APROACHES FOR VALIDATING FREQUENT EPISODES BASED ON PERIODICITY IN TIME-SERIES DATA

by

DHAWAL Y BHATIA

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 2005

ACKNOWLEDGEMENTS

Firstly, I would like to express my deepest sincere gratitude to my advisor, Sharma Chakravarthy, for his magnanimous patience, guidance and support through the course of this research work. I would also like to thank Mohan Kumar and David Levine for serving on my thesis committee and would like to acknowledge the support, in part, by NSF grants (ITR 0121297, IIS-0326505, and EIA-0216500) for this research.

A special thanks to Raman, who spared his valuable time in discussing this research and for maintaining a well-administered research environment. This research would have been incomplete without the support extended by my fellow ITLABians: Akshaya, Sunit, Ajay, Vamshi, Shravan, Vihang, Srihari, Nikhil, Vishesh, Hari, Laali and Manu for maintaining high standards of professionalism and for making ITLAB the perfect place to work in, filled with fun. A special thanks to Akshaya for being by my side and helping me relieve stress levels during the entire tenure of graduation.

I would also like to thank Shilpa and Ankita, who were my colleagues at the Indian Institute of Management, Ahmedabad (IIM-A), for a thorough review of this thesis to improve its overall quality and readability.

My sincere thanks to my Uncle and Aunt, Ugersain and Usha Chopra, who motivated and guided me in building the best strategy to achieve my key goals and heartfelt aspirations.

Last, but certainly not the least, thanks to my family: my parents, Yogendra and Vimla, my elder brother Jayesh, my sister-in-law Komal and my nieces Simran and Pooja; your love and confidence has made this possible and added more meaning to this research and the degree.

November 4, 2005

ABSTRACT

APPROACHES FOR VALIDATING FREQUENT EPISODES BASED ON PERIODICITY IN TIME-SERIES DATA

Publication No. ______

Dhawal Y Bhatia, M.S.

The University of Texas at Arlington, 2005

Supervising Professor: Sharma Chakravarthy

There is ongoing research on sequence mining of time-series data. We study Hybrid Apriori, an interval-based approach to episode discovery that deals with different periodicities in time-series data. Our study identifies the anomaly in the Hybrid Apriori by confirming the false positives in the frequent episodes discovered. The anomaly is due to the folding phase of the algorithm, which combines periods in order to compress data.

We propose a main memory based solution to distinguish the false positives from the true frequent episodes. Our algorithm to validate the frequent episodes has several alternatives such as the naïve approach, the partitioned approach and the parallel approach in order to minimize the overhead of validation in the entire episode discovery process and is also generalized for different periodicities. We discuss the advantages and disadvantages of each approach and do extensive experiments to demonstrate the performance and scalability of each approach.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The proliferation of computers in our daily activities has created abundant generated data. Collection and analysis of this data is critical for decision-making in our lives. Thus, information systems that support decision making in order to automate several aspects of life have become a necessity. Database management systems developed for such information systems store, manipulate and enable retrieval of data. A multitude of database applications are designed and have resulted in the emergence of the field known as Data mining. This field has attracted academicians and the industry due to the abundance of data and the imminent need for turning it into useful information and knowledge. Data mining involves an integration of techniques from multiple disciplines such as database technology, statistics, machine learning, high-performance computing, pattern recognition, neural networks, data visualizations, information retrieval, image and signal processing, and spatial data analysis. Data mining systems are categorized based on the underlying techniques employed such as classification, clustering, prediction, deviation analysis, association analysis and sequential mining.

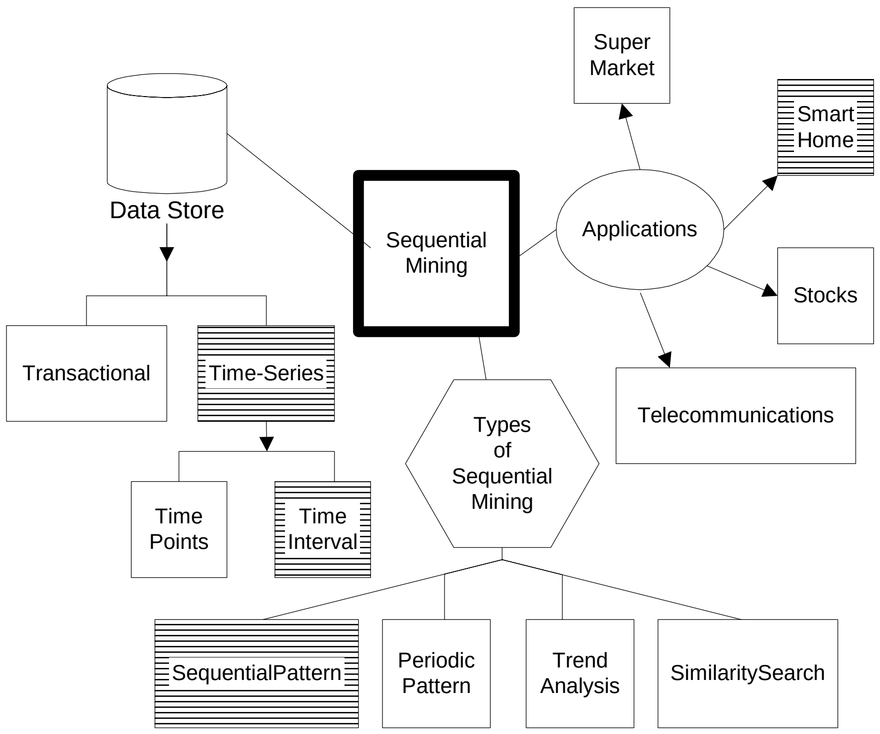


Figure 1 Sequential Mining: An overview

1.1 Sequential pattern mining

Sequential pattern mining entails the identification of frequently occurring patterns related to time or other sequences. An example of sequential pattern is *"A customer, who bought Fellowship of the Rings DVD six months ago, is likely to buy the Two Towers DVD within a month"*. Since many business transactions, telecommunication records, weather data and production processes fall into the category of time sequence data, sequential mining is useful for target marketing, customer retention and so on. The emphasis in our research is on accurate and scalable data mining techniques for sequential mining in large database.

1.1.1 Sequential mining for transactional data

Sequential pattern mining was introduced in [2] and it can be conducted on transactional data or time-series data. Transactional data stored in a database consists of transactions; each transaction is treated as a unique record. If we consider the example of a supermarket, the information stored in a record would be the customer-id, transaction time and the items purchased. The objective here is to identify sets of items that are frequently sold or purchased together. A market basket data analysis of this kind enables the vendor to bundle groups of items to maximize sales. For time-series data, a database record will consists of sequences of values or events changing with time. [3]. These values are typically measured at equal time intervals. Mining transactional data sets will typically look for association between data items and will discover a rule of type {Beer} implies {Chips}. In contrast, mining a time-series data set will provide more insight in to the same rule by discovering that the rule {Beer} implies {Chips} has a larger support during 8 pm to 10 pm every Friday. Research in time-series data mining covers issues related to trend analysis, similarity search in time series data, prediction of natural disasters and mining sequential patterns and periodic patterns in time-related data. Time-series analysis can also be used for studying daily fluctuations of a stock market, scientific experiments, and medical treatments.

1.1.2 Sequential mining for time-series data

This type of data can be represented as follows: when A occurs, B also occurs within time t_i from the time of occurrence of A. In general three attributes characterize sequence data: object, timestamp, and event. Hence, the corresponding input records consist of occurrences of events on an object at a particular time. The major task associated with this kind of data is to identify existing sequential relationships or patterns in the data. Appropriate techniques are applied to discover the trends or the patterns in the data with respect to multiple granularities of time (i.e., different levels of abstraction). These trends or patterns may be further used for prediction or decision making. The patterns discovered are based on measures of interestingness such as support and confidence. Support of an event is defined as the number of occurrences of the event. Confidence of a pattern is the probability of its events occurring together. The threshold values for these measures are domain specific and are controlled by the user. Two algorithms have been proposed in [4] to discover frequent episodes from a given set of sequences. The algorithms define a frequent episode as a collection of events that occur within the given time interval (window) in a given partial order.

1.1.3 Sequential mining for interval based time-series data

Sequential mining algorithms for time-series data can run on point-based data or on interval-based data that represents intervals of high activity. Intervals represent groups of time or activity that best represents the data with certain characteristics. The characteristics of an interval can be its density, length or strength. Every interval has a start time and an end time. The difference between the two timings is the length of the interval (l). Strength of the interval is the sum of the strength of the points that form the interval (s) while density (d) of an interval relates its total strength(s) with its length (l). Several approaches to represent time points as intervals are discussed in [5] where the focus is on mining of sequential patterns for interval based time-series data. Multiple sequential mining algorithms [2, 4, 6-9] for time-series data exist in the literature. However, these algorithms operate on point data for mining frequent episodes/patterns. The advantage of interval-based sequential mining algorithm over traditional sequence mining approaches is that interval-based sequential mining algorithm operates on compressed data for sequence discovery.

1.2 Problem Domain

One of the applications of a sequential mining is a smart home and the problem domain for this thesis is MavHome [10]. This smart home project is a multi-disciplinary research project at the University of Texas at Arlington (UTA) that focuses on the creation of an intelligent and versatile home environment. The goal here is to create a home that acts as a rational agent, perceiving the state of the home through sensors and acting upon the environment through effectors. The agent acts in a way to maximize its goal; that is, it maximizes comfort and productivity of its inhabitants, minimizes cost, and ensures security.

To accomplish the goals of a smart home, the time intervals during which the inhabitant interacts with specific set of devices needs to be identified. Once this is done, the operations of the devices can be automated to eliminate the need for manual interaction between the inhabitant and the devices. Examples of patterns of interest in MavHome are:

“Every morning Bill turns on the exercise bike and the fan between 7 am and 7:15 am”

"Every evening between 8 pm and 8:30 pm, Cindy turns on the drawing room light and the television to watch 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 see that the frequent episodes of interest relate to a group of devices with which a smart home inhabitant interacts, which occur during the same time interval with sufficient periodicity.

1.3 Hybrid-Apriori

This is an interval based episode discovery algorithm, proposed in [11], which discovers such episodes. Instead of performing computations on large raw data, Hybrid-Apriori algorithm works on compressed data that has intervals instead of points. This reduces the amount of time spent per pass significantly; the number of passes, however, remains the same. Generation of frequent episodes is done in three phases:

1. Folding Phase
2. Significant Interval Discovery Phase (SID)
3. Frequent Episodes Discovery Phase (Hybrid Apriori)

The first phase compresses the time points by folding the data over the periodicity provided by the user (e.g., daily, weekly). The second phase represents the folded data as intervals and discovers the intervals [5], termed as significant intervals, that have the user specified support and interval length. In the third phase, Hybrid-Apriori algorithm takes these significant intervals as input and identifies the frequent episodes that satisfy user specified confidence.

1.3.1.1 Anomalies in Hybrid-Apriori

In the folding phase of hybrid-apriori approach, the periodicity information is lost. Consequently, we may find some false positives in the output of this algorithm. The elimination of false positives is critical to our problem domain where the episodes represent behavior of the inhabitant and assist the agents focused on providing automation in these environments. For instance, consider the scenario of the laundry room mentioned earlier. Here, Judy uses the laundry only on Tuesdays and Saturdays between 2 p.m. to 3 p.m. Due to the folding of data, information related to the time granularity at the next level, i.e., weekday information for daily periodicity, is lost. A frequent episode {LRMachOn, LRLightsOn, 2 p.m., 3p.m, 0.8} representing the laundry scenario is identified as a daily episode where 'LRMachOn' and 'LRLightsOn' represent the laundry machine and the lights respectively. The episode starts at 2 p.m. and ends at 3 p.m. and 0.8 is the confidence of the episode. But in reality, the episode occurs only on Tuesdays and Saturdays. If this episode is automated as a daily episode, the ultimate objective of a Smart Home, 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 daily episodes from the false positives in the set of frequent episodes identified by Hybrid Apriori.

1.4 Proposed Solution

We propose a main memory algorithm that makes a single pass over the raw dataset and the frequent episodes generated by the Hybrid-Apriori algorithm to eliminate the false positives present in the frequent episodes. Multiple approaches to validate the frequent episodes have been developed in this thesis. These approaches address the issues of performance and scalability and ensure that the overhead of validating the episodes for an interval based episode discovery algorithm is minimal. Thus, the entire Hybrid-Apriori algorithm to discover the true frequent episodes now consists of four phases:

1. Folding Phase
2. Significant Interval Discovery Phase (SID)
3. Frequent Episodes Discovery Phase (Hybrid Apriori)
4. Pruning of false positives (Validation)

Our algorithm to validate the frequent episodes has alternatives such as the Naïve approach, the Partitioned approach and the Parallel approach. We discuss the advantages of each approach. Through extensive experiments and analysis, we attempt to demonstrate the performance and scalability of these alternatives.

1.5 Other Contribution

We have also compared the interval-based Hybrid-Apriori algorithm with a point based main memory algorithm termed ED for episode discovery [1]. This comparison has been done with the objective of demonstrating that Hybrid Apriori, in spite of the need for validation, would be a better alternative as compared with traditional episode discovery algorithms with respect to performance and scalability. Additionally, in the process of finding frequent episodes, Hybrid-Apriori generates significant intervals and clusters the ones that are useful in their own right for inferring individual activities in a smart home environment.

CHAPTER 2

RELATED WORK

2.1 Introduction

Traditional algorithms [1, 2, 4, 7, 8] to discover frequent episodes operate on time stamped data. To the best of our knowledge, Hybrid-Apriori [11] has been the only interval-based sequential mining algorithm that discovers frequent episodes from time-series data. This algorithm takes significant time-intervals as an input to discover episodes of different periodicity. We provide a survey of approaches found in the literature in the following sections. We also highlight significant differences between the traditional approach to episode discovery and the Hybrid-Apriori approach for discovering episodes from significant intervals. We then discuss the anomaly in the interval-based episode discovery and provide a brief overview of our proposed solution.

2.2 GSP

The GSP (Generalized Sequential Patterns) [2] 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. 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 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:

- *c* is derived from *s* by dropping an event from its first or last event-set.
- *c* is derived from s by dropping an event from any of its event-sets that have at least two elements.
- *c* is a contiguous subsequence of *c'*, that is a contiguous subsequence of *s*.

This algorithm consists of two phases: the first phase scans the database to identify all the frequent items of size one. 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.

2.3 WINEPI and MINEPI

The authors in this paper [4] 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. WINEPI is an algorithm, 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 episode 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 tow 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.

2.4 ED

The algorithm Episode Discovery (ED) proposed in [1] 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 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. The algorithm has five different phases.

First, it partitions the input sequence based on the input parameters such as the window time span and other capacity parameters. Second it generates candidates using the set intersection and difference operations. Third, pruning is done based on the MDL-based evaluation measure - compression ratio achieved. The apriori property to prune is not sufficient in this approach as episodes with several characteristics needs to be discovered. Fourth, the candidate evaluation phase where the generated candidates are evaluated using the compression ratio and the periodicity and regularity of the patterns is discovered using the autocorrelation techniques. Finally, the episodes with greatest compression ratio are selected as interesting episodes and candidates that overlap with the interesting episodes are pruned.

2.5 Hybrid-Apriori

Hybrid-Apriori [11] is an SQL-based sequential mining algorithm that takes the significant intervals as input from Significant Interval Discovery (SID) algorithm and discovers frequent sequences to automate the devices in a smart home. It uses CDIST_O (distinct occurrences with possibility of event timestamp overlap) as sequence counting method. This method considers the maximum number of all possible distinct occurrences of a sequence over all objects; that is, the number of all distinct timestamps present in the data for each object. The novelty of the approach lies in using interval-based data as input. The interval-based data is a reduced data set consisting of significant intervals of events in the raw data discovered by the SID suit of algorithms [5].

2.5.1 Hybrid-Apriori and Traditional mining algorithm

1. The primary difference is the use of time-intervals instead of time points. As an ordering criterion, during a tie between sequences having the same interval boundaries, the interval with the maximum interval-confidence is chosen above the others. Similarly, among sequences with the same start point and interval-confidence, the sequence with the earliest end point is chosen. Thus, greater importance is placed on sequences with higher interval-confidence and smaller lengths, thereby extracting the tightest sequential pattern.
2. Hybrid-Apriori algorithm eliminates some of the steps used by the traditional apriori approach. Application of SID algorithm results in partitioning and extraction of intervals with sufficient interval-confidence from the dataset. Therefore most of the points, which would have been eliminated in the support counting phase of the traditional approach, have been eliminated before the start of sequential mining.
3. Pattern-confidence (PC) replaces support counting in the hybrid-apriori algorithm that represents the minimum number of occurrence of the sequence within the interval. The pattern-confidence of a sequence within an interval is the minimum of the interval-confidence (IC) of its events. 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. Instead of using m-copies of

frequent items of size one (F1) for support counting, the pattern-confidence is found by a two-way join of Fm-1 and F1.

When m=2 and F1.item1< F1.item1

F2.pattern-confidence = minimum (F1.item1.IC, F1.item1.IC),

When m>2 and F1.item1 < last item of Fm-1 and F1.item1.start-time and end-time is between start and end time of Fm-1.

Fm = minimum (Fm-1.PC, F1.item1.IC)

Fm represents the set of m-length frequent patterns.

4. The sequential window constraint of Hybrid-Apriori automatically satisfies 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) form the significant intervals generated from the SID [n-1] algorithm. The figures in the parenthesis indicate the intervals discovered for the events. Assuming a window of 10 units, the first pass forms AB (1,10), AC (1,15), BC (2,15), CD (7,25). The second pass discovers ABC (1,15). First, if all subsets are above threshold pattern-confidence, ABC is automatically generated in the third pass. 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. Secondly if we assume that the pattern-confidence of sequence BC or any of its subsets is below threshold,

the pattern-confidence of the subset ABC automatically falls below the threshold from the above equation and is pruned out automatically.

5. Another difference with respect to traditional sequential mining lies in the effective use of sequential window parameter. For a given window parameter, two types of interval semantics are defined, which can be used to generate mth item set from the (m-1)th set. Semantics-s generates all possible combinations of events, which occur within window units of the first event. Semantics-e, on the other hand, generates combinations of events that start and complete within the 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.

   Use of semantics-s results in more sequences as compared with semantics-e since events that occur with an interval greater than the window, will not participate in the generation of maximal sequences in semantics-e. Since the output generated between the two semantics greatly differs in quantity, semantics-s can be used to run with representative data sets so as to gather more information on the average pattern-length, size and so on. The process can then be run with semantics-e on the actual dataset, by setting parameters such as stop-level and window-length appropriately.

2.5.2 Benefits and issues in Hybrid Apriori

Being a SQL-based algorithm, Hybrid Apriori has a greater support for large datasets and is able to discover sequences of greater length without facing the space constraints typically encountered by main memory algorithms. Hybrid Apriori takes reduced dataset of significant intervals is input. The size of these intervals is significantly less compared to the raw dataset. Hence, the time taken per pass is less as compared to the traditional algorithms operating on time stamped data. But the significant intervals discovered by SID are, however, not lossless. The periodicity information is lost due to the folding of data during the interval formation phase. Due to folding, the episodes discovered by Hybrid-Apriori may have false positives in it. There may be episodes that are discovered as occurring on all days of the week but these actually occur only on a particular day. Detection of false positives and their elimination is critical to domains such as Smart home, telecommunications alarm management, and crime detection. In our thesis, we consider the problem domain to be a smart home - MavHome. The MavHome (Managing An Intelligent and Versatile Home) project 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 [19]. Finding frequent patterns enables us to automate device usage and reduce human interaction. The MavHome project focuses on the creation of a home that acts as a rational agent. We propose several approaches to identify the false positives in the frequent episodes discovered and discuss the issues faced in each approach with their proposed solutions.

By distinguishing the false positives from the frequent episodes discovered, the objectives of MavHome will be served with greater accuracy.

CHAPTER 3

APPROACHES TO VALIDATE FREQUENT EPISODES

In chapter 1 (Introduction), we briefly explained why it is important to identify the false positives in the frequent episode discovered for interval based time-series data. In this section, we explain why false positives are generated and propose approaches to identify and prune them from a set of given frequent episodes.

3.1 False Positives and Periodicity of Frequent Episodes

Hybrid-Apriori discovers episodes for two types for periodicities; daily and weekly. It can also be further generalized to monthly and yearly periodicities. In the daily periodicity, the entire dataset is folded over 24-hour period. Weekly periodicity, in contrast, takes into consideration the time component as well as 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 month.

3.2 False Positives and the Process of Discovery of Episodes – An Illustration

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

Consider a small two weeks dataset. This data set has two events, “Fan On” and “Lamp On”, representing a sample scenario where the inhabitant uses the study room. The following graph 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.

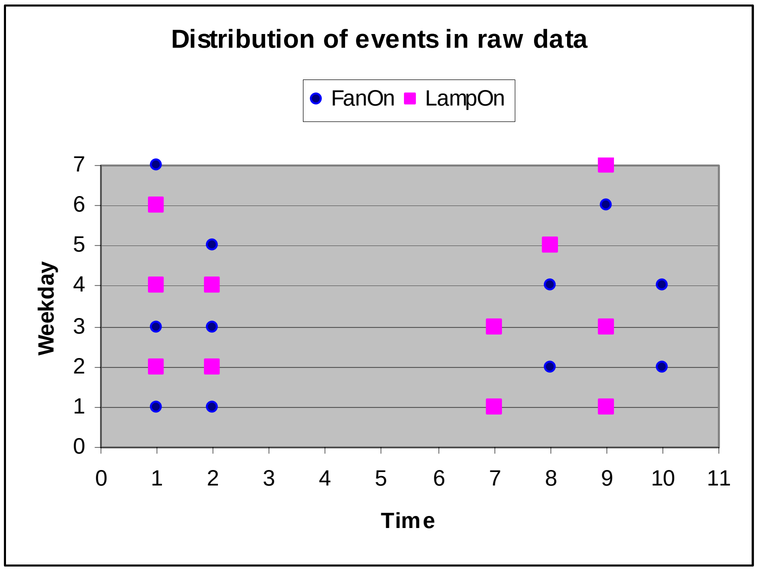


Figure 2 Distribution of events in raw data set

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 e.g., “Lamp On” event which occurred at time t=9 units on weekdays 1, 3 and 7 now has a support of three at time t=9 units.

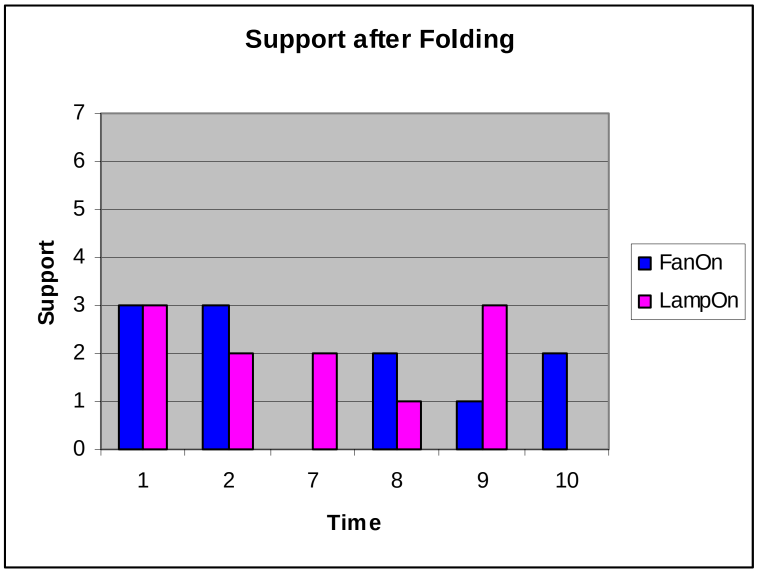


Figure 3 Raw data set after folding

The Significant Interval Discovery (SID) algorithm works on the folded dataset and discovers significant intervals based on user specified parameters such as interval length and interval confidence. Significant intervals discovered for each device are shown in the following graph.

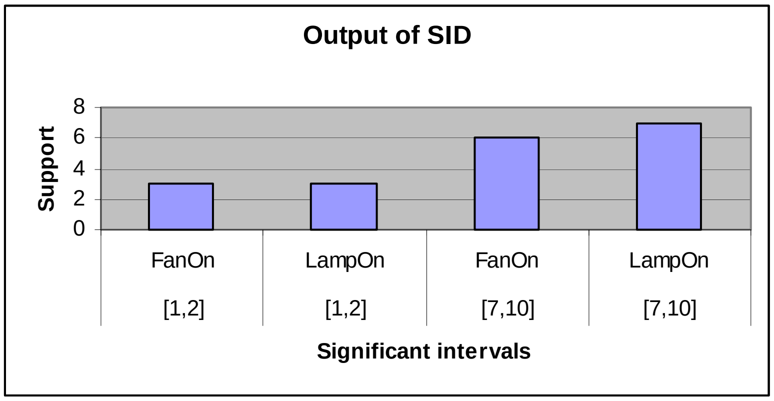


Figure 4 Significant intervals discovered by SID

The episode discovery algorithm takes the SIDs discovered in the previous step as input and finds the frequent episodes based on user specified parameters such as sequential window, episode confidence, and maximum episode size. The number of events in an episode determines the size of the episode. Two episodes of size two are displayed in the figure.

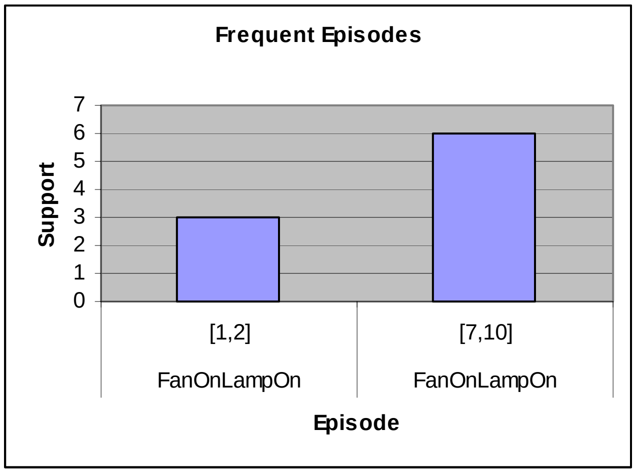


Figure 5 Episodes discovered by Hybrid Apriori

With the small dataset 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 raw dataset available, then we can compute the following statistics. This can help us decide whether an episode is a false positive or a valid episode.

The statistics in the table below 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.

Table 1 Support of Events in an Episode

Episode Start Time	7		
Episode Start 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

As seen from the above table, the event "Fan On" occurred on Monday, Wednesday and Saturday whereas "Lamp On" 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 (i.e., the weekday when it occurs).

3.3 Algorithm Overview

We propose a main memory algorithm that makes a single pass over the raw dataset and the frequent episodes generated by the Hybrid-Apriori algorithm This main memory algorithm will select the correct episodes and eliminate the false positives present in the set of frequent episodes discovered by the Hybrid-Apriori algorithm. Multiple approaches to validate the episodes have been developed to address the issues of response time, performance, and scalability.

The algorithm to validate episodes takes frequent episodes produced by the Hybrid-Apriori algorithm as input. It eliminates the false positives in the input to give a set of valid episodes as the final output. It scans all the events in the raw data set once and computes the support of each event/item in the episode based on the granularity specified during the discovery of episodes. The granularity may be daily or weekly. Unless specified explicitly, we discuss the case of daily periodicity in this chapter. If the support of the any item/event in the episode is less than the minimum support required for an episode then the episode is identified as a false positive.

The algorithm to validate episodes can be partitioned into three phases:

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

3.3.1 Building Phase

This phase retrieves the episodes discovered by Hybrid-Apriori 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/Os. 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.

3.3.2 Support Counting Phase

The computing phase makes a single pass over the raw data set and computes the support for each event in an episode for a specified granularity. For each event in the raw dataset, its episode list is fetched. This episode list gives the list of episodes where this event occurs. For each episode in this list, we check if the transaction time of the event falls in the range of the episode interval. If the time is in the range, we ungroup the transaction time and extract the day when the event occurred and accordingly update the statistics for the event in the episode. This requires ungrouping of the transaction time into time granularity – a transaction time such as “11-23-2005 22:10” for an Event D1 is ungrouped into “22:10 Wednesday November 2005” and update the support for the event D1 for Wednesday. Thus at the end we have the support statistics for each event in the episode ungrouped based on the periodicity of the episode.

3.3.3 Pruning Phase

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

3.4 Basic Issues in Identifying False Positives

This section explains the issues addressed in order to identify the false positives in the frequent episodes discovered by Hybrid apriori. The issues discussed are: periodicity of the episode, wrapping episodes, size of the episode discovered and computing the support of events in an episode in a single pass

3.4.1 Periodicity

Due to the folding and interval representation of raw data, information regarding the next-level granularity is lost. Thus, this lost information is not taken into account at the time of generating frequent episodes. This may lead to the generation of false positives. In order to identify the false positives, we need to go from a low granularity of time to a higher one. For this, we need to identify whether all the events in the frequent episode discovered in a given time interval occurs together on the same day or on different days.

For a given episode with daily periodicity shown below,

Table 2 Example of an Episode

Episode	Event1	Event2	StartTime	EndTime	Confidence
73	LampOn	RadioOn	14:29:00	14:37:00	0.8

We need to compute support count for each event for all the weekdays such as:

Table 3 Support of Events in an Episode

Episode StartTime	14:29:00		
Episode EndTime	14:37:00		
Episode Confidence	0.8		
Event	LampOn	Weekday	Support
		Sunday	2
		Monday	3
		Tuesday	27
		Wednesday	22
		Thursday	70
		Friday	59
		Saturday	6
Event	RadioOn	Weekday	Support
		Sunday	10
		Monday	29
		Tuesday	34
		Wednesday	23
		Thursday	41
		Friday	14
		Saturday	12

Based on the support counts computed for each weekday, we infer whether all the events in an episode meet the minimum support threshold for at least one common week day. An episode with all its events satisfying this condition is considered as a valid episode. Else, it is a case of false positive and is eliminated from the set of frequent episodes. Let us consider the scenario of a smart home inhabitant using the laundry room on weekends. In order to automate and thereby reduce the inhabitant's interaction with the devices, we need to identify the day on which the frequent episode representing the laundry scenario occurs. The episode discovered by Hybrid-Apriori does not give this information. However, after our algorithm that validates the frequent episodes makes a pass over the raw data set, we are able to unfurl the higher granularity information lost during the folding phase and detect with certainty the day/days on which an episode occurs.

3.4.2 Wrapping Episodes

The validation of episodes based on periodicity is complicated by the type of episodes discovered by the Hybrid Apriori. The episodes discovered by Hybrid-Apriori are of two types. They could be normal episodes or they could be episodes generated due to folding. The normal episodes start and end on the same day but due to the inherent time-wrap property of time-series data, episodes spanning two periods/days are discovered. Such episodes are defined as wrapping episodes. Computation of support and validation of such episodes is different from the normal episodes. We illustrate this with the help of following example:

Raw dataset:

1. Fan On 16 Jul 2005 23:51:00
2. Fan On 16 Jul 2005 23:52:10
3. Fan On 17 Jul 2005 00:07:00
4. TV On 16 Jul 2005 23:55:10
5. TV On 17 Jul 2005 00:05:45

6. TV On 17 Jul 2005 00:10:10

Folding of raw data:

1. Fan On 23:51:00
2. Fan On 23:52:10
3. Fan On 00:07:10
4. TV On 23:55:10
5. TV On 00:56:10
6. TV On 00:10:00

Significant Interval discovered by SID

1. Fan On 23:51:10 00:07:00 IC1
2. TV On 23:55:10 00:10:00 IC2

Episode discovered by HA

1. Fan On TV On 23:51:10 00:10:00 PC1

This episode spans two days. It starts on Saturday night and ends on Sunday morning.

We divide this episode into two sub episodes and compute support for the first one for the interval [Start time of the episode, midnight] and for the second one for the interval [Midnight, End time of the episode] and add the support of the two to get the total support of the folding episode. We illustrate this with the following example:

Table 4 Example of a Wrapping Episode

Episode	Event1	Event2	StartTime	EndTime	Confidence
79	FanOn	TVOn	23:51:00	0:10:00	0.8

For a wrapping episode, we compute support for two sub-intervals: [23:51:00, 0:00:00] and [0:00:00, 0:01:00] as shown below:

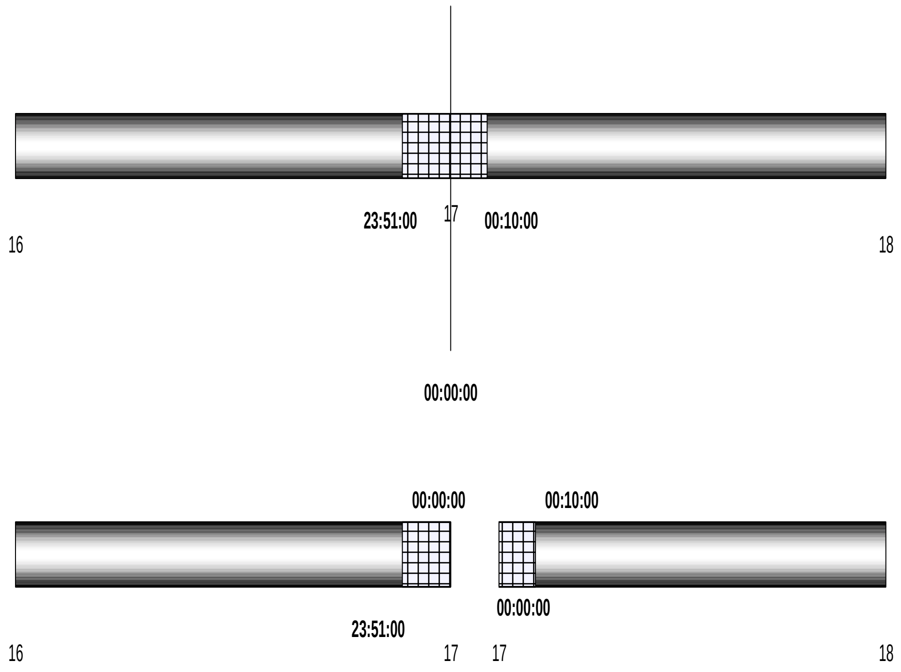


Figure 6 Wrapping Episode - An Episode spanning multiple periods/days

The following table shows how we compute the final support for a wrapping episode. Here the support for a device FanOn in interval [23:51, 00:00] on Monday is added to the support of FanOn in interval [00:00, 00:10] on Tuesday and not [00:00, 00:10] on Monday to get the correct final support for a folding episode.

Table 5 Support Count of each Event for Daily Periodicity

Episode StartTime	23:51:00			Episode StartTime	**0:00:00**			
Episode EndTime	**0:00:00**			Episode EndTime	0:10:00			
Episode Confidence	0.8			Episode Confidence	0.8			
Event	**FanOn**	**Weekday**	**PartialSupport1**	**Event**	**FanOn**	**Weekday**	**PartialSupport2**	**TotalSupport**
		Wednesday	34			**Thursday**	**2**	**36**
		Thursday	61			Friday	6	67
		Friday	38			Saturday	2	40
		Saturday	21			Sunday	1	22
		Sunday	24			Monday	4	28
		Monday	34			Tuesday	5	39
		Tuesday	27			Wednesday	5	32
Event	**TVOn**	**Weekday**	**PartialSupport1**	**Event**	**TVOn**	**Weekday**	**PartialSupport2**	**TotalSupport**
		Wednesday	**27**			**Thursday**	**1**	**28**
		Thursday	56			Friday	5	61
		Friday	27			Saturday	2	29
		Saturday	22			Sunday	1	23
		Sunday	17			Monday	3	20
		Monday	9			Tuesday	2	11
		Tuesday	23			Wednesday	1	24

3.4.3 Size of the episode discovered

The number of items/events in an episode determines the size of an episode. Hence the number of events in an episode is not known before hand and has to be determined at runtime to represent it correctly in main memory.

3.4.4 Computing the support of events in an episode in a single pass

In order to compute the support of an event in an episode for each weekday in a given time interval, we can make several passes over the raw dataset and update support counts for each event in an episode. For large datasets, this would be inefficient. We propose multiple approaches that can identify the false positives in a single pass over the raw dataset. In addition, these approaches also address the issues of performance and scalability. The proposed approaches are:

Approach#1: Naïve Approach

Approach#2: Partition Approach

Approach#3: Parallel Approach

We describe each of them in terms of their design issues, significant differences, advantages and limitations. In the next chapter, we explain the implementation issues of each approach with the proposed solutions.

3.5 Analysis of Time Complexity

Let us assume the following:

p denotes the size of the raw data set

t represents the total number of unique devices in the raw dataset of size p

q represents the total number of episodes to validate r is the average size of the episode / average number of devices in the episode<=t

s is the average number of episodes where a single device occurs<=q

The time complexity of the entire algorithm is O(p)*O(s)*O(r)

An event in an episode has three characteristics: event name, event status and event time. In our validation approach, we create two hash tables: hash table of episodes and a hash table of events. The hash table of episodes contains episodes hashed on the episode-id while the hash table of events contains a list of episode-id grouped by the events in the episode. For every event in the raw dataset, we hash the episode-id hash table on the event name and its status. This operation is O(1) and it returns us the episode-id list which contains the id of episodes where the event occurs. The size of this episode list is assumed to be s. For each episode-id in the list we hash the episode hash table and get a corresponding episode. This operation is O(1). The number of events in an episode determines its size and we assume it to be r. In order to update the support of an event, we need to locate this event in the episode. The events in an episode are stored in a vector and the operation to fetch the correct event and its support array takes O(r) where r is the average number of events in any episode. Once the appropriate event is located, we update the support of the event stored in an array which is O(1). For each event instance in the raw data set, we fetch all the episodes where this event occurs and update the corresponding support. Hence the response time of the algorithm heavily depends on the number of episodes where an event occurs and this number increases with the increase in number of episodes (q) to be validated.

3.6 Naïve Approach to Identify False Positives

This main memory algorithm validates the episodes discovered by the Hybrid-Apriori algorithm by identifying the false positives. Each frequent episode is stored in main memory and the support count for all the events in the episode are computed by making a single pass over the raw data. At the end of the pass, we have the support count of each event in an episode ungrouped on the periodicity specified. This ungrouped support count is then compared to the minimum support threshold to identify and prune the false positives in the set of episodes validated.

3.6.1 Pseudo code for Building Phase

The pseudo code for the building phase in the naïve approach to validate the frequent episodes based on periodicity consists of the following steps:

```
For each episode detected by Hybrid-Apriori algorithm
    Fetch the episode and determine the type of episode
    Store the frequent episode in main memory
    For each event in the episode,
        If the episode list exists for this event,
            Add the episode Id of this episode to the list
        Else
            Create an episode list for this event
            Add the episode Id of this episode to the list
```

At the end of build phase, we have the following two data structures populated with the episodes and the episode list – set of episodes grouped by the events in the episode

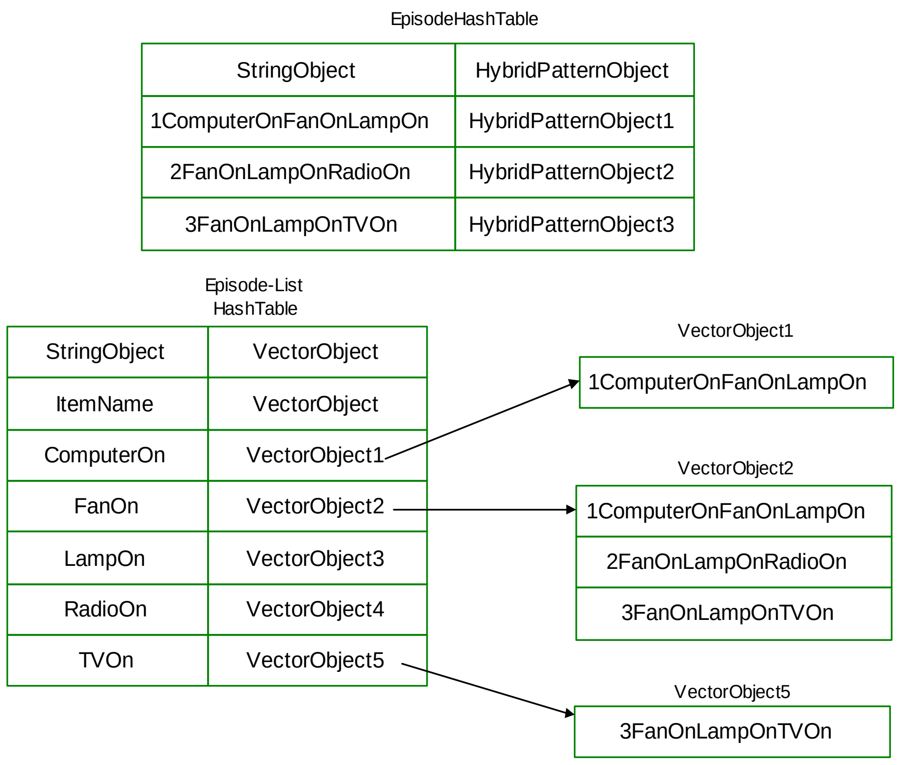


Figure 7 Output of Building Phase

3.6.2 Pseudo code for Support Counting Phase

The pseudo code for the support counting phase in the naïve approach consists of the following steps:

Fetch an event transaction from the raw dataset

Retrieve the corresponding episode list

For each episode in the episode list

Update the support statistics for this event if the transaction time falls in the episode time interval

At the end of the support computation phase, support count for a given granularity is available for each event in the episode. The data structure representing the episode and the state of the episode after the computation phase now looks as follows:

Table 6 Episode with daily periodicity

Episode	Event1	Event2	StartTime	EndTime	Confidence
73	LampOn	RadioOn	14:29:00	14:37:00	0.8

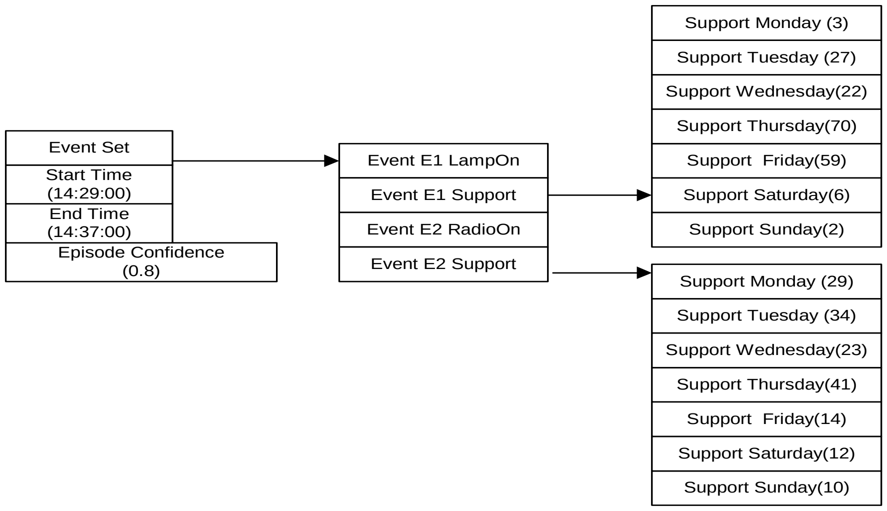


Figure 8 Output of Support Counting Phase

3.6.3 Pseudo code for Validate Phase

```
1.  For each episode in the memory
2.      Determine the type of episode
```

```
3.    If the episode is a normal episode
4.        Determine the number of events in the episode
5.        For each weekday
6.            For each event,
7.                **Fetch the support count for the weekday**
8.                **Compare this support count with the support threshold value**
9.                If the support count is greater than the support threshold
10.                   Set the EventValid flag to true
11.               Else
12.                   Set the EventValid flag to false
13.                   **Break** //no need to check the other events in the episode for
                       this weekday
14.               **If** EventValid is true
15.                   Set episodeValid flag to True
16.               Else
17.                   Set episodeValid flag to false
18.   Else If the episode is a spanning episode
19.       Determine the number of events in the episode (Same as line#4)
20.       For each weekday  (Same as line#5)
21.           For each event, (Same as line#6)
22.               **Fetch the support count for two weekdays: current and the
                   immediate next**
```

```
23.             Compare the sum of the support count of two days with the support
                threshold value
24.             If the support count is greater than the support threshold (Same as
                line#9)
25.             Set the EventValid flag to true (Same as line#10)
26.         If  EventValid is true (Same as line#14)
27.            Set episodeValid flag to True (same as line#15)
28.     If  episodeValid flag is True for at least one weekday
29.         Episode is a valid episode
30.     Else
31.         Episode is a false positive
```

The validation phase analyses the support computed to determine the validity of the episode. This can be depicted as follows:

Table 7 Analysis of Validation Output

No of days = 180 No of weeks = 26 Min Confidence=0.7 Min Support = 18.2	Support of Event E1 LampOn	Support of Event E2 RadioOn	Episode Status Support of all events > MinSupp
Support Monday > MinimumSupport	No	Yes	InValid
Support Tuesday > MinimumSupport	Yes	Yes	Valid
Support Wednesday > MinimumSupport	Yes	Yes	Valid
Support Thursday > MinimumSupport	Yes	Yes	Valid
Support Friday > MinimumSupport	Yes	No	InValid
Support Saturday > MinimumSupport	No	No	InValid
Support Sunday > MinimumSupport	No	No	InValid

3.7 Design for Algorithm to Validate Frequent Episodes

3.7.1 Design for Building Phase

The building phase for the naïve approach accomplishes two things: One, it represents all the episodes using main memory data structures. Two, it groups all episodes by the events in it; by creating an episode-id list. The creation of episode-id list is done simultaneously with episode caching. For each event in the episode, we either create a new episode-id list or update the episode list if one exists. An episode list exists for events occurring in multiple episodes. This episode-id list is used in the next phase, the computation phase, to retrieve all the episodes corresponding to an event while scanning the raw data.

As shown in figure 7, the building phase constructs two hash tables in main memory. The first hash table consists of the episodes. Each episode is hashed into one bucket. Simultaneously we construct the second hash table that contains the list of episodes grouped by the devices in the episode. Each bucket in this hash table is a list of episode grouped by the events in the episode. As observed from the figure, the event “FanOn” occurs in three episodes. Hence the episode id hash table contains a list of three episode-ids in them. Based on this episode id we can retrieve the episode from the hash table of episodes.

3.7.2 Design for Support Counting Phase

Once all the episodes discovered by the Hybrid-Apriori are stored in main memory and episode lists are created for each unique event, we scan the raw data set. For each device/event transaction fetched, a corresponding episode list is retrieved. We then traverse through this episode list sequentially to fetch an episode_id one at a time. We then retrieve the episode corresponding to this episode_id from the main memory data structure that has all the episodes. Once the episode is retrieved, we have the start time (Ts) and the end time (Te) of the episode. We check whether the transaction time of the device/event in the raw data set is within the interval [Ts, Te]. If it falls in the interval range, we further drill down into the transaction time and fetch the day – Sunday, Monday, …, Saturday – on which the event occurred and update the support count of the event in the episode for that particular day of the week. This is an iterative process which is repeated for each episode whose episode-id exists in the episode lists for the event in the transaction fetched from the raw dataset.

To summarize, we make a single pass over the raw dataset, and for each event E_m in the raw dataset we retrieve the corresponding episode list from the main memory data structure. Now, for each episode id in this list we retrieve the corresponding episode from the episodes data structure and update the support statistics of that event E_m for specified granularity.

3.7.3 Design for Pruning Phase

The computation phase computes the support count of all the events in an episode for a given periodicity. In the pruning phase, we retrieve each episode and compare the support of each event in the episode against the minimum support threshold. If all the events in an episode satisfy the minimum support threshold for a given periodicity then the episode is considered to be a true episode else it is considered a false positive. The periodicity could by daily or weekly. For daily periodicity, we need to make sure that all the events in an episode satisfy the minimum support threshold for the same weekday. For weekly periodicity, we make sure that the weekday on which the episode occurs is in the same month of the year.

3.8 Characteristics of the Naïve approach

This approach represents each episode as a main memory object and validates it. Hence the number of episodes that can be validated would be directly proportional to the main memory available. Moreover, the time taken to validate all the episodes will be linear to the number of episodes discovered.

This approach makes one pass over the episodes generated by the HA algorithm to create in-memory data structures. It makes one pass over the raw data set to populate the in-memory data structures created during the build phase with support values. Finally, the data structures are examined to differentiate between false positives and invalid episodes.

Note that the Hybrid-Apriori algorithm does not generate false negatives. In order to generate a false negative, it has to output an episode that does not have enough support and confidence. On account of folding the support can only increase and cannot decrease. In addition, the Hybrid-Apriori algorithm produces and output in which all episodes satisfy the confidence and interval constraints. Hence false negatives are not generated.

The main memory requirement of this algorithm is proportional to the number of episodes, number of events in each episode and the granularity size that is being validated (e.g., 7 days if folded on daily, 12 months if folded on weekly etc.). For large number of episodes the memory requirement may become high and hence this approach may not be scalable for data sets that generate large number of episodes.

3.9 Partitioned Approach to Identify False Positives

In order to overcome the amount of main memory needed, we apply the divide and conquer rule in the partitioned approach. We implement a validation algorithm which partitions the input data and the episodes to be validated. The partition can be done either on the basis of time or the number of episodes. The partitions are processed sequentially and hence the memory requirement is proportional to the number of episodes in a partition and not the total number of episodes to be validated. Each partition contains the normal episodes, the wrapping episodes and the spanning episodes. The normal episodes are the one that start and end in the same partition while the spanning episodes are those that span across multiple partitions. The wrapping episodes are the one which span across multiple periods and are formed due to the inherent time wrap property of time-series data. For each partition, the false positives among the normal episodes are identified at the end of the validation process while the spanning episodes that do not have the minimum support are carried forward to the next partition for further validation. The wrapping episodes are different from the spanning episodes in the sense that they are always validated in the last partition. The reason that wrapping episodes may start or end in any partition or they may span across multiple partitions but since we start the validation process from the first partition we cannot compute the final cumulative support until we have scanned the entire set of raw data events i.e. reached the last partition. The following figure shows the distribution of episodes in a partitioned approach.

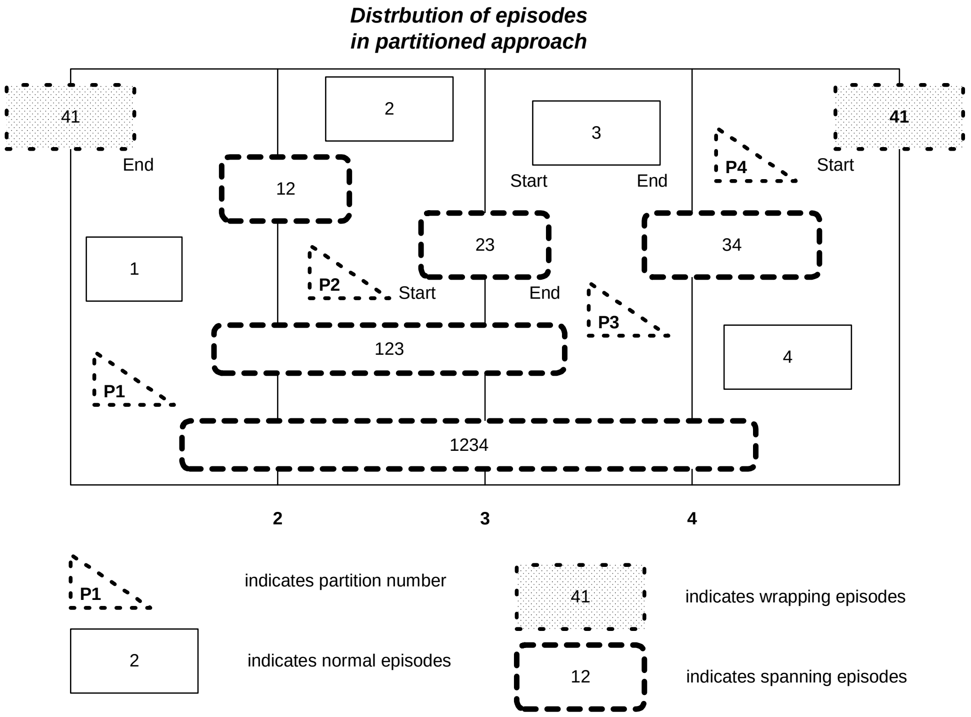


Figure 9 Distribution of Episodes in Partitioned Approach

The above figure shows the partitioned approach for four partitions. As seen, there are three types of episodes we need to handle here. They are the normal episodes, wrapping episodes and the spanning episodes. In the figure above, the normal episodes are episode number 1, 2, 3 and 4. These episode start and end in the same partition. We build them into the main memory, compute their support and validate them in the same partition. The second type is the wrapping episodes. Episode number 41 is an example of wrapping episode. This episode is discovered by Hybrid-Apriori due to the inherent time-wrapping property of time-series data. This episode spans at least the last and the first partition and depending on the episode length it may span across multiple partitions. The third and final type of episodes is the spanning episodes. The spanning episodes in the above figure are episodes number 12, 123, 1234, 23 and 34. This episodes span across at least two partitions and may span across multiple partitions. In order to validate the wrapping and the spanning episodes, we need to compute their partial support in each partition where they span. The partial support of each episode has to be carried forward to the consecutive partitions to get their cumulative support. The end time of the episode determines where an episode ends and need to be validated and pruned to avoid any more computation.

3.10 Issues in Partitioned Approach

3.10.1 Size of a partition

In order to overcome the limitations of main memory, we partition the number of episodes based on the main memory available. The number of partitions is a user-defined parameter or can be inferred based on the main memory available. Pragmatically, the number of partitions should be such that all the episodes in a single partition can fit into the available main memory.

3.10.2 Distribution of episodes

Distribution of episodes is extremely important in the partitioned approach to achieve the desired performance. The following scenarios explain why the distribution of episodes needs to be considered before we partition the given set of episodes.

Case#1a: All the inhabitants of MavHome works from home

Case#2a: All the inhabitants of MavHome works from office and the office timings are 10 am to 5 pm

Case#1b: Customers going to Wal-Mart between 5pm and midnight

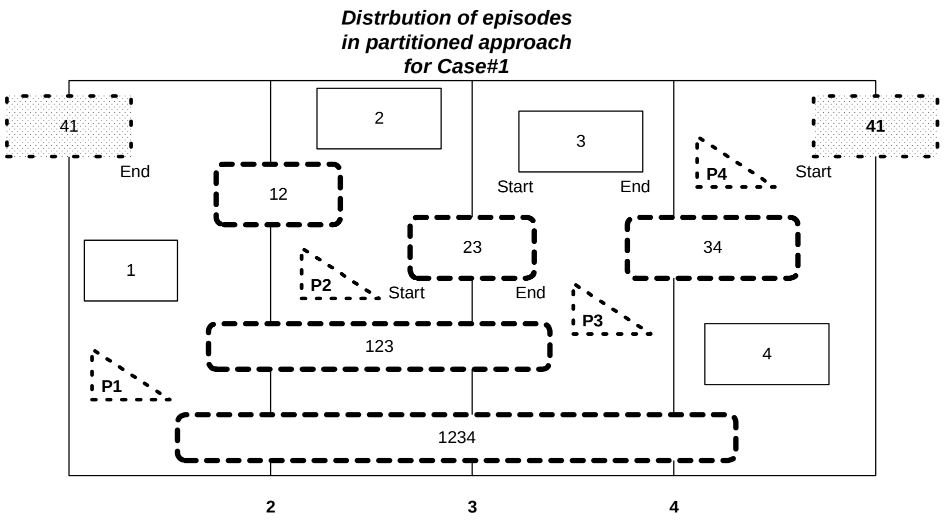
Case#2b: Customers going to Wal-Mart between 10am and 5 pm

Case#1c: People going to watch movie between noon and 6pm

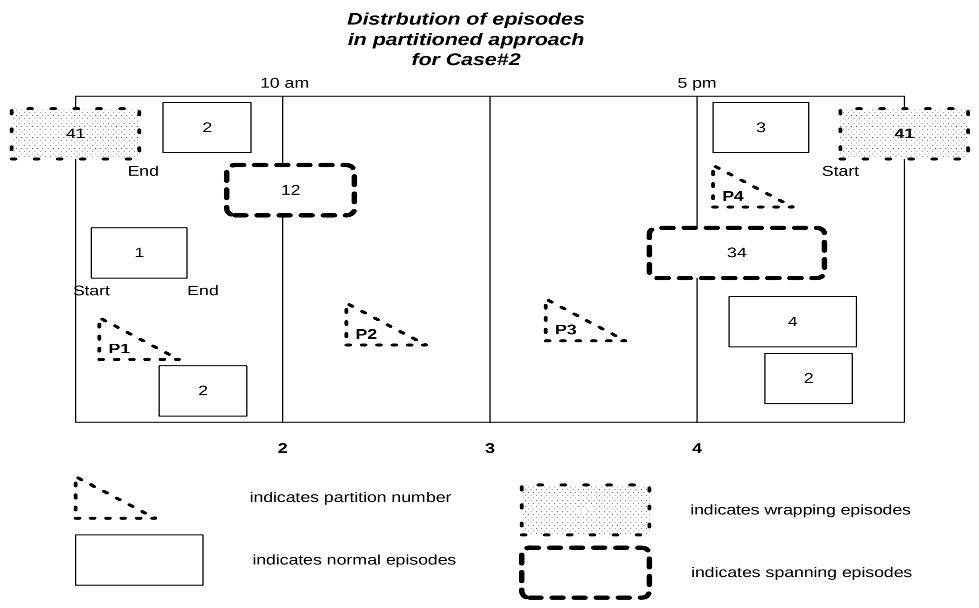
Case#2c: People going to watch movie between 6pm and midnight

In the above scenarios, cases 1a, 1b and 1c represent uniform distribution or regions of high activity while cases 2a, 2b and 2c represent non-uniform distribution or regions of low activity where the number of event instances is few.

The sample distribution of episodes discovered for cases 1a, 1b or 1c would be similar to the following figure while the figure [x] represents the distribution of episodes for cases 2a, 2b or 2c. Hence a single approach to partition the episodes would not give partitions with an approximately equal number of episodes in it.



(a)


Distrbution of episodes in partitioned approach for Case#2
10 am
5 pm
41
2
3
41
End
Start
12
P4
1
34
Start
End
4
P1
P2
P3
2
2
2
3
4
indicates partition number
indicates wrapping episodes
indicates normal episodes
indicates spanning episodes

(b)

Figure 10 Distribution of Episodes in a partition (a) Uniform (b) Skewed.

In the above figure, partitioning the non-uniform distribution of episodes using the fixed partition scheme creates partition numbers P2 and P3 that are the regions of inactivity – the time period when all the inhabitants are not at home. These partitions either have very few episodes or no episodes to validate. These two cases demonstrate the fact that a single divide and conquer approach would not give the desired performance benefits if partitioning the set of frequent episodes does not create partitions with an approximately equal number of episodes to validate. In order to ensure the best performance, we propose two approaches for partitioning the episodes. The first approach is the case where the distribution of episodes in a data set is uniform. Here, the episodes are assumed to be uniformly distributed over the periodicity (daily or weekly). Hence partitioning on fixed time values would generate approximately equal number of episodes in each partition. For example, if the number of partitions is set to four then we divide the entire day into four equal parts: 0-6, 6-12, 12-18, and 18-24. All the episodes that start before 6 am belong to the first partition while episodes starting between 6 am and noon are assigned the second partition and so on. The second approach is for non-uniform distribution as demonstrated by case#2 in the figure above. Applying the fixed scheme creates partitions that either have lot of episodes or have very few episodes in it that leads to imbalance in the computational load. This defeats the purpose of partitioning a large set of episodes into partitions manageable with the available memory. Our second approach ensures that balance in computational load is achieved by assigning approximately equal number of episodes and keeping the number of episodes close to each other across all partitions. This approach takes into consideration the total number of episodes rather than their start or end time. This makes the partitioning process independent of the distribution of episodes discovered. More details on this approach are discussed in the implementation chapter.

3.10.3 How to partition an episode

Partitioning of episodes can be done either on the start time or the end time of the episode. Partitioning on start time leads to a natural partitioning process since first and last partition is adjacent logically and you only need to carry forward the support. Natural Partitioning means the first half of the spanning episode will be validated in the current partition and the second half will be validated in the next partition. We can also partition on the end time of an episode. But this will only take care of the episodes whose end time is less than the partition time. It will not consider the episodes whose start time is less than the partition time and which partially belong to this partition.

3.11 Phases in Partition Approach

1. Partitioning Phase
2. Fetching Phase
3. Building Phase
4. Support Counting Phase
5. Pruning Phase
6. Carry forward Phase

3.11.1 Partitioning Phase

The number of partition to be done is a user specified parameter. The algorithm supports two types of partitioning approaches:

3.11.1.1 Uniform Distribution

This approach is a static approach. For n partitions, it divides the periodicity (e.g., 24-hour period) into n equal partitions. For example, for daily periodicity and n=4, the partition points would be 6, 12, 18, and 24. Based on these values, all the episodes and raw data are partitioned. All episodes with their start time less than 6 a.m. and all transactions in raw data with their transaction time less than 6 a.m. are processed into partition#1. While, all episodes with their start time between 6 a.m. and noon and all events in raw data with their transaction time between 6 a.m. and noon are processed into partition#2. Similarly, we partition the rest of the episodes and the raw data available.

This approach works well for episodes with uniform distribution. But for skewed episodes, this either creates partitions with very few episodes or too many episodes leading to an imbalance in the number of episodes in each partition.

3.11.1.2 Non-Uniform Distribution

This approach is used for episodes with a skewed distribution. It aims to balance the number of episodes in each partition. Instead of partitioning on fixed time units, this approach partitions the episodes based on the number of episodes. Suppose that in the set of frequent episodes discovered, we have 100 episodes of size two, 150 episodes of size three and 60 episodes of size four. Thus the total number of episodes is 310 and if there are four partitions, than each partition should receive approximately 77 episodes each. We cannot balance the partitions equally but are able to balance the load nearly equally since the size of the episode, the number of episodes of each size and their corresponding start and end times are not fixed.

Once the partition points are computed for both the approaches, we choose an approach that creates partitions of approximately equal sizes and proceed to next phase.

3.11.2 Fetching Phase

In this phase, the episodes that span in this partition are fetched from the other partitions so that their partial support could be computed. For the first partition, the folding episodes are fetched and included in the building phase. For the rest of the partitions, the fetch phase comes after the building phase since the spanning episodes already exist in the main memory with their partial support computed in the previous phases.

3.11.3 Building Phase

Due to partitioning, now normal episodes can span across two partitions. These are now classified as spanning episodes along with the folding episodes which by default span two partitions. Hence, while building the episodes in main memory we identify and tag the episode type for later use. The episode is tagged as a normal episode, a folding episode or a spanning episode. Based on the type of episode, the pruning phase decides whether to validate an episode or carry forward to the next partition. The rest of the process is similar to the respective phase in the Naïve approach.

3.11.4 Support Counting Phase

This phase is similar to respective phase in the Naïve approach. For a given partition of episodes, the raw data corresponding to this partition is scanned to compute the support for each event in an episode.

3.11.5 Pruning Phase

The only difference in this phase with respect to the Naïve approach is that not all the episodes whose support is computed are validated. Only the normal episodes and the spanning episodes which end in the current partition are validated. The remaining spanning episodes are carried forward to the next partition for further processing.

3.11.6 Carry forward Phase

In this phase, the spanning episodes that do not end in the current partition are carried forward with their partial support count to the next partition.

3.12 Advantages and Limitations of Partitioned Approach

This approach allows us to validate each partition in isolation thereby reducing the main memory requirements and allowing a larger set of episodes to be validated. Instead of representing all the episodes in the main memory at the same time, we do it in chunks. We partition the set of episodes and consider each partition in isolation. This allows us to deal with larger set of episodes, for a given amount of main memory. Partitioning also reduces the response time significantly. As discussed in section 3.5, the complexity of the algorithm is O(p)*O(s)*O(r) where p is the size of the raw data set , s is the average number of episodes where a single device occurs and r is the average size of the episode / average number of devices in the episode. The value of s decreases with the decrease in number of episodes in a partition and would thus lead to reduced response time. This approach thus takes care of the scalability issues but validates each partition sequentially and hence the total time required can be further reduced if all the partitions are validated in parallel.

3.13 Parallel Approach to Identify False Positives

This approach is somewhat similar to the partitioning approach. The major difference between the two is that this approach processes all the partitions in parallel while the partition approach does it sequentially.

In this approach, we assume the availability of number of processors, all with the same architecture. We assign one partition to each processing node. The computation phase for all the episodes and the pruning phase for the normal episodes are executed in parallel at each node. Here we introduce an additional phase, the merge phase, which takes care of the spanning episodes in the set of episodes discovered. This merge phase starts after all the nodes finish their respective pruning phase and return the spanning episodes with their partial support counts.

3.14 Issues in Parallel Approach

3.14.1 Episode spanning multiple partitions

Each partition is validated on a different processor. Hence we need to ensure that an episode that spans multiple partitions is included in each partition before we start validating an episode. We need to identify the spanning episodes and duplicate the spanning episode in all the partitions to compute their partial support. The partition approach was a sequential approach and hence dealing with the spanning episodes was quite straightforward. But here since all the partitions are to be processed in parallel we need to identify the span of an episode before we start the entire process of identifying the false positives. This is needed to ensure that the partial support of spanning episodes is computed at all partitions where it spans. To explain how we compute the overlapping episodes, we revisit the figure explained in the partitioned approach.

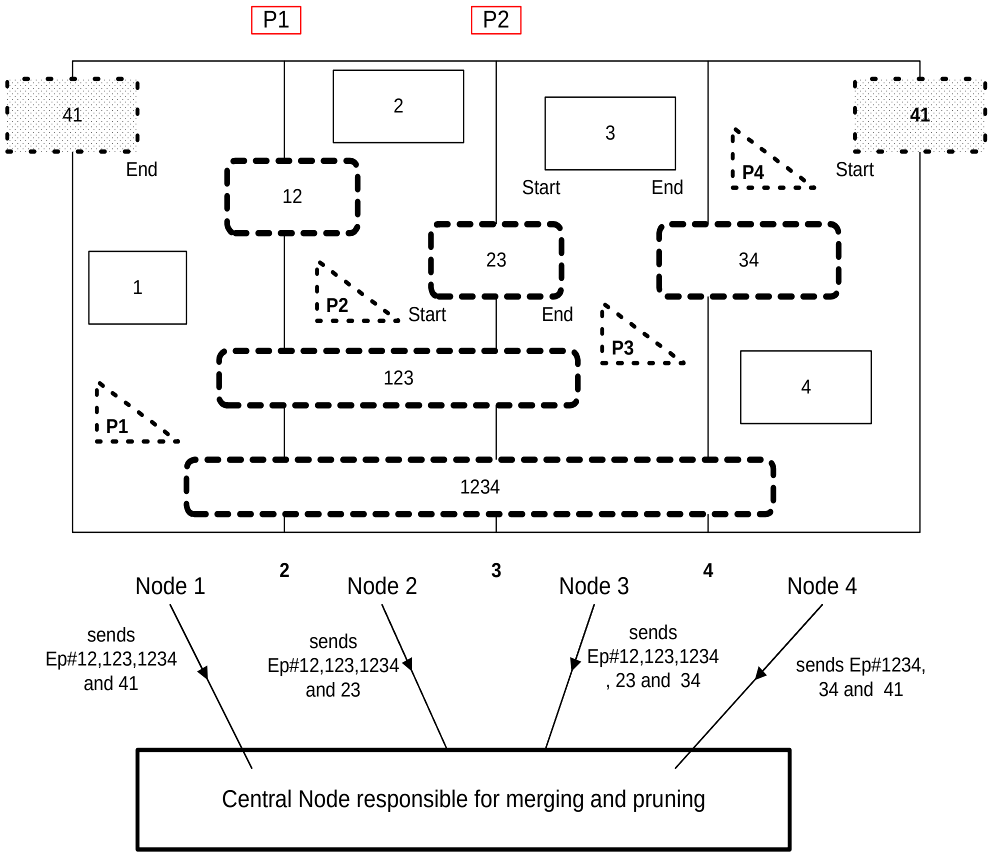

P1
P2
41
2
41
End
3
P4
Start
12
Start
End
23
34
1
P2
Start
End
P3
123
4
P1
1234
Node 1
2
Node 2
3
Node 3
4
Node 4
sends Ep#12,123,1234 and 41
sends Ep#12,123,1234 and 23
sends Ep#12,123,1234 , 23 and 34
sends Ep#1234, 34 and 41
Central Node responsible for merging and pruning

Figure 11 Distribution of Episodes after Partition

In the above figure, we have four partitions; one partition is allocated to one machine on the cluster of machines available for the parallel approach. Each partition has a start time and an end time. The start time of the partition is the end time of the previous partition and the end time of the partition is the start time of the consecutive partition. Similarly an episode has a start time and the end time. We use these four characteristics to compute the set of episodes to be validated by a machine. For each machine in the cluster, we select all the episodes satisfying any of the following conditions:

1. The start time of the episode is greater than the start time of the partition AND less than the end time of the partition.
2. The end time of the episode is greater than the start time of the partition AND less than the end time of the partition.
3. The start time of the episode is less than the start time of the partition AND the end time is greater than the end time of the partition.

The first condition takes care of episodes starting in a partition; the second condition takes care of episodes ending the current partition and the third condition takes care of episodes spanning across the partition. Hence if we consider partition#2, the set of episodes selected by our SQL would consist of episode number 12, 2, 23, 123 and 1234.

Similarly, we select the raw data set corresponding to the set of episodes to be validated in a partition. The partitioning of raw data set is based on the event occurrence time.

The implementation details of the SQL based approach to solve the above problem can be found in the next section.

3.14.2 Merge the partial support count of spanning episodes

The normal episodes are validated by each processor while the validation for spanning episodes is delayed and is done later by a single processor after the merge phase is complete. Each processor sends the partial support count of the spanning episodes to a central node that is responsible for merging the support count of these episodes.

3.15 Phases in Parallel approach

1. Partitioning Phase (One Time)
2. Fetching Phase (Iterative)
3. Building Phase (Iterative)
4. Computing Support Phase (Iterative)
5. Pruning Phase (Iterative)
6. Merge Phase (One Time)

Partitioning Phase

This phase is the same as described in the partition approach. Based on the number of nodes/machines available for parallel computation, we partition and assign a partition to each node.

Fetching Phase

This phase is similar to the fetching phase in sequential approach. The major difference here is that the validation of all partitions is done in parallel and hence we need to duplicate the spanning episodes at each node beforehand. This is done to ensure that the partial support counts of spanning episodes are computed at each node where it spans. We illustrate fetching phase with the following example**:**

HTNi – table of all episodes at node ***'i'***

THTNi – temporary table of spanning episodes at node ***'i'*** to be duplicated at all other nodes in the cluster

E_m = Episode in partition m

E_{mnp} = Episode spanning three partitions: partition number m, n and p.

e.g., E_{234} implies that episode starts in second partition and ends in fourth partition

N= Number of nodes in the cluster=4

E = Set of Episodes to validate=$\{E_1,E_{12},E_{123},E_{1234},E_2,E_{23},E_{234},E_3,E_{34},E_4,E_{41},E_{412}\}$

Episodes to validate at node#1

HTN1 = {E1, E12, E123, E1234}

THTN1 = {E12, E123, E1234}

Episodes to validate at node#2

HTN2= {E2, E23, E234}

THTN2= {E23, E234}

Episodes to validate at node#3

HTN3= {E3, E34}

THTN3= {E34}

Episodes to validate at node#4

HTN4= {E4, E41, E412}

THTN4= {E41, E412}

Episode to validate at each node after duplicating the spanning episodes

HTN1 = {E1, E12, E123, E1234, E41, E412}

HTN2= {E2, E23, E234, E12, E123, E1234, E412}

HTN3= {E3, E34, E123, E23, E234, E1234}

HTN4= {E4, E41, E412, E1234, E234, E34}

Building Phase

The building phase is similar to the building phase in the partitioned approach where each node builds the episode and the episode-id lists in main memory.

Support Counting Phase

This phase is also similar to computing support phase in the partitioned approach where each node computes the support count of events in an episode through a single scan of the raw dataset.

Pruning Phase

This phase validates only the normal episodes. The spanning episodes with their partial support are sent to a central node responsible for validating the spanning episode. Each node sends the spanning episode with their partial support count to this node.

Merge Phase

The merge phase starts after all the nodes in the cluster send the spanning episodes to a central node responsible for merging the partial support of these episodes and validating them. First, the partial support count of each episode from all the nodes is added to get their cumulative support count. Once the cumulative support count for each episode is obtained, the pruning phase starts. This phase, as described in the Naïve approach, is executed to identify whether the episode is a valid episode or a false positive.

3.16 Advantages and Disadvantages

We assume that the number of spanning episodes is less and hence duplicate them at all nodes. If the number of spanning episodes is significant, the number of episodes to be validated by each node would increase. This would make the computing support phase more expensive leading to an increase in the total time to validate. Moreover, each node in the cluster will have to send the partial support count of the spanning episodes to a central node for merging the partial support count and validating them. This is an additional phase compared to the partitioned approach where we carry forward the partial support of spanning episodes and discard once they satisfy the minimum support. The time to merge depends on the number of spanning episodes and is this overhead is meager when compared to the benefits achieved in the total time to validate all episodes in parallel. On the other hand, the merge phase cannot start until it receives spanning episodes from all the nodes taking part in the validation. This might prove to be a bottleneck if load balancing is not achieved while distributing the load to each node in the cluster. For very large data sets where the number of episodes to be validated is really high, we can observe significant reduction in the response time as the number of nodes available to validate increases.

CHAPTER 4

IMPLEMENTATION OF VALIDATION ALGORITHM

We choose a main memory approach to validate the frequent episodes. Here, we need to maintain two types of information for each episode. First, the output of Hybrid-Apriori algorithm needs to be stores, which consists of the events in the episode, the start time and the end time of the episode and the episode confidence needs. Second, we need to compute the support count of each event in the episode for the specified granularity and store it while making a single scan over the raw data set. We compute the support count for each event in an episode by scanning the raw data. To distinguish the false positives from the correct frequent episodes, the validation of events starts after the computation phase. Having all the information at a single place simplifies the analysis and increases the efficiency of the validation phase. In addition, the episode discovered by Hybrid-Apriori could be of different granularity and different sizes. Considering all the possible variations of an episode, we chose to represent an episode as an object. An object gives us the flexibility to encapsulate all the attributes at a single place. We can store multiple objects inside a single object, and decide the size of the object at runtime thereby allowing us to deal with any type of variations in the set of episodes to be validated.

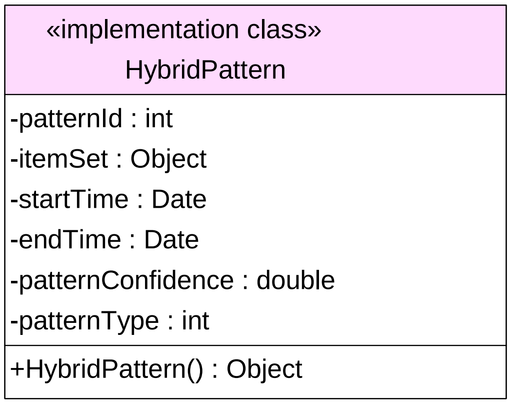


Figure 12 Episode Object

The figure above represents an object that encapsulates the attributes of an episode and also contains the data structures to store the computed support of each event in the episode. The *patternId* uniquely identifies each episode to be validated. The start time and the end time denote when an episode starts and ends and are used to compute the support of events participating in the episode. The *patternConfidence* attribute holds the confidence of the episode discovered. The *patternType* attribute is a derived attribute of an episode. It is used in the partition and the parallel approaches. Its value depends on the partition time-points, the start time and the end time of the episode. It can take one of the three values: {0, 1, 2}. The mapping for these values is: Normal Episode = 0, Spanning Episode = 1 and Folding Episode = 2. The *itemSet* object is a vector that consists of all the events forming a frequent episode. The vector data structure in Java is a dynamic array that can grow at run time as needed. The size of this vector depends on the number of events in an episode. The number of events in an episode is not known in advance and hence a data structure such as an array cannot be used making the vector an ideal choice. Each event in the vector has two attributes: item name and item support. Each event in the *itemSet* vector has an instance of the item object. The class diagram for the item object is shown in the following figure:

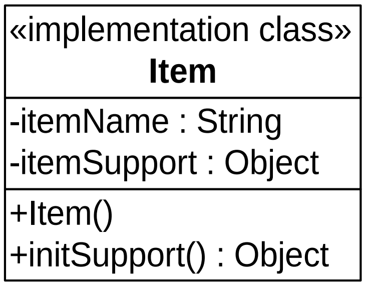


Figure 13 Event Object

Thus, an *itemSet* will contain multiple items and each item would contain the item name and its support array. For daily periodicity, we need to compute the support for all weekdays and hence the array size is seven. For weekly periodicity, we need to compute the support for all months in a year and hence the array size is twelve. We choose an array to store the support of an event on a given day for two reasons. First, the size of the support array depends on the periodicity and the periodicity of the discovered episodes is known to us in advance. Secondly, the array index can be easily used to represent a unique weekday and updating the support would be easy. The Java API for date returns the weekday or the month as an integer and we map this to the array index of the support array. The following figure displays an i*temSet* for an episode of size three with daily periodicity. This vector of events contains the event names and their corresponding support stored in an array.

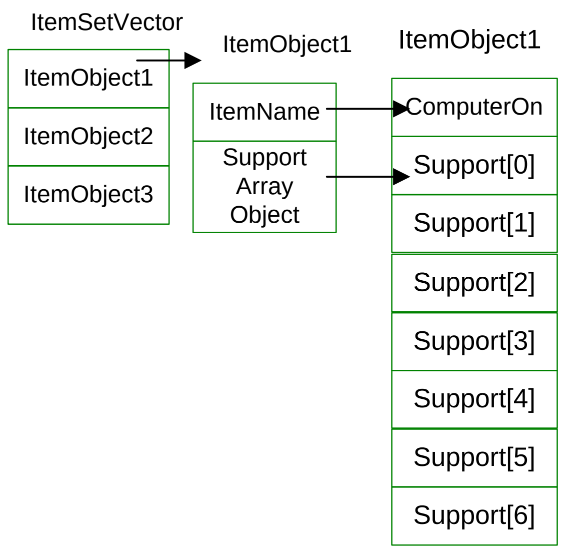


Figure 14 Vector of Events with their Support

The validation process starts with the building phase. Here, we read all the episodes from the database and represent them as objects in main memory with the help of the data structures explained above. We use the hash table data structure in Java to store all the information related to an episode. The building phase creates two hash tables: The first is the hash table of episodes while the second is the hash table of episode-ids grouped by the events in an episode. We maintain two hash tables because updating the support of an event in a given episode is a computationally intensive operation and the number of events in the raw data set is very large. For each event we need to retrieve all the episodes and check whether the event occurrence time in the raw data set is within the range of the start and the end time of the episode. We can do this with a single hash table too. But that would be an exhaustive approach whereby for each event in the raw data set we need to examine the start time and the end time of all the episodes irrespective of the fact that an event may or may not be present in an episode. Even after going through the entire table it is not necessary that an event has at least one corresponding episode where it occurs since the raw dataset has many events that do not occur in any episode. In addition, the number of times the hash table of episodes will be accessed will increase with increase in the size of the raw data set. In order to deal with such situations we create a second hash table. This hash table contains all the episodes grouped by their episode-ids. Thus for a given event in the raw data set we fetch the list of episode-id where it occurs. Now for each episode-id in this list we fetch a corresponding episode from the hash table of episodes and check the time interval to update the support. Thus by grouping the episodes on their events in a separate hash table significant reduction is achieved in the number of episodes we access for each event. In addition, the cost of creating the second hash table is not significant since it is done simultaneously with the creation of hash table of episodes. This hash table contains the episode list that is a vector object which grows dynamically based on the number of episodes where an event occurs. The event is the key and the list of episodes where the event occurs is the value in this episode list hash table. As explained above, this vector of episode-ids in hash table helps us to efficiently retrieve the relevant episodes in the computing phase of the validation.

The following figure displays the hash table of episodes for three episodes and the corresponding hash table of events containing the list of episode-ids where an event occurs.

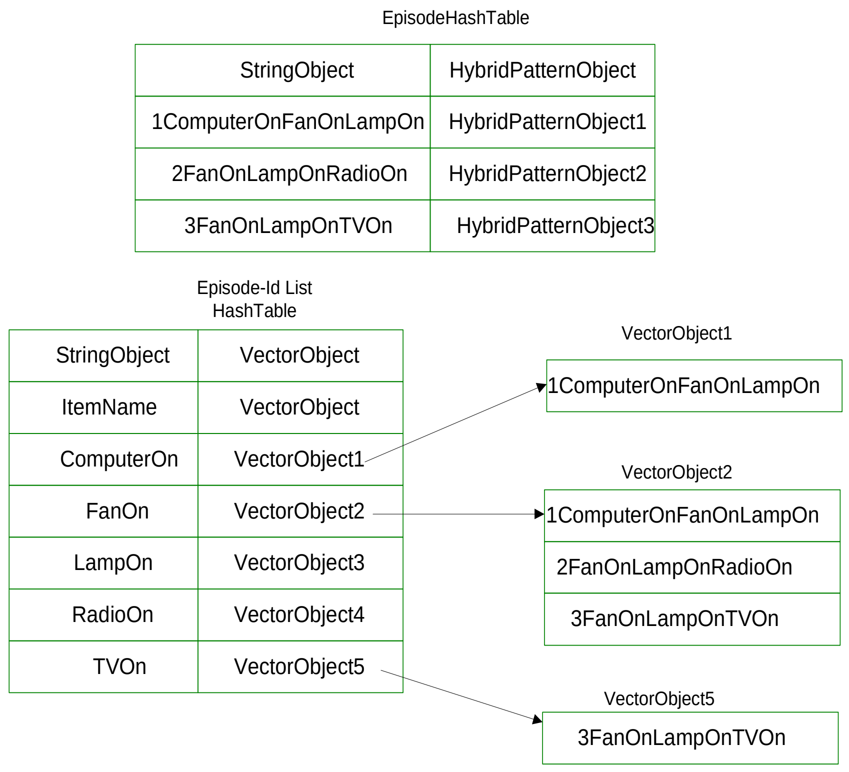


Figure 15 Hash Table of Episode and Episode-Id

We now explain the different phases in the validation algorithm with respect to the implementation details. The validation phase starts with the building phase that reads all the episodes from the database and creates a hash table of episodes and a hash table of episode list. It reads an episode at a time from the database and stores in the main memory data structure discussed above. The episode-object is then stored in a hash table with the unique episode-id as its key. Simultaneously, we create a new episode-id list for each of the events in the episode or append the episode-id if one exists. The computing phase starts once all the episodes have been stored in the hash table and the corresponding episode-id list is built. In the computing phase, we make a single pass over the raw dataset in the database. Each tuple in the raw dataset is an event occurring at a particular time point. For each tuple, we fetch the corresponding episode list. The episode list contains a list of episode-ids of episodes where this event participates. For each episode-id in the list, we fetch the corresponding episode object from the episodes hash table. We now check whether the event transaction time falls in the range of the episode time interval [*StartTime*, *EndTime*]. If it falls in this interval, we ungroup the day and time of the transaction time of the event in the raw data set to compute the weekday value. If the time component of the transaction time falls in the interval range of the episode start time and end time, we use the weekday component of the transaction time in the raw data set tuple and increment the corresponding week day support array index for this event in the episode by one and update the hash table of episodes. This process is repeated for each episode in the episode list. Once all the episodes corresponding to this device are updated, we fetch the next tuple from the raw data set and repeat the entire process of computing support.

In this way, we make a single scan over the raw data set and update the support of corresponding events in each episode for the specified granularity. The process of computing the support of an event is not straightforward since the episodes discovered by Hybrid-Apriori do not contain any information for the next granularity e.g., an episode for daily periodicity contains only the start time and the end time. As Hybrid-Apriori is a SQL based approach, the episodes discovered have the first day of the current month and year prefixed to the start and the end time. This is done so that the aggregate operators of SQL such as the least and the greatest operator can be used in the episode discovery process. On the other hand, the raw data contains all the information pertaining to the date of the transaction i.e. the time, the day, the month and the year. In order to compare the timings of the episode with the transaction time of the event in the raw data set, we extract the time component from the episode and the event and prefix the Java epoch date to this and then check whether the event time falls within the range of the episode timings. If it does fall, we extract the weekday from the transaction date of the event in the raw data set and update the appropriate support index of this event in the episode.

Once the computing phase is over, validation is carried out for all the episodes to distinguish the valid episodes from the false positives.

4.1 Implementation of the Partitioned Approach

The partitioned approach partitions the set of episodes and the raw data before starting the validation process. Partitioning of episodes is done using two approaches: Uniform distribution and Non-Uniform distribution.

The uniform distribution approach partitions the set of episodes on fixed time units determined by the number of partitions. It takes advantage of the partitioning feature of Oracle database. In Oracle8 and beyond, partitions are themselves segments and they can be stored and managed independently of one another. Segments are comprised of extents, and extents are comprised of blocks, with blocks being the smallest unit of space actually managed by the database. The non-partitioned table is also a segment. The EXCHANGE PARTITION operation used here changes the identity of the partition to become that of a non-partitioned table, and then changes the identity of the non-partitioned table to become that of a segment. This is done entirely within the Oracle data dictionary. The identities of the segments themselves do not change. Nor does any of the data within the segments get moved. Instead, the identity of segment A (formerly a partition) is changed from SEGMENT_TYPE = 'TABLE PARTITION' to 'TABLE', and the identity of segment B (formerly a non-partitioned table) is changed from SEGMENT_TYPE = 'TABLE' to 'TABLE PARTITION'. This kind of identity switching is valid if and only if the partition and the table have the same "logical shape" (i.e. same columns, data types, data-integrity constraints, etc). There are many types of partitioning methods available. In our case, we use RANGE partitioning. This type of partitioning creates partitions based on the "range of column" values. Each partition is defined by a "Partition Bound" (non inclusive) that basically limits the scope of partition.

We partition the table of episodes in the underlying database on their start time. The sample queries shown below demonstrate the process of partitioning the table of episodes.

1. *ALTER SESSION set nls_date_format = 'dd-mon-yy hh24:mi:ss'*
2. *ALTER TABLE F_22 add (dtstarttimeNum number)*
3. *UPDATE F_22*

   set dtstarttimeNum =

   lpad(TO_NUMBER(TO_CHAR(to_date(dtstarttime,'dd-mon-yy hh24:mi:ss'), 'HH24miss')),6,'0')
4. *CREATE TABLE F_22_P*

(ID number, item1 varchar2(40) , item2 varchar2(40) ,

dtstarttime date, dtendTime date, confidence number,

dtstarttimenum number) NOLOGGING

partition by range (dtstarttimenum)

(Partition part6_4 values less than (240000))

5. *ALTER TABLE F_22_P EXCHANGE PARTITION part6_4 with TABLE F_22*
6. *ALTER TABLE F_22_P SPLIT PARTITION part6_4 AT (60000) INTO (partition part6_1, partition part6_4)*
7. *ALTER TABLE F_22_P SPLIT PARTITION part6_4 AT (120000) INTO (partition part6_2, partition part6_4)*
8. *ALTER TABLE F_22_P SPLIT PARTITION part6_4 AT (180000) INTO (partition part6_3, partition part6_4)*

The above set of SQL statements are used for partitioning based on uniform distribution with number of partitions equal to four. All the episodes which start before 6 am belong to the first partition while episodes starting between 6 am and noon are assigned the second partition and so on. Line number 1 sets the date format to our context. Line number 2 and 3 are necessary because the episodes discovered by Hybrid-Apriori do not contain the higher granularity information. The only valid component is the time component. Therefore, line number 3 converts the time component of the start time of the episode to a number. For example, an episode with start time "1-Oct-2005 13:07:56" will be converted to a number 1307056 and stored in the *dtstarttimenum* column of the episode table *F_22*. Range partitioning is now done on this number (i.e., *dtstarttimenum* column as shown in line number 4). The statement in line number 5 swaps the F_22_p table's partition part6_4 with the table F_22. The contents of F_22 (episode table of size 2) are now in the part6_4 partition of F_22_p, and the F_22 table is empty. Since this operation merely changes the data dictionary and does not physically move data, it does not generate redo and is extremely quick. Next, as shown in line number 6, we split this single partition, starting with the lowest boundary 6000 identified as partition part6_1. This SQL statement creates a new partition called part6_1 and moves the rows with a *dtstarttimenum* value of less than 60000 into partition part6_1 from part6_4. Since the table is defined as NOLOGGING, this doesn't generate much redo. After this operation, the partition part6_4 contains data for the partitions other than part6_1. We repeat this process for partition part6_2 and partition part6_3 as shown in line number 7 and 8. At the end partition6_4 contains data pertaining to its partition i.e., episodes in the time interval [18:00:00, 23:59:59]. This process is repeated for episodes of all sizes. The partitioned approach to validate can now access each partition in isolation by querying the episodes and the corresponding raw data by their partition names.

Partitioning based on uniform distribution does not distribute equal workload for skewed episodes – episodes having their time interval [start time, end time] more or less similar. In order to deal with such episodes we implement a partitioned approach based on the number of episodes discovered. For all episodes with the same episode size, the total number of episodes is divided by the number of partitions to get the maximum number of episodes in a partition (mk). We then fetch the start time of the episode with episode-id equal to mk This process is repeated for episode of all sizes and the maximum of the start time from these is taken as the partition time for the partition.

Once we are done with both the partitioning approaches, we examine the results of partitioning to determine the approach that distributes the given set of episodes in the best manner – achieving almost equal load balance among all partitions. We then start the partition or the parallel approach to validate the frequent episodes by selecting the best between the two approaches to partitioning.

The partitioned approach deals with three kinds of episodes. The first is a normal episode, the second is spanning episode and the third is the wrapping episode. The normal episode is the one that start and end in a single partition. The wrapping episode is the one which spans multiple periods due to the inherent time wrap property while the spanning episodes is the one which spans multiple partitions and it depends on the number of partitions, the partition time point and the time interval of the episode. The normal episodes are validated in the same partition but the wrapping and the spanning episodes need to be carried forward with their partial support count computed in the current partition to the consecutive partition. In order to differentiate the normal episodes from the wrapping and the spanning episodes we use the episode type attribute of the episode object as mentioned in the design section of this thesis. In the build phase of each partition we identified the episodes type based on their start time and the current partition times. This information is now used in the validation phase. In the validation phase, the normal episodes are distinguished as valid episodes or false positives while the spanning episodes are validated only if their end time is less than the partition end time else they are carried forward to the next partition with their partial support. The wrapping episodes are different from the spanning episodes in the sense that they are always validated in the last partition. The reason being wrapping episodes may start or end in any partition or they may span across multiple partitions but since we start the validation process from the first partition we cannot compute the final cumulative support until we have scanned the entire set of raw data events i.e. reached the last partition.

The *readPatternsWithPartition* class implements the partition approach to validate the frequent episodes. It is a sequential approach where each partition is validated in sequence. At the end of each partition it carries forward the spanning and the wrapping episodes with their partial support to the next partition and appends to the hash table of episodes populated for that partition. Once the validation phase for the last partition starts, it validates the wrapping episodes along with the other episodes.

4.2 Implementation of the Parallel Approach

The parallel approach begins with the partitioning of the episodes to validate. The number of nodes available for computation decides the number of partitions to be done. The number of partitions can also be computed based on the available memory. All the participating nodes carry out the build and the computation phase in parallel. In the validation phase, the normal episodes are validated in parallel while the support count of spanning and wrapping episodes is sent to a central node where the partial support count from each node has to be merged before they van be validated. The merge process starts only after it receives partial support count from all the nodes participating in the validation process. Once the merging is done, the process of validating the episodes is similar to naïve approach.

4.3 Selecting Episodes spanning multiple partitions

Partitioning of episode using the partition feature of Oracle database does not work in this approach since the partitions are to be validated in parallel. Hence we need duplicate the spanning episodes in all partitions and send it to the nodes before the process of validation can start.

Each server receives the information regarding the partition it has to process and the corresponding partition values. In order to select the spanning episodes, we take a SQL based approach. The sample SQL demonstrates how a server responsible for processing partition number 2 would select the spanning episodes belonging to its partition from a given set of episodes.

```
SELECT  *  FROM  EpisodeTable f
WHERE
(                    /*EP STARTING IN A PARTITION*/
to_char(f.dtstarttime,'hh24:mi:ss') < to_char(f.dtendtime,'hh24:mi:ss')
AND    to_char(f.dtstarttime,'hh24:mi:ss')>='prevPartitionTime' AND
       to_char(f.dtstarttime,'hh24:mi:ss')<'currPartitionTime'
)
OR                   /*EP ENDING IN A PARTITION */
(      to_char(f.dtstarttime,'hh24:mi:ss') < to_char(f.dtendtime,'hh24:mi:ss')
9.    AND to_char(f.dtendtime,'hh24:mi:ss')>='prevPartitionTime' AND
          to_char(f.dtendtime,'hh24:mi:ss')<='currPartitionTime'
10.   )
11.   OR                 /*EP SPANNING IN A PARTITION */
12.   (    to_char(f.dtstarttime,'hh24:mi:ss') < to_char(f.dtendtime,'hh24:mi:ss')
13.   AND to_char(f.dtstarttime,'hh24:mi:ss')<'currPartitionTime' AND
          to_char(f.dtendtime,'hh24:mi:ss')>='currPartitionTime'
14.   )
15.   OR                 /*TIME WRAP EPISODES DUE TO FOLDING*/
16.   (
17.   to_char(f.dtstarttime,'hh24:mi:ss') > to_char(f.dtendtime,'hh24:mi:ss')
18.   AND   to_char(f.dtstarttime,'hh24:mi:ss')>='prevPartitionTime' AND
          to_char(f.dtstarttime,'hh24:mi:ss')<'currPartitionTime'
19.   )
20.   OR
21.   (     to_char(f.dtstarttime,'hh24:mi:ss')  >  to_char(f.dtendtime,'hh24:mi:ss')
22.   AND to_char(f.dtendtime,'hh24:mi:ss')>='prevPartitionTime' AND
          to_char(f.dtendtime,'hh24:mi:ss')<='currPartitionTime'
23.   )
24.   OR
25.   (    to_char(f.dtstarttime,'hh24:mi:ss')  >  to_char(f.dtendtime,'hh24:mi:ss')
26.   AND to_char(f.dtendtime,'hh24:mi:ss')>'currPartitionTime' AND
          to_char(f.dtstarttime,'hh24:mi:ss')>='prevPartitionTime' AND
          to_char(f.dtstarttime,'hh24:mi:ss')<='240000'
```

```
27.     )
```

Line number 1 selects all the episodes of size two from the database. Line numbers 4 to 14 deals with normal episodes and spanning episodes while line numbers 15 to 27 deals with folding episodes which by default span at least two partitions. Line number 4 to 6 selects normal and spanning episodes that start in the current partition. Line number 7 to 10 selects episode that end in current partition. Line number 11 to 14 selects episodes that neither starts nor ends in the current partition but still span across the current partition. Line number 15 to 27 selects all the wrapping episodes. The conditions checked here are similar to the ones for normal and wrapping episodes. To summarize, for each partition we look for episodes that start, end or span across this partition.

The parallel approach to the validation of episodes is a distributed approach which uses the Java Remote method invocation framework by Sun Microsystems. RMI capability in Java supports calls to remote procedures. It allows a thread in one JVM to invoke (call) a method in an object in another JVM that is perhaps on a different physical machine. A new thread is created in the other (remote) JVM to execute the called method. Parameters to the remote method and the method's return result, if any, are passed from one JVM to the other using object serialization over the network.

4.4 RMI Architecture for parallel approach

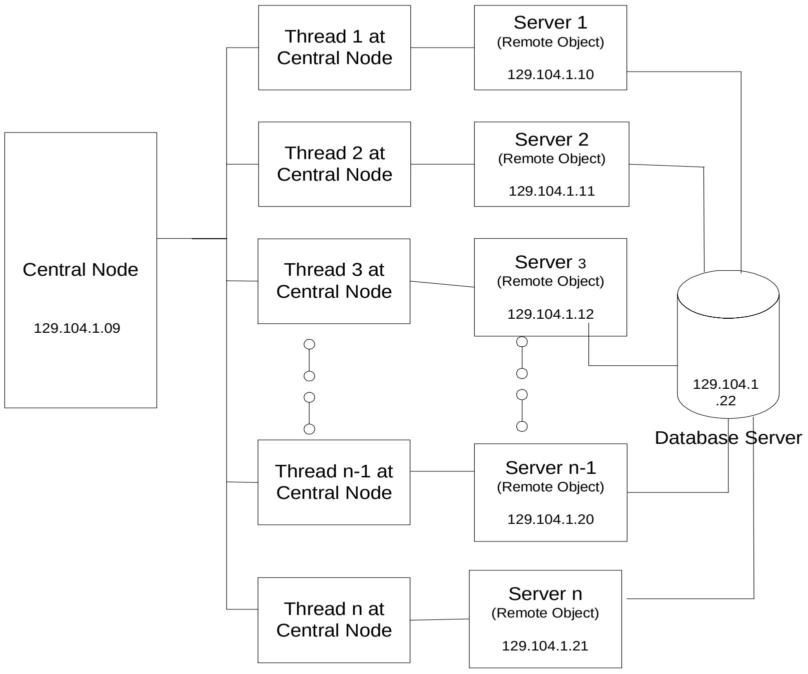

Central Node
129.104.1.09
Thread 1 at Central Node
Thread 2 at Central Node
Thread 3 at Central Node
Thread n-1 at Central Node
Thread n at Central Node
Server 1 (Remote Object) 129.104.1.10
Server 2 (Remote Object) 129.104.1.11
Server 3 (Remote Object) 129.104.1.12
Server n-1 (Remote Object) 129.104.1.20
Server n (Remote Object) 129.104.1.21
129.104.1.22
Database Server

Figure 16 Architecture for the Parallel Approach

As shown above, the central node is responsible for allocating the workload to each node available for parallel computation and validation of the episodes. The central node reads a configuration file that contains all the information related to the remote objects. This file contains the number of nodes available for processing, the uniform resource locator (URL) and the port number of server. The number of partitions created is equal to the number of nodes available. Based on the number of partitions, database of episodes is scanned to generate the partition start and end time for each partition. This is done for non-uniform distribution approach. For uniform distribution of episodes, we simply generate timestamps based on the number of partitions. Once the partition time points are generated, the central node generates a thread for each node and passes it the URL, port number and the partition information. This thread is responsible for invoking the *validatePartition* method of the remote object by sending the partition information to the server on the remote node. The remote servers upon receiving the partition information dynamically create the SQL for selecting the relevant episodes from the database of episodes. The entire process of validating the episodes as explained in the naïve approach – build phase, support counting phase and the validation of normal episode now starts at each node in parallel. The spanning and the wrapping episodes with their partial support count are sent back the client thread. The client thread sends the results back to the central node that waits for the results to arrive from all the nodes. Once all the results arrive the merge phase starts at the central node.

4.5 Merge Phase at the central node

Once all the spanning episodes with their partial support counts from all nodes are received at the central node, the merge phase starts. In this phase, we merge the partial support counts of an episode to get their cumulative support count. The pseudo code for this is shown below. The hash table containing the partial support count of spanning episodes computed at each node is termed as local hash table (localHT) and the hash table with the cumulative support count is termed as global hash table (globalHT). Once the merge phase is over, we start the validation phase that is same as the validation phase in Naïve approach.

```
1.  Create an global hash table 'globalHT' for merging the spanning episodes from all
    nodes
2.  For each local hash table 'localHT' of spanning episodes
3.  Fetch the episode-key Ki of a spanning episode
4.  If the key Ki does not exist in the globatHT
5.          Fetch the corresponding spanning episode from the localHT
6.          Insert the episode-key and the spanning episode into the globatHT
7.  Else
8.          Fetch the spanning episode corresponding to this episode-key Ki from the
            localHT
9.          Fetch the spanning episode corresponding to this episode-key Ki from the
            globalHT
10.         For each event in the spanning episode of localHT
11.         Update the support count of the event in the spanning episode in the
            globalHT for the given granularity
```

Once the merge phase is over we have the cumulative support of the spanning episodes. The validation phase now distinguishes the false positives from the valid episodes.

4.6 How Java RMI works for the parallel approach

The following table displays the major classes in our implementation of the parallel approach with their corresponding purpose.

Table 8 Parallel Approach – Implementation overview

Purpose	**Class Name**
1) Interface for remote objects	ValidatePatternInterface
2) Remote Object responsible for validation	RemoteObjectValidate
3) Serve responsible to serve the clients and invoke the remote objects	ValidateServer
4) Client responsible to send the work load to the remote object	ValidateClientThread
5) Responsible for validating the episodes	ValidateWorker
6) Central node responsible for partitioning and creating threads for each remote object available for validation, merging and validating the spanning episodes	ValidateClient

Here the *remoteObjectValidate* is our remote object that implements a *validatePatternInterface* interface containing a *validatePartition* method responsible to validate the partition of episode assigned to it. This method is a remote method which can be invoked by the clients to validate a set of frequent episodes. The remote object registers itself with the RMI server *validateServer* for the clients to locate it. The *validateClient* class that acts as a central node is responsible for creating multiple threads, one each for a server responsible to do the majority of computing. The central node creates a thread each for the remote servers. The local client thread *validateClientThread* passes a *validateInfo* object as a parameter to the RMI server. The *validateInfo* object is an information triplet (IP address, port number, partition to process) indicating the partition to process and the server responsible for that partition.

The RMI server receives this information and passes it to the remote object for further processing. The remote object for the given work load does all the processing and returns the results back to the client thread which in turn passes it to the central node. Thus the client thread lets the remote object to have work performed on its behalf (computing support of the episodes, validating the normal episodes and returning the spanning episodes yet to be validated) and returns the results to the central node for further validation. After receiving the results from all the client threads starts, the central node starts the merge phase. In the merge phase, the partial support counts of each spanning episode are summed up to get their cumulative support.

The table below summarizes the sequence of steps in the parallel approach discussed earlier.

Table 9 Sequence of steps in the parallel approach

Steps	**Validate Client / Central Node**	**Validate Client thread/ Multiple Threads at Central Node**	**Remote object validate / Multiple Remote Objects/Servers**
1	Partition the episodes to be validated based on the number of servers available.		
2	Create a thread each for all the servers and send each thread one partition information and a reference of central node for callback when the partition results are available from the server		
3	Listen to the all the threads to receive the result set		
4		Receive the partition information from the central node – validate client	
5		Look up the remote object/server and send it the partition information	
6		Wait for the server to return the hash table of spanning episodes and the result of normal episodes	
7			Receive the partition information from the validate client thread
8			Compute the support and validate the normal episodes
9			Send the spanning episodes with their support count and the result of normal episodes
10		After receiving the spanning episodes with their support count, return the result back to the central node	
11	Start merging when results from all servers have arrived.		
12	Merge the partial support counts and validate the spanning episodes		

4.7 Summary

This chapter discussed the implementation process to validate frequent episodes. We propose three approaches to validation and discuss the implementation requirements of each followed by the implementation issues associated with each approach. We also discuss the advantages and limitations of each approach. In the next chapter, we demonstrate the performance and scalability aspects of each approach through extensive experiments.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we discuss the experimental results for our three approaches to validate frequent episodes. The objective is to test the performance and scalability of these algorithms.

Experimental Setup: Experiments for the naïve and partitioned approach were run on machine with id=1. Parallel approach with three nodes used all the three machines.

Table 10 Experimental set up

Id	1	2	3
Memory in GB	1	2	2
Linux type	i386 GNU/Linux	i386 GNU/Linux	i386 GNU/Linux
Version	Red Hat Linux 3.2.3-53	Red Hat Linux 3.2.3-52	Red Hat Linux 3.2.3-53
Database	Oracle 10G	-	-

Dataset: A five-month synthetic data is used to test the correctness and the performance of the validation approaches. We extract data set for one week, one month, two months and four months from this data set and run the Naïve approach to study the change in response time with respect to increase in the data set size and hence the number of episodes to validate. The partitioned approach and the parallel approach are run for the largest data set size of five months and their performance is compared with the Naïve approach. For the partition approach, we vary the number of partitions and analyze variations in response time with the increase in the number of partitions.

The final response time is calculated by taking the average of five runs.

5.1 Performance of Naive approach for daily periodicity

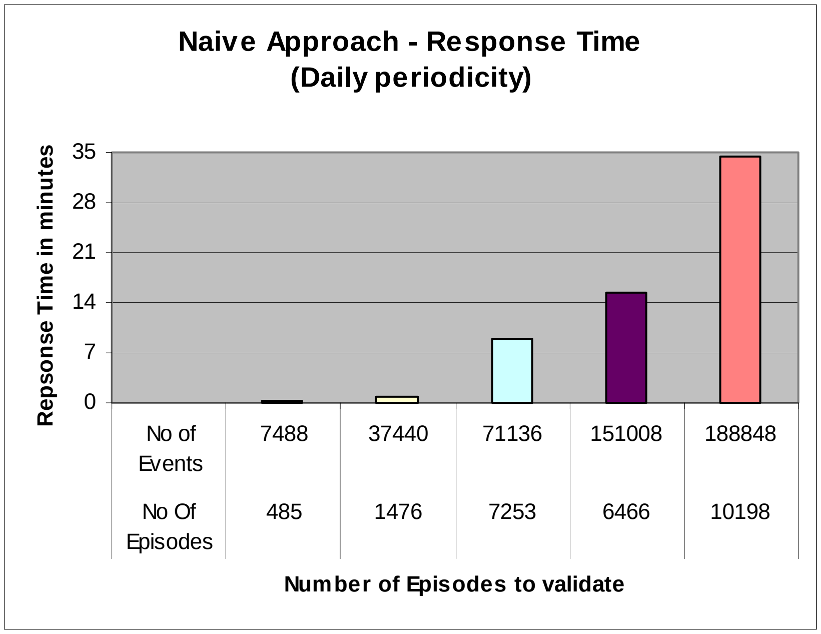


Figure 17 Performance of Naïve Approach with different synthetic data sets

The graph above shows the response time of Naive approach for different sizes of synthetic data sets as displayed in the table below.

Table 11 Synthetic data set

Type of Data set	Synthetic data	
Number of Days in data set	**Number of Events**	**Number of Episodes**
7	7488	485
30	37440	1476
60	71136	7253
120	151008	6466
150	188848	10198

The experiment is run for a one week, one month, two months, four months and five month data set. As the graph demonstrates, the time taken by naive approach depends on the size of the raw data set and the number of episodes to validate. For the four month data set, we observe that the number of episodes to validate is less than that of the two month data set but the two month data set is almost double in size and hence the time taken to validate a four month data set having a lesser number of episode is still more than the time taken to validate a two month data set.

5.2 Comparison of response time of partitioned approach for daily periodicity

Effect of increase in number of partitions on response time in partitioned approach for daily periodicity (five month synthetic dataset)

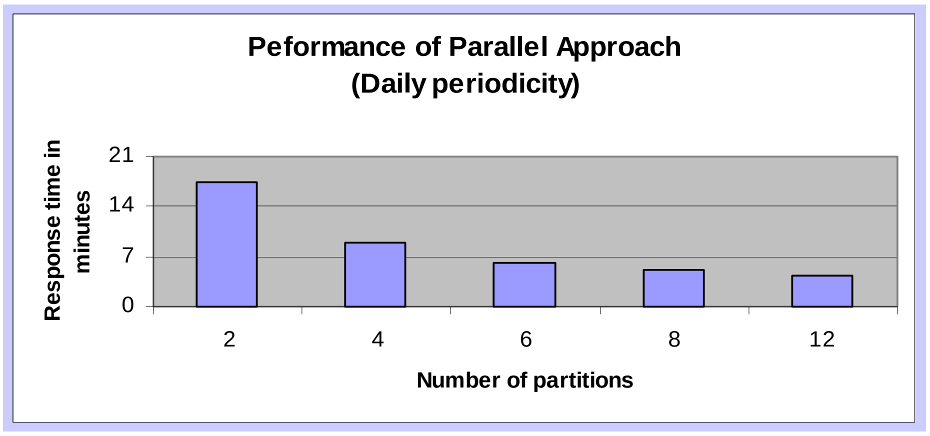


Figure 18 Performance of Parallel Approach for synthetic data set

The above graph demonstrates the effect of memory on the response time for a synthetic data set worth five months. The details of this data set are described in Table 11. As we increase the number of partitions the number of episodes required to be validated in a partition decreases but the available memory remains the same. As a result significant reduction in response time is seen with the increase in the number of partitions. But we can also observe from the following table that the percentage reduction achieved is not proportional to the increase in the number of partitions, which implies that the benefit decreases as the number of partitions increases.

Table 12 Evaluation of Partitioned Approach

Base Case	**Partitions = 2**	
	Response time in minutes	17.34
Number of Partitions	**% Improvement in response time**	**Actual Response time**
4	52%	9.01
6	35%	6.12
8	29%	5.04
12	24%	4.24

Effect of increase in number of partitions on response time in partitioned approach for daily periodicity (six month MavHome data set)

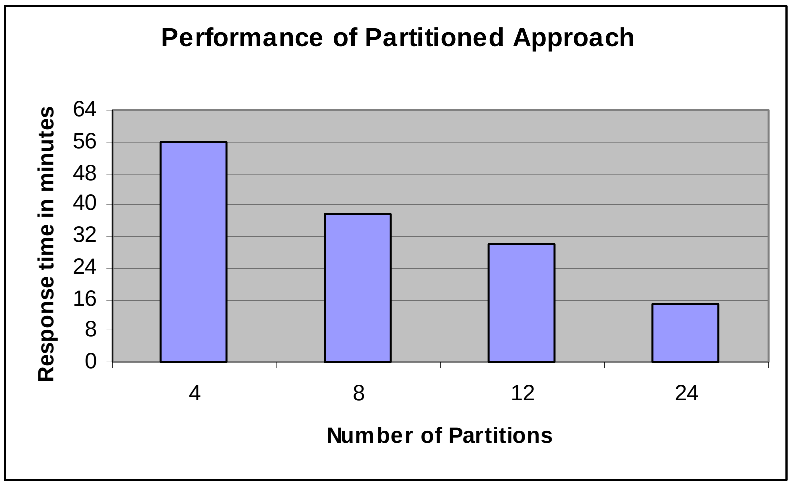


Figure 19 Performance of Partitioned Approach for daily periodicity

The graph above demonstrates the response time of partitioned approach for experiments run on MavHome dataset for six months in table 14 for daily periodicity. As seen from the graph above, the partitioned approach performs well as we increase the partition. For twenty-four partitions, it validates all the episodes in less than fifteen minutes. The primary reason for this reduction is the reduced size of the hash table of episode. As a result, the number of episodes to validate per event instance in the raw dataset reduces significantly.

Table 13 Partitioned approach - percentage improvement in response time

Base Case	**Partitions = 4**	
	Response time in minutes	56
Number of Partitions	**% Improvement in response time**	**Actual Response time**
8	67.06%	37.55
12	53.41%	29.91
24	26.32%	14.74

5.3 Performance of Parallel Approach for daily periodicity

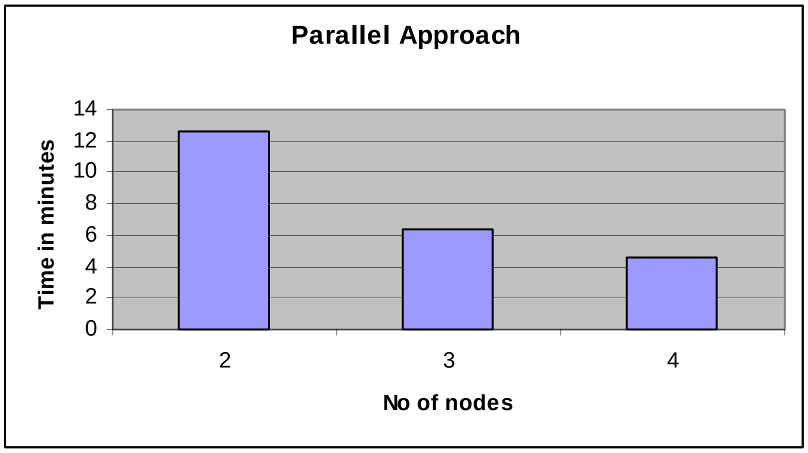


Figure 20 Performance of Parallel approach for synthetic data set

The above graph demonstrates the experiments for a five-month dataset run in parallel with two nodes, three nodes and four nodes. This data set contains 188848 events and Hybrid-Apriori discovers 10198 episodes with maximum episode size of four. The response time decreases with the increase in the number of nodes available for validating. But as seen from the table below the reduction in response time is not proportional to the increase in number of nodes.

Table 14 Parallel Approach - percentage improvement in response time

		Actual Response time
Base Case	Nodes=2	12.57
Number of Partitions/Nodes	**% Improvement in response time**	
3	50.84%	6.39
4	34.37%	4.32

Table 15 MavHome data set

Dataset	MavHome	Number of Days=180
Events	217275	
SIDs	1634	
Episodes	12956	
Periodicity	Daily	
Algorithm	**Response Time (minutes)**	**Logarithmic Response time**
SID	2.60	0.42
HA	3.29	0.52
Naïve	652.20	2.81
Partition		
No. of Partitions		
4	56.00	1.75
8	37.55	1.57
12	29.91	1.48
24	14.74	1.17
Status of Episodes		
Valid	58469	
Invalid	32223	

Dataset: The following experiments were conducted on the MavHome dataset worth six months. This dataset has been collected from MavHome and it consists of 217,275 unique event instances with 145 unique events in it. SID discovered 1634 significant intervals and Hybrid Apriori found 12959 episodes.

5.4 Performance comparison of each approach for daily periodicity

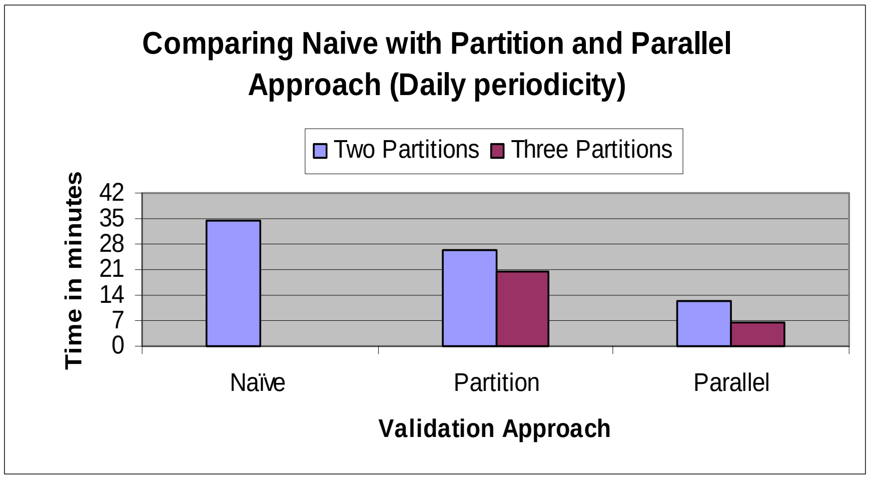


Figure 21 Performance of all three validation approaches

The above graph shows the response time of each approach for five months synthetic data. This data set contains 188848 events and Hybrid-Apriori discovers 10198 episodes with the maximum episode size of four. The partition and parallel approach are initially run for two partitions. For parallel approach, the number of partitions is the number of nodes validating in parallel. As the graph demonstrates time taken by parallel approach is the least while that by naïve approach is the most. As we increase the partitions or the number of nodes, the time to validate the episodes reduces further.

The following graph demonstrates the response time of SID, Hybrid-Apriori, Naïve and Partitioned algorithm with number of partitions equal to 24. The experiments were run for daily periodicity with the MavHome dataset for six months whose details are shown in the table 13 above. This particular graph converts the response time in minutes to the logarithmic scale as the difference between SID and Naïve is very large.

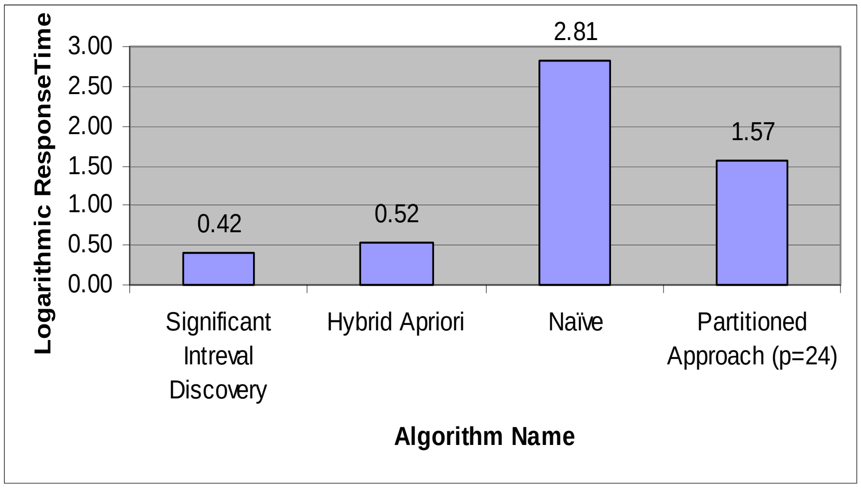


Figure 22 Performance Comparison of all phases in Episode Discovery process

As seen from the above graph, the naïve approach to validate takes significant time to validate the episodes. It takes approximately eleven hours to validate. This is because we store and validate all the episodes in the main memory at the same time while for partitioned approach with number of partitions set to twenty-four, each partition is stored and validated in isolation and significant difference in response time is seen. The reduction in response time is because partitioning reduces the number of episodes to validate and corresponding raw data set but the available memory remains the same.

5.5 Performance of Naïve Approach for Weekly Periodicity

The following graph shows the response time of Naïve approach for synthetic data sets of 9 weeks and 22 weeks. The nine weeks data set consisted of 74880 events and 2949 episodes while the twenty-two weeks data set consisted of 217275 events and Hybrid Apriori discovered 11108 episodes. As observed for daily periodicity, the response time for weekly periodicity also increases with the increase in the size of the raw data set and the number of episodes to validate.

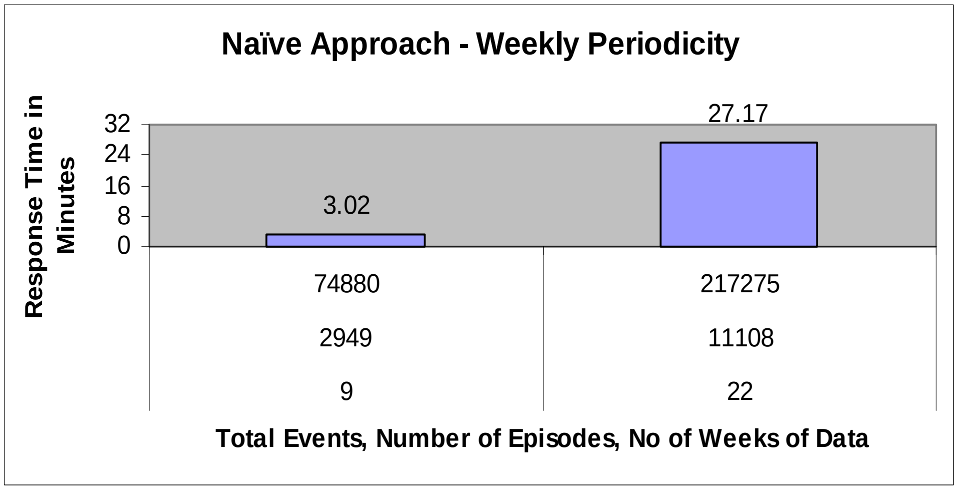


Figure 23 Performance of Naïve Approach for Weekly Periodicity

5.6 Configuration File

The following table displays the parameters used in episode discovery process by SID, Hybrid-Apriori and the three approaches to validation.

Table 16 Configuration Parameters

Configuration Parameters	**Description**
Parameters for Significant Interval Discovery	
Measure	Metrics to be used to generate intervals It could be interval length, and/or interval confidence
Req_Confidence	Desired confidence for the intervals generated. This value is between 0 and 1
Granularity	It specifies the granularity of time for which the information is collected It may be seconds, minutes or hourly.
Interval_Length	Specifies the interval length that the user is interested in
Number_of_Days	Number of days is the total number of days for which data is collected
Approach_Number	Type of SID to use. It could be SID[1], SID[n-1], or SID[n-2]
Time_Series	Type of data to operate on. It could be time series or numerical
Input_File	Input file for numerical data
Period	Periodicity could be daily or weekly
Parameters for Hybrid Apriori	
Interval_Semantics	Type of semantics to discover episodes. It could be semantics start or semantics end
Sequential_Window	Window parameter for intervals to form an episode
Pattern_Confidence	Confidence parameter to prune the episodes discovered
StopLevel	Upper bound on the size of episode discovered Terminating condition for the episode discovery algorithm
Parameters for partitioned approach	
TypeOfPartition	Partitioning method
NoOfPartitions	Number of partitions to be done
Parameters for parallel approach	
NoOfProcessors	Number of nodes available for the parallel approach
URLforRemoteObject	URL for the remote object
PortNoForRemoteObject	Port number on which the Server listens for client requests
Parameters for database	
RawDataSource	Data source for the algorithm
NameOfDatabase	Database name
NameOfMachine	Machine name where it is located
NameOfUser	User name
PasswordOfUser	Password
DBPort	Port number of the database
Other Parameters	
Complete_Debug	Debugging flag
NumberOfThreads	No of threads to be created. Used by SID to generate significant intervals for each device in parallel

5.7 Log files

The validation approach generated log files at the end of each run. It generates logs related to run time information, episode status and device support. The sample snippets of each are shown below:

5.7.1 Log file for Episode Status

```
CreateLog: Starting Logging Tue Sep 06 23:43:33 CDT 2005
Pattern=$948KidsRm1Lamp_OffKitchLght1_Off$ is invalid for weekday=0
Pattern=$948KidsRm1Lamp_OffKitchLght1_Off$ is valid for weekday=1
Pattern=$948KidsRm1Lamp_OffKitchLght1_Off$ is valid for weekday=2
Pattern=$948KidsRm1Lamp_OffKitchLght1_Off$ is valid for weekday=3
Pattern=$948KidsRm1Lamp_OffKitchLght1_Off$ is invalid for weekday=4
Pattern=$948KidsRm1Lamp_OffKitchLght1_Off$ is valid for weekday=5
Pattern=$948KidsRm1Lamp_OffKitchLght1_Off$ is valid for weekday=6
Episode=$948KidsRm1Lamp_OffKitchLght1_Off$ is valid for at least one weekday
```

5.7.2 Log file for device support

```
CreateLog: Starting Logging Tue Sep 06 23:43:33 CDT 2005
###############Next Pattern ##############
Pattern = 948KidsRm1Lamp_OffKitchLght1_Off
StartTime08:22:00,End Time09:22:00
<---Item=KidsRm1Lamp_Off-------->
  Weekday=1, Support=1
  Weekday=2, Support=2

  Weekday=3, Support=1

  Weekday=4, Support=1

  Weekday=5, Support=2

  Weekday=6, Support=2

<---Item=KitchLght1_Off-------->

  Weekday=1, Support=1

  Weekday=2, Support=2

  Weekday=3, Support=2

  Weekday=5, Support=1

  Weekday=6, Support=1
```

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

There is ongoing research on sequence mining of time-series data. We study Hybrid Apriori, an interval based approach to episode discovery that deals with different periodicities in time-series data. Our study identifies the anomaly in the Hybrid Apriori by confirming the false positives in the frequent episodes discovered. The anomaly is due to the folding phase of the algorithm that combines periods in order to compress data.

We propose a main memory based solution to distinguish the false positives from the true frequent episodes. Our algorithm to validate the frequent episodes has several alternatives such as the Naïve approach, the Partitioned approach and the Parallel approach in order to minimize the overhead of validation in the entire episode discovery process and is generalized for different periodicities. The naïve approach validates all the episodes and verifies the correctness, partitioned Approach allowing us to validate large number of episodes by controlling main memory usage and the parallel approach allows using as many processors as possible making it a scalable and performance-oriented approach. The following table summarizes the key features of each approach.

Table 17 Comparison of Validation approaches

Features	Naïve	Partitioned	Parallel
Approach Type	Sequential	Sequential	Parallel
Main memory dependency	Most	Less	Least
Response Time	Good	Very Good	Best
Scalable	Least	More	Most
Dependency on episode distribution	No	Yes	Yes
Support for Data set size	Small	Very Large	Very Large

Extensive experiments have been conducted to demonstrate the performance and scalability of each approach using synthetic and MavHome data sets. The experiments conducted indicate that the naïve approach has a limit on number of episodes it can validate while the partitioned approach, divide and conquer approach, can validate larger set compared to naïve approach. With the increase in number of episodes and increase in maximum size of episodes, the response time of naïve approach also increases. The type of partitioning method used affects the performance of the partitioned approach e.g., for same dataset, partitioning based on uniform distribution gives better performance than partitioning on non-uniform distribution. The number of partitions of episodes to validate affects the performance of the partition approach but the percentage improvement achieved is not proportional to the percentage increase in the number of partitions. This holds true for number of nodes in the parallel approach too.

6.2 Future work

The current implementation of the Naïve approach computes the support count by scanning each event instance, ordered by time, and updating the support of the episodes containing this event. This can be optimized by ordering the raw data set on the events instead of the time and fetching a batch of event instances and modifying all the corresponding episodes for the entire batch. This will significantly improve the response time of the naïve approach.

The response time of the partitioned approach and the parallel approach are dependant on the distribution of the episodes over a given period. The current implementation of algorithm to partition the episodes works well for uniform distribution of episodes but is unable to achieve equal workload for different types of skewed distributions. An efficient algorithm to deal with skewed distribution, which can ensure equal distribution of workload in each partition/node, is required to achieve better response time for the validation algorithm.

REFERENCES

1. Heirman, E., *Using Information-theoretic Principles to Discover Interesting Episodes in a Time-ordered Input Sequence*, Computer Science and Engineering. 2004. The University of Texas at Arlington: Arlington.
2. Agrawal, R. and R. Srikant. *Mining Sequential Patterns*. In *ICDE*95, Taipei, Taiwan, pages 3--14, 1995.
3. Han, J. and M. Kamber, *Data Mining : Concepts and Techniques*. 2001: Morgan Kaufmann Publishers.
4. Mannila, H., H. Toivonen, and I. Verkamo. *Discovering Frequent Episodes in Sequences*. In *KDD95*, Montreal, Canada, pages 210--215, 1995.
5. A.Srinivasan, S.Shrestha, and S.Chakravarthy. *Discovery of Significant Intervals in Sequential Data*. In *1st ADBIS Workshop on Data Mining and Knowledge Discovery*, Tallinn, Estonia, 2005.
6. Han, J., W. Gong, and Y. Yin. *Segment-Wise Periodic Patterns in Time Related Database*. In *KDD98*, New York City, New York, USA: AAAI Press, pages 214--218, 1998.
7. Zaki, M.J., *SPADE: An Efficient Algorithm for Mining Frequent Sequences.* Machine Learning Journal, special issue on Unsupervised Learning. Vol. 42(1/2): p. 31--60, 2000.

8. Zaki, M.J. *Sequence Mining in Categorical Domains: Incorporating Constraints*. In *CIKM2000,* McLean, Virginia, USA, pages 422-429, 2000.

9. Srikant, R. and R. Agrawal. *Mining Sequential Patterns: Generalizations and Performance Improvements*. In *EDBT*96, Avignon, France, pages 3--17, 1996.

10. Cook, D.J., et al. *MavHome: An Agent-Based Smart Home*. In *PerCom03,* Fort Worth, Texas, USA, pages 521--524, 2003.

11. A.Srinivasan, D.Y.Bhatia, and S.Chakravarthy. *Discovery of Interesting Episodes in Sequence Data*. (To Appear) In *SAC06*, Dijon, France, 2006.

BIOGRAPHICAL INFORMATION

Dhawal Bhatia was born on December 21, 1976 in Ahmedabad, India. He received his Bachelor of Science in Mathematics degree from Gujarat University, Gujarat, India in April 1997 and Master of Computer Application degree from Gujarat University, Gujarat, India in July 2000. On graduation he was recruited by the Center for Electronic Governance (CEG) at the Indian Institute of Management, Ahmedabad (IIM-A) where he worked for a period of three years. In his role as a research associate/software developer, he was involved in development of E-Government projects for the State Government and the Central Government of India. He also participated in the cost-benefit analysis of E-Government projects for the World Bank and the Asia Foundation. In fall 2003, he commenced his second graduate program in Computer Science at The University of Texas, Arlington. He received his Master of Science in Computer Science from The University of Texas at Arlington in December 2005. Dhawal's research interests include Decision Support Systems, Prediction Algorithms, Data warehousing, Data Mining and Business Intelligence.