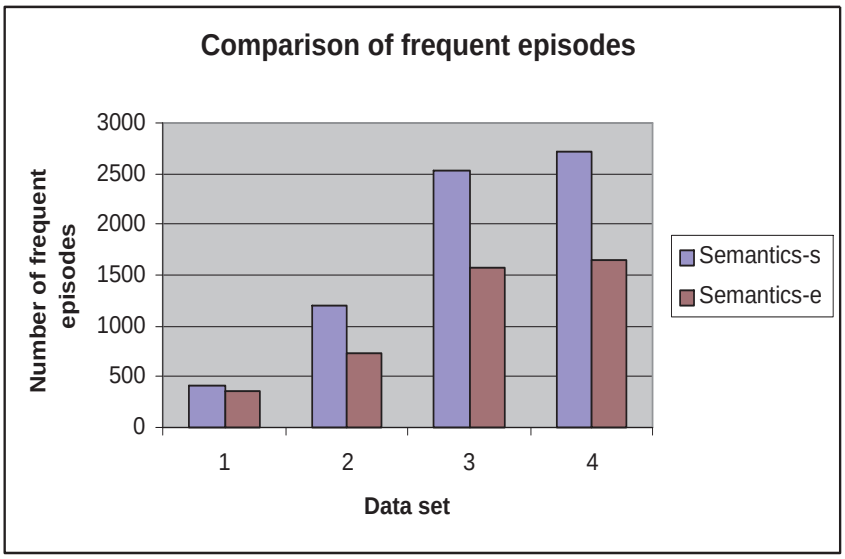

Figure 4.10 Comparison of frequent episodes discovered (Daily Periodicity)

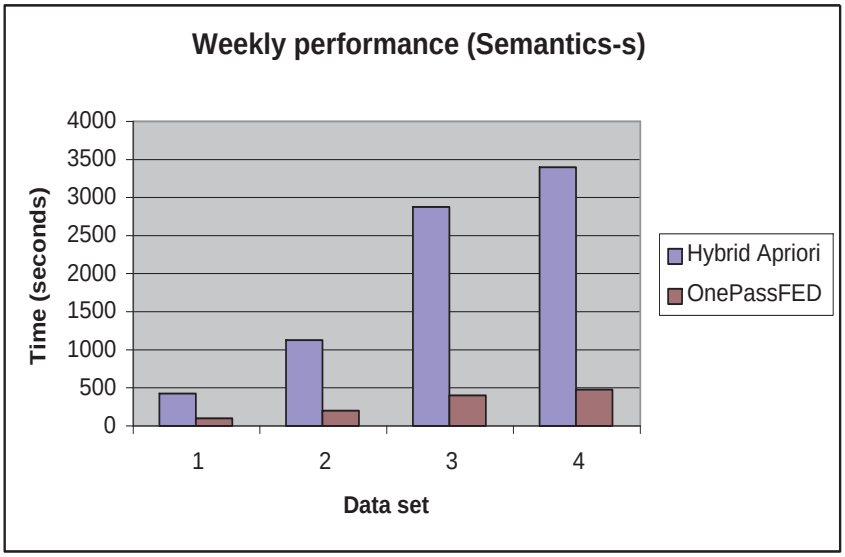

Figure 4.11 Execution Time summary (weekly periodicity and Semantics-s)

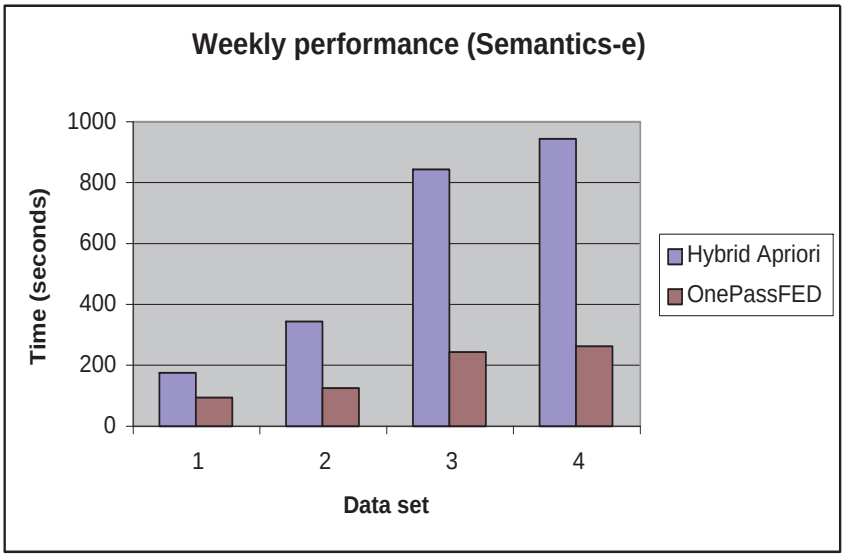

Figure 4.12 Summary of experiments for weekly periodicity (Semantics-e)

algorithm shows a performance improvement increasing from 331% to 608%. For further explanation of these results refer to section 4.4.1.

Figure 4.12 gives a comparison of the time taken by Hybrid Apriori algorithm and OnePass-FED algorithm for `Semantics-e`. As seen from the figure OnePass-FED algorithm shows a performance improvement increasing from 89% to 259%. For further explanation of these results refer to section 4.4.1.

Figure 4.13 gives a comparison of the number of frequent episodes discovered by using `Semantics-s` and `Semantics-e` for the same data set. The figure shows that the number of frequent episodes discovered by `Semantics-s` is greater than the number of episodes discovered by `Semantics-e`. For further explanation of these results refer to section 4.4.1.

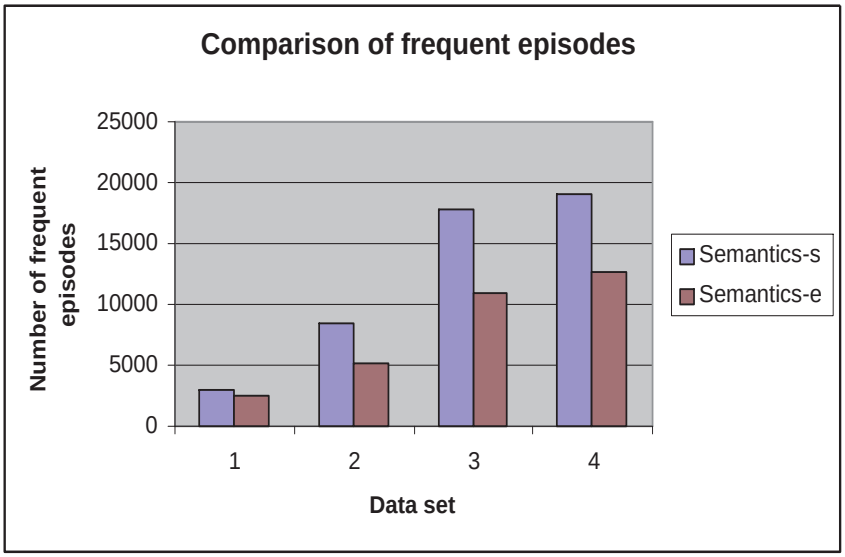

Figure 4.13 Comparison of frequent episodes discovered (Weekly Periodicity)

4.4.1 Conclusions derived from experiments

OnePass-FED algorithm outperforms Hybrid Apriori algorithm: As seen from the experiments, OnePass-FED algorithm significantly outperforms Hybrid Apriori algorithm in terms of time taken for execution. The reasons are: i) Hybrid Apriori is an SQL-based algorithm while OnePass-FED is a memory-based algorithm. Hence, the I/O required for SQL is very time consuming and it hurts the performance of Hybrid Apriori algorithm when compared to OnePass-FED algorithm, ii) duplicate elimination in Hybrid Apriori algorithm takes a lot of time. This is because, database does not allow rearrangement of columns (only rows by using group by and order clauses), we transpose the rows of each frequent episode into columns, sort and reconstruct them to remove the duplicate frequent episodes. In contrast, for OnePass-FED algorithm, we order the events of the frequent episodes in memory before inserting them to the database and then perform a GROUP BY clause to eliminate the duplicates, iii) OnePass-FED algorithm does not generate intermediate intervals. In contrast, Hybrid Apriori follows an apriori

style which generates intermediate intervals and hence requires more time due to the number of passes it has to carry out.

Semantics-s versus Semantics-e: The number of frequent episodes discovered by using `Semantics-s` is greater than those discovered by using `Semantics-e`. The reason is, `Semantics-e` generates frequent episodes which start as well as end within Sequential Window units of the first event in the frequent episode. In contrast, `Semantics-s` generates frequent episodes which start within Sequential Window units of the first event in the frequent episode. Hence, there is a much tighter constraint when using `Semantics-e` which is reflected on the number of frequent episodes discovered as well as the time taken by each of them. This is also the reason that `Semantics-s` takes more time as compared to `Semantics-e`.

4.5 Summary

In this chapter, we discussed OnePass-FED algorithm for frequent episode discovery. The OnePass-FED algorithm proposed in this chapter, takes a different approach than the traditional apriori class of frequent episode discovery algorithms. The proposed OnePass-FED algorithm discovers the frequent episodes in a single pass in a much efficient manner. Experimental results have shown that, OnePass-FED algorithm significantly outperforms Hybrid Apriori algorithm (another frequent episode discovery algorithm) in terms of the time taken to discover the frequent episodes.

CHAPTER 5

VALIDATION OF FREQUENT EPISODES

In the folding phase for significant interval discovery, the periodicity information is lost. Consequently, if we try to interpret the frequent episodes for higher periodicity we may find some false positives in the frequent episodes discovered by using the Hybrid Apriori [15] or OnePass-FED (discussed in the previous Chapter) algorithms. The elimination of false positives is critical to our MavHome problem domain where the episodes represent behavior of the inhabitant and assist the agents focused on providing automation in these environments. If a false episode is automated as a frequent episode, the ultimate objective of a MavHome, which is to maximize comfort of its inhabitants by reducing the manual interaction with the devices, is defeated. This calls for an algorithm that can distinguish the actual frequent episodes from the false positives in the set of frequent episodes identified by Hybrid Apriori or OnePass-FED algorithms.

5.1 Introduction

In this section we shall see answers to the following questions 'What is a false positive?' and 'Why and how does a false positive occur?'

5.1.1 False Positives and Periodicity of Frequent Episodes

Hybrid Apriori/OnePass-FED algorithm discovers episodes for two types for periodicity: daily and weekly. It can also be further generalized to monthly and yearly periodicity. In the daily periodicity, the entire data set is folded over 24-hour period. Weekly periodicity, in contrast, takes into consideration the time component as well as

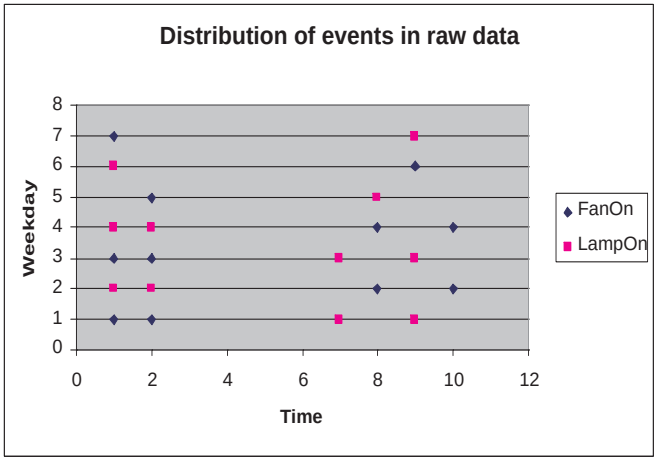

Figure 5.1 Distribution of events in raw data set

the weekday of the event occurrence. Hence, episodes discovered for daily periodicity may have false positives as all the events in an episode may occur at the same time interval but on different weekdays. Similarly, for weekly periodicity, false positives would have events which occur on same weekday and time interval but the week days may be of different months.

5.1.2 False Positives and the Process of Discovery of Episodes

The following example illustrates the process of discovery of episodes for daily periodicity and how false positives (for a higher granularity) may be possible in it.

Consider a small two week data set. This data set has two events, 'FanOn' and 'LampOn', representing a sample scenario where the inhabitant uses the study room. The Figure 5.1 displays the spread of the sample data before folding. The Y-axis corresponds to the weekdays and the X-axis to the time of occurrence of an event.

After the raw data is folded the information about the weekday, month and year is lost. Here the occurrences of the event are grouped by their time example, 'LampOn'

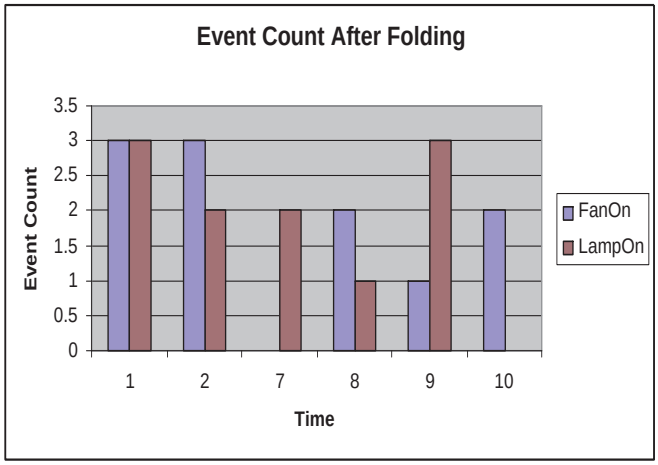

Figure 5.2 Raw data set after folding

event which occurred at time t = 9 units on weekdays 1, 3 and 7 now has an event count of three at time t = 9 units.

The significant interval discovery (SID) algorithms [15, 16, 24] work on the folded data set shown in Figure 5.2 and discover significant intervals based on user specified parameters such as `max-len` and `min-conf`. Significant intervals discovered for each device are seen in the Figure 5.3.

The frequent episode discovery algorithm takes the SID's discovered in the previous step as input and finds the frequent episodes. The number of events in an episode determines the size of the episode. Two frequent episodes of size two are obtained which are displayed in the Figure 5.4.

With the small data set above we can observe that the information for the weekday is lost. But if we can ungroup this information for each episode discovered and compute the support for each weekday from the available raw data set, then we can determine whether an episode is a false positive or a valid episode for higher periodicity.

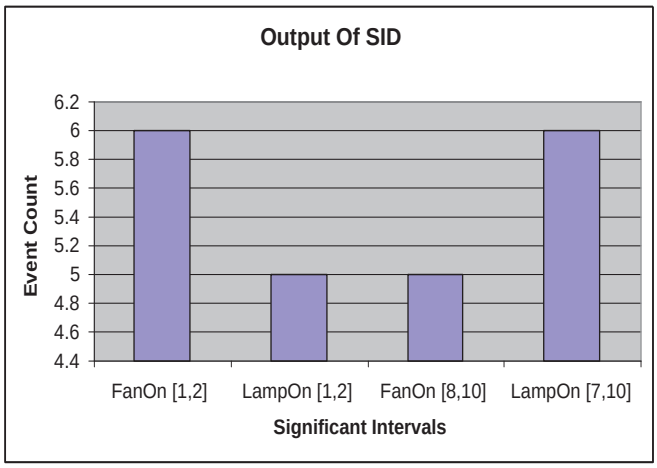

Figure 5.3 Significant intervals discovered by SID

The statistics in the Figure 5.5 show an example of a false positive. The example conveys that all the events participating in the episode of size 2 did occur in the specified time interval but they did not occur together on the same weekday.

As seen from the Figure 5.5, the event 'FanOn' occurred on Monday, Wednesday and Saturday whereas 'LampOn' event occurred on Sunday, Tuesday, Thursday and Saturday. Thus, all the items did not occur together on the same weekday but still were detected as an episode. This happens because the intervals discovered by SID operate on folded data that does not have the information pertaining to the periodicity of the event (that is, the weekday when it occurs).

5.2 Motivation

Bhatia [17] proposed three approaches (Naive, Parallel, Partitioned) to validate the frequent episodes discovered. A detailed study of the design and implementation of these approaches was done to determine if there was a scope for improving the performance and design. After the study it was concluded that, the approaches can be further improved by incorporating some changes to the design. The changes and improvements discussed

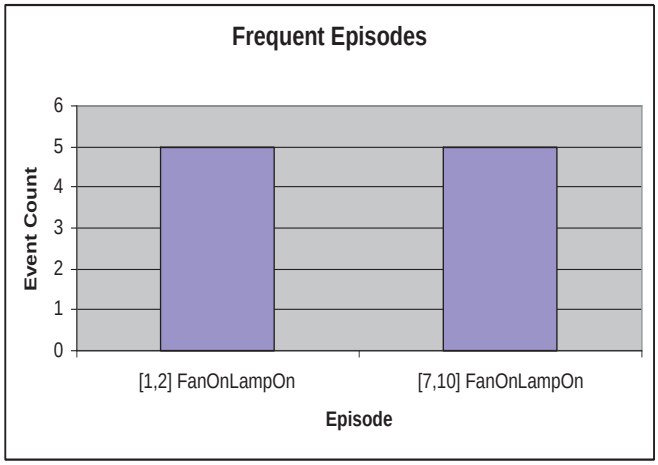

Figure 5.4 Episodes discovered by Hybrid Apriori

here were made to the Naive approach. The changes can also be easily incorporated to the other two (Partitioned and Parallel) approaches. The new algorithm effectively developed is called BatchValidation algorithm.

5.3 BatchValidation Algorithm

This section describes the BatchValidation algorithm, that is, all the phases in the algorithm, the improvements and design changes made with respect to the previous Naive algorithm and other useful details related to BatchValidation algorithm. The BatchValidation algorithm also consists of three phases when compared with its counterpart Naive algorithm. But the design and processing done by each phase is different then before. The three phases carried out by the BatchValidation algorithm are

1. Building phase
2. Support counting phase
3. Pruning phase

We shall now explain each phase in detail.

Episode Start Time		7	
Episode End Time		10	
Event In Episode	FanOn	WeekDay	Support
		Monday	2
		Wednesday	2
		Friday	1
Event In Episode	LampOn	WeekDay	Support
		Sunday	2
		Tuesday	2
		Thursday	1
		Saturday	1

Figure 5.5 Support of Events in an Episode

5.3.1 Building phase

This phase retrieves the episodes discovered by the episode discovery algorithm that are in a database and stores them in a main memory data structure. Representing them in main memory allows us to fetch and update the support count of each event in the episode in the computation phase without incurring additional I/O's. It also allows us to group the episodes by the events in the episode. Grouping the episodes by their events creates an episode list that helps us in fetching the episodes by their events. This grouping is done for each event in the entire set of episodes to be validated. The episode list created by grouping of episodes is unique to each event and helps in identifying the episodes in which a particular event occurs.

Design changes when compared with respect to Naive approach:

A detailed study of the building phase resulted in a few ideas which would help improve the performance of the building phase. The changes made to the original design are explained below:

1. **Building of the two hash tables:** The building phase builds two hash tables, that is, Episode hash table and Event hash table. The Episode hash table stores

the details of the frequent episodes from the database and the Event hash table stores details of the distinct events in these episodes. The Naive approach [17] builds these hash tables sequentially (one after another), that is, first the Episode hash table is created and from it the Event hash table is created. This resulted in iterating through the entire Episode hash table to create the Event hash table. Even though this design works fine, it is not an efficient one. The Event hash table can be created while the Episode hash table is been created. This will save a considerable amount of processing time.

2. **Keeping minimum start time and maximum end time for each distinct event:** In the support counting phase phase, the entire raw data set is read from the database. If we keep track of the minimum start time and maximum end time for each distinct event, we can ignore some of the data tuples from the raw data set for a whole set of episodes which contain that particular event. This not only reduces the I/O's but also, reduces the computation required in the support counting phase.

Pseudo Code for building phase:

The pseudo code for the building phase is seen in Algorithm 4. The building phase as we had explained gets all the frequent episodes from the database and builds two hash tables to represent the frequent episode and use them for further processing.

Explanation of Pseudo Code for building phase:

The building phase starts by bringing all the frequent episodes discovered into main memory. Then for each frequent episode, a series of steps are carried out. First (in Line 2), the type of the frequent episode is determined, that is, if it a normal or wrapping (spanning two days, week, month, etc.) frequent episode. The frequent episode under consideration is then stored in the Event hash table (in Line 3). Then, for all the events which constitute the current frequent episode a series of steps are carried out. First (in

Algorithm 4 Building Phase

```
1:  for all Frequent episodes discovered by the episode discovery algorithm do
2:      Get frequent episode and determine its type (that is, normal or wrapping)
3:      Store the frequent episode in Episode hash table.
4:      for all Event present in the current frequent episode do
5:          if Event present in Event hash table then
6:              Add the episode id of this frequent episode to the episode list for that event.
7:              Update the minimum start time and maximum end time for the event.
8:          else
9:              Create an episode list for this event.
10:             Add the episode id of this frequent episode to the newly created episode list for that event.
11:             Update the minimum start time and maximum end time for the event.
12:         end if
13:     end for

14: end for
```

Line 5), the existence of the event in the Event hash table is checked. If the event is already present in the Event hash table then Lines 6 and 7 are executed. Lines 9 through 11 are executed if the event is not present in the Event hash table. In Line 6, the episode id of the current frequent episode is added to the list of episode ids for that event. This list of episode ids gives all the frequent episodes in which this event occurs. Line 7 checks the minimum start time and maximum end time of the event with the start time and end time of the frequent episode and retains the minimum of the start times and maximum of the end times. If the event is not present in the Event hash table then we create a new entry corresponding to this event in the Event hash table. An episode id list is created for this event in Line 9. In Line 10, the episode id of the current frequent episode is added to this newly created episode id list. Additionally, the start time and end time of the frequent episode is stored as the minimum start time and maximum end time for this event (Line 11).

Output:

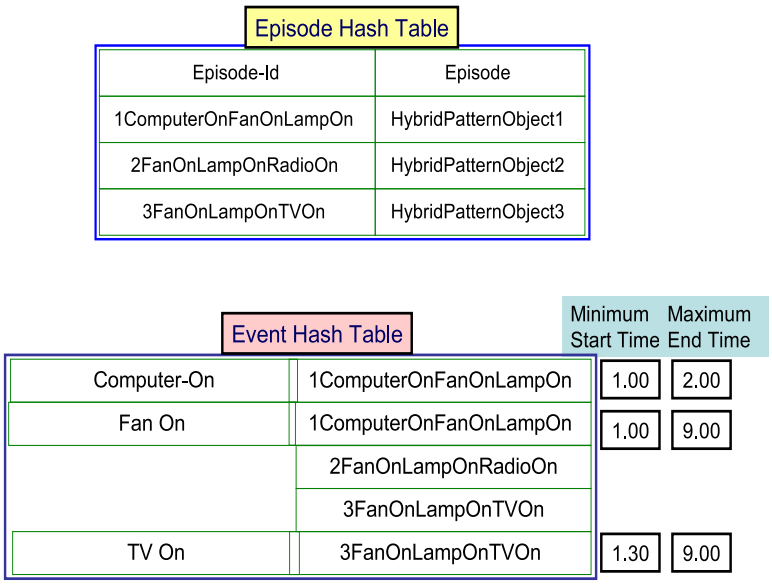

Figure 5.6 Sample output of building phase

A sample output of the hash tables created after the building phase is as seen in Figure 5.6

5.3.2 Support Counting Phase

The support counting phase makes a single pass over the raw data set and computes the support for each event in a frequent episode for a specified granularity. For an event its episode list is obtained. For each episode in this list, we check if the transaction time of the event falls in the range of the episode interval. At the end of the support counting phase we have the support statistics for each event in the episode ungrouped based on the periodicity of the episode.

Design changes when compared with respect to Naive Approach:

The changes in the support counting phase for BatchValidation algorithm are explained as below:

Reading the raw data: The Naive approach reads the raw data ordered by time. Each tuple represents an event. After identifying the event, the frequent episodes corresponding to this event are obtained from the Event hash table. The individual frequent episode is obtained from the Episode hash table. The time at which this event has occurred is compared with each frequent episode. If the time is within the start time and end time for the frequent episode then the support for that frequent episode is incremented. This comparison is done for each frequent episode corresponding to a particular event. This whole series of comparison is done for all the tuples in the raw data set. On analyzing this method of reading and comparing the raw data set the following points are identified as the time consuming operations.

1. for every event (tuple) read from the raw data set both the hash tables (Episode hash table and Event hash table) have to be looked up,
2. a tuple which will not satisfy any of the frequent episodes has to still be read and compared with each frequent episode the event corresponds to

To overcome the above mentioned time consuming factors the following design changes are made to the support counting phase.

1. The raw data is read ordered on event. This ordering enables us to reduce the number of lookups to the hash tables. For a particular event, we have to lookup the Event hash table only once. This drastically reduces the number of hash table lookups when compared to Naive approach.
2. The minimum start time and maximum end time which we had defined and stored in the building phase is used in this phase. When reading raw data corresponding to an event we use this minimum start time and maximum end time. While reading raw data for a particular event the tuples which lie in this range are read and the other tuples are not read. The reasoning behind this is that, the discarded tuples will not satisfy any of the frequent episodes time range. Hence, we can reduce

the number of lookups to the Episode hash table by not reading these tuples at all. Not only the hash table lookups are reduced, but also comparing each of the discarded tuple with each frequent episode is eliminated. The number of I/O's are also reduced.

Pseudo Code for support counting phase:

The pseudo code for the support counting phase is seen in Algorithm 5. The support counting phase gets all the tuples (raw data set) from the database and finds the raw support for all the frequent episodes found in the Episode hash table.

Algorithm 5 Support Counting Phase

```
1:  for all Distinct Events do
2:      Get the frequent episode id list corresponding to the event from the Event hash table.
3:      Get tuples from raw data set lying in the minimum start time and maximum end time range for the corresponding
        event
4:      for all Tuples obtained from raw data set do
5:          for all frequent episodes corresponding to this event do
6:              if Tuple time lies in between the start time and end time for this frequent episode then
7:                  Increment the raw support for the frequent episode.
8:              end if
9:          end for
10:     end for
11: end for
```

Explanation of pseudo code for support counting phase:

For the support counting phase the raw data set is read ordered by event. Hence, for each distinct event a series of steps is carried out. First (Line 2), we get the frequent episode id list (corresponding to the event under consideration) from the Event hash table. Then (Line 3), we get the tuples (corresponding to the event under consideration) which lie within the minimum start time and maximum end time from the raw data set. After getting all the tuples in main memory, the support for each frequent episode

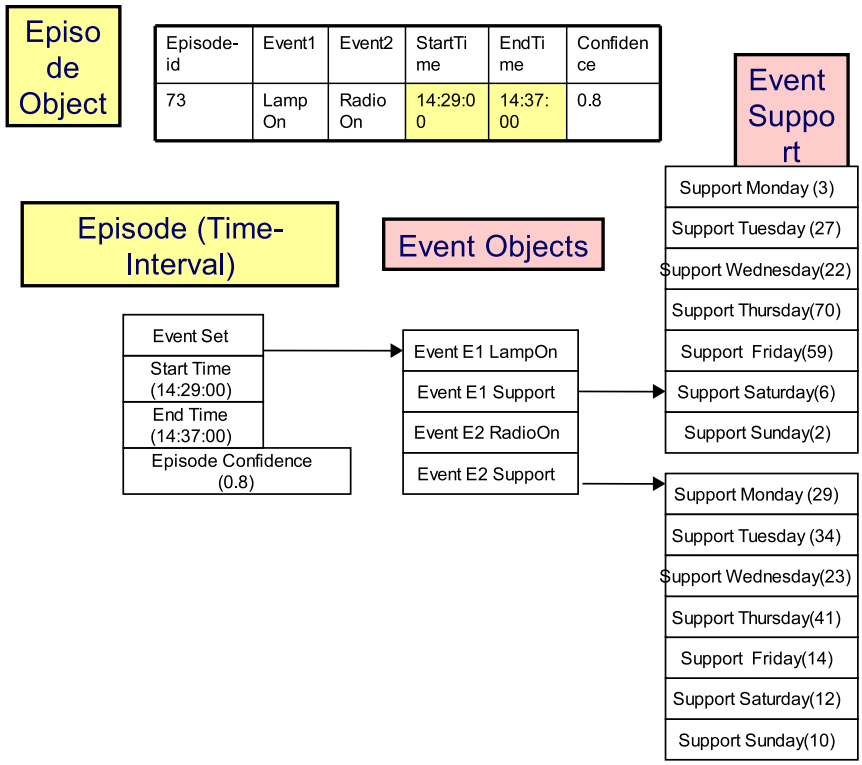

Figure 5.7 Sample output of Support counting phase

corresponding to the event is checked in Lines 5 through 8. Line 6 is where we check if the time of the current tuple lies in the start and end time range of the frequent episode. If it does lie within the time range then the support for that frequent episode is incremented.

Output:

A sample output of the support for a frequent episode is as seen in Figure 5.7

5.3.3 Pruning Phase

The pruning phase checks the support count for each event in an episode. If the support count of each event in the episode meets the minimum support threshold values for at least one day/week then the episode is a valid episode otherwise it is a false positive.

Data Set	Frequent Episodes	Raw Data Set	No. of Distinct Events
1	530	54K	4
2	1213	67K	5
3	2480	81K	6
4	4896	94K	7
5	9519	108K	8
6	18424	121K	9

Figure 5.8 Summary of data sets for experiments

This phase is same as that proposed by Bhatia in [17]. There are no design modifications made in this phase. It is the same irrespective of the approach used, that is, Naive, BatchValidation, Parallel and Partitioned. We shall only give the pseudo code for this phase.

Pseudo Code for pruning phase: The pseudo code for pruning phase in seen in Algorithm 6.

5.4 Experimental Results

The experiments performed for this phase were performed to compare the execution time of the BatchValidation algorithm to the Naive algorithm. The algorithm was implemented using Java and the database used was Oracle 9i. The experiments were conducted on a Linux machine (running on dual processors with 2.4 GHz CPU speed and 2 GB memory) of the Distributed and Parallel Computing Cluster at UTA.

A summary of the data sets which were used for performing these experiments is seen in figure 5.8. These data sets were obtained after performing significant interval discovery and frequent episode discovery on the raw data sets.

Algorithm 6 Pruning Phase

```
1:  for all Episodes in Episode Hash Table do
2:      Set EpisodeValid flag to false
3:      if Episode is normal episode then
4:          for all Weekday/Month (depending on granularity) do
5:              for all Event occurring in the Episode at hand do
6:                  if Raw support on that weekday/month >= Support threshold value then
7:                      Set EventValid flag to true.
8:                  else
9:                      Set EventValid flag to false.
10:                     Break (no need to check for other events for this weekday/month)
11:                 end if
12:             end for
13:             if EventValid then
14:                 Set EpisodeValid flag to true.
15:             end if
16:         end for
17:     else
18:         for all Weekday/Month (depending on granularity) do
19:             for all Event occurring in the Episode at hand do
20:                 if Raw support on that weekday/month + Raw support on next weekday/month >= Support threshold
                    value then
21:                     Set EventValid flag to true.
22:                 else
23:                     Set EventValid flag to false.
24:                     Break (no need to check for other events for this weekday/month)
25:                 end if
26:             end for
27:             if EventValid then
28:                 Set EpisodeValid flag to true.
29:             end if
30:         end for
31:     end if

32: end for
```

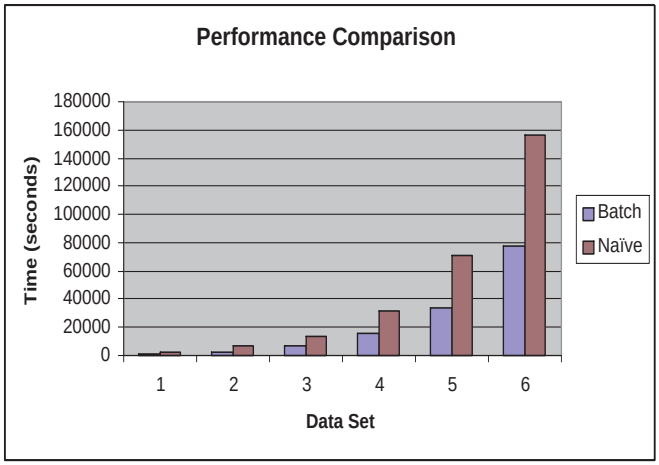

Figure 5.9 Summary of Experiments

The bar chart in figure 5.9 shows the time comparison required for execution of BatchValidation algorithm and Naive algorithm. As observed from the chart, the design modifications made in the BatchValidation algorithm helps reduce the amount of time taken for frequent episode validation. There is a performance improvement in between 100% to 150% as observed from the chart. The following reasons contribute to an improvement in the performance: i) the hash tables (Event hash table and Episode hash table) are created using a much better design, ii) the number of lookups to the hash tables are reduced by bringing the raw data set in memory by using ordering on events, and iii) the number of raw tuples which are read are reduced by using the minimum start time and maximum end time (reducing the I/O's by the database).

5.5 Conclusion

In this chapter, we discussed frequent episode validation for higher periodicity. We discussed the design issues with our previously developed Naive algorithm. We redesigned the phases carried out by frequent episode validation and proposed a new algorithm (BatchValidation). We implemented this algorithm and compared its performance with

the Naive algorithm. The experiments proved that BatchValidation algorithm performed much better when compared to the Naive algorithm in terms of time taken for execution.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In many application domains, intervals provide more information on the characteristics of the data as compared to time points. In real-world applications (example, predicting automating devices in a smart home), we are interested in identifying intervals with a high degree of certainty in which event happen, instead of time points. In these applications, it is important to predict tight and accurate events to effectively automate the application. Although, there is a considerable body of work on sequential mining of transactional data, most of them deal with time point data and make several iterations over the entire data set for discovering frequently occurring patterns.

In this thesis, we proposed an alternative approach which consisted of three phases. In the first phase, time-series data was folded over a periodicity (day, week, etc.) and significant intervals for each event (or device) were computed (independently). In the second phase, a frequent episode discovery algorithm was used on these significant intervals for many events (associated with devices) to discover interesting episodes/patterns among the events. The third and final phase was frequent episode validation. This phase was used to validate the frequent episodes if they were to be interpreted on a larger granularity. For example, if the significant interval discovery had been done using daily periodicity and if the prediction needed to take into account days of the week (that is, weekly periodicity), then validation for weekly periodicity was needed to be done.

This thesis proposed efficient algorithms for each of the three phases. The algorithms proposed for significant interval discovery and frequent episode discovery were both single pass main memory algorithms. Since they were main memory algorithms,

scalability issues had been considered while designing each of them. The frequent episode validation phase was redesigned to improve its performance. Finally, we carried out extensive experiments for each phase. These experiments compared the performance of the newly developed algorithms with the previous algorithms for the respective phases.

Currently, we are working on applying the algorithms presented in this thesis to time-series data from other domains (example, traffic analysis, internet usage, and others). Incremental versions of these algorithms will be important as data is collected continuously and new data may change the significant intervals and frequent episodes. Hence, the ability to incrementally compute significant intervals and frequent episodes is important and needs to be investigated. Additionally, improving the performance of the current algorithms is a challenging task and needs to be looked into.

REFERENCES

[1] J. Han and M. Kamber, *Data Mining:Concepts and Techniques.* Morgan Kaufmann Publishers, 2005.

[2] W. Frawley, G. Piatetsky-Shapiro, and C. Matheus, "Knowledge discovery in databases: An overview," *AI Magazine, Fall 1992*, pp. 213–228.

[3] D. Hand, H. Mannila, and P. Smyth, *Principles of Data Mining*, 2nd ed. Cambridge, MA: MIT Press, 2001.

[4] T. Mitchell, *Machine Learning.* McGraw Hill, 1997.

[5] R. T. Ng and J. Han, "Efficient and effective clustering methods for spatial data mining." in *VLDB*, 1994, pp. 144–155.

[6] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases." in *SIGMOD Conference*, 1994, pp. 419–429.

[7] R. Srikant and R. Agrawal, "Mining sequential patterns," in *ICDE*, Taipei, Taiwan, 1995, pp. 3–14.

[8] B. Özden, S. Ramaswamy, and A. Silberschatz, "Cyclic association rules." in *ICDE*, 1998, pp. 412–421.

[9] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan, "Web usage mining: discovery and applications of usage patterns from web data," *SIGKDD Explor. Newsl.*, vol. 1, no. 2, pp. 12–23, 2000.

[10] J. F. Roddick and M. Spiliopoulou, "A survey of temporal knowledge discovery paradigms and methods." *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 4, pp. 750–767, Jul/Aug. 2002.

[11] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovering frequent episodes in sequences." in *KDD*, 1995, pp. 210–215.

[12] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements." in *EDBT*, 1996, pp. 3–17.

[13] D. J. Cook, G. M. Youngblood, *et al.*, "Mavhome: An agent-based smart home." in *PerCom*, 2003, pp. 521–524.

[14] M. H. Böhlen, R. Busatto, and C. S. Jensen, "Point-versus interval-based temporal data models." in *ICDE*, 1998, pp. 192–200.

[15] A. Srinivasan, "Significant interval and episode discovery in time-series data," Master's thesis, The University of Texas at Arlington, 2004. [Online]. Available: http://www.cse.uta.edu/research/publications/Downloads/CSE-2003-39.pdf

[16] S. Shrestha, "SQL-Based Approach to Significant Interval Discovery in Time-series Data," Master's thesis, The University of Texas at Arlington, 2005. [Online]. Available: http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/Shr05MS.pdf

[17] D. Bhatia, "Aproaches for Validating Frequent Episodes Based on Periodicity in Time-series Data," Master's thesis, The University of Texas at Arlington, 2005. [Online]. Available: http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/Bha05MS.pdf

[18] E. O. H. III, "Using information-theoretic principles to discover interesting episodes in a time-ordered input sequence," PhD Dissertation, The University of Texas at Arlington, Department of Computer Science and Engineering, 2004.

[19] M. J. Zaki, "Spade: An efficient algorithm for mining frequent sequences." *Machine Learning*, vol. 42, no. 1/2, pp. 31–60, 2001.

[20] M. Zaki, "Sequence mining in categorical domains: Incorporating constraints," in *CIKM*, 2000, pp. 422 – 429.

[21] E. O. H. III and D. J. Cook, "Improving home automation by discovering regularly occurring device usage patterns." in *ICDM*, 2003, pp. 537–540.

[22] J. Rissanen, *Stochastic Complexity in Statistical Inquiry.* World Scientific Series in Computer Science, 1989.

[23] A. Srinivasan, S. Shrestha, and S. Chakravarthy, "Discovery of significant intervals in sequential data." in *Proceedings, ADBIS Workshop on Data Mining and Knowledge Discovery*, Sept. 2005, pp. 87–98.

[24] S. Savla and S. Chakravarthy, "A single pass algorithm for discovering significant intervals in time-series data," in *Proc. of 2nd ADBIS Workshop on Data Mining and Knowledge Discovery (ADMKD'06)*, Thessaloniki, Greece, 2006, pp. 49–60.

BIOGRAPHICAL STATEMENT

Sagar Savla was born in Mumbai, India, in 1983. He obtained a B.E. degree in Information Technology from the University of Mumbai in 2004. His interest in Database Systems brought him to the University of Texas at Arlington, where he obtained his M.S. degree in Computer Science and Engineering in 2006. His research interests include database systems, data mining mainly time series data mining. He is a member of the Tau Beta Pi National Engineering Honor Society and Upsilon Pi Epsilon Engineering Honor Society.