SCALABLE PATTERN MINING ALGORITHMS FOR LARGE SCALE INTERVAL DATA

A DISSERTATION

Submitted in the partial fulfilment of the

requirements for the award of the degree

of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

by

PRAKHAR DHAMA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

ROORKEE - 247667 (INDIA)

May, 2017

DECLARATION

I hereby declare that the work which is being presented in the dissertation entitled **"Scalable Pattern Mining Algorithms for Large Scale Interval Data"** towards the partial fulfilment of the requirements for the award of the degree of **Master of Technology in Computer Science and Engineering** submitted in the Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, Uttarakhand (India) is an authentic record of my own work carried out during the period from July 2016 to May 2017 under the guidance of **Dr. Sateesh K. Peddoju**, Assistant Professor, Department of Computer Science and Engineering, IIT Roorkee.

The matter presented in this dissertation has not been submitted by me for the award of any other degree of this or any other Institute.

Date: **PRAKHAR DHAMA**

Place: Roorkee (15535029)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: **DR. SATEESH K. PEDDOJU**

Place: Roorkee Thesis Supervisor

ACKNOWLEDGEMENT

First and foremost, I would like to extend my heartfelt gratitude to my guide and mentor **Dr. Sateesh K. Peddoju**, Assistant Professor, Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, for his invaluable technical guidance, great innovative ideas and overwhelming moral support during the course of the dissertation. His wisdom, knowledge and commitment to the highest standards inspired and motivated me. He has been very generous in providing the necessary resources to carry out my research. He is an inspiring teacher, a great advisor, and most importantly a nice person. I thank him for his guidance and constant supervision to accomplish the given task within the time constraints. I also thank **Dr. Dhaval Patel** for introducing me the idea of temporal pattern mining and providing me with the experimental setup to carry out my research.

I also express my gratitude to the Department Of Computer Engineering and the entire faculty members, for their teaching and guidance and encouragement. I am also thankful to research scholars in UGPC lab, our classmates and friends for their valuable suggestions and support.

Finally, hearty thanks to my parents and siblings, who encouraged me in good times, and motivated me in the bad times, without which this dissertation would not have been possible.

PRAKHAR DHAMA

(15535029)

ABSTRACT

The problem presented in this thesis is of mining temporal patterns on a large scale interval data. Recently, many real-world data streams comprise of time interval-based event data instead of point-based data. And considerable efforts are going on in finding patterns in these interval-based events that have non-zero duration. The pattern mining algorithms for point-based events are not suitable for temporal data. Also, algorithms based on sequential data are not feasible for obtaining insigth into these interval based events which span across for some duration of time. The relations among the interval based events are complex than simple point-based events, therefore mining patterns on interval-based data is an arduous task. Such interval pattern mining algorithms are highly exponential and so the algorithm must be efficient and effective when dealing with interval based data. Interval Pattern Mining is adequate to express the complex temporal relationships in domains such as medical, multimedia, meteorology and finance where the events' durations play an important role.

The data generated is steadily increasing exponentially and serial pattern mining algorithms are infeasible for such large scale data. There has been some work done on parallel pattern mining for addressing the scalability issue for point-based data. But to the best of my knowledge, all the current pattern mining algorithms on interval data are sequential in nature. Therefore there is an increasing need to develop a pattern mining algorithm that can run in a distributed way. Such algorithm that can utilize several separate computing nodes can achieve high parallelism and scalability required for large scale data. The work carried under this dissertation presents a new algorithm PIMiner to meet above objectives.

The proposed PIMiner algorithm takes into the account both the task parallelism as well as data parallelism to mine temporal patterns on a big interval data. The task parallelism is achieved by identifying various tasks that can run concurrently so as to use all the available cores for simulatenous computing. This kind of in-built task parallelism in the algorithm is notably effective for finding patterns when several cores are available in the same chip. Also, the data parallelism is achieved by incorporating MapReduce technique. The large data is divided into smaller chunks called blocks so that the block level parallelism is attained. The parallel processing on such blocks is followed by combining the partial computation and producing the desired output. This way the proposed PIMiner algorithm can find the temporal patterns in distributed fashion. The experimetal setup required for obtaining such distributed computing is exemplified with the Apache Hadoop distributed framework. We show the way to generate synthetic data suitable for carrying out interval pattern mining. Experimental studies show how the task level parallelism and data level parallelism alone are quite effective when compared to serial interval pattern mining algorithm. Furthermore, we perform the scalability tests when different input parameters effect the algorithm performance. Finally the future work that can be followed for scalable interval pattern mining is presented.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

1

Frequent pattern mining is critical to association rule mining which discovers relevant relations among variables. It is useful in various domains like analysis in financial markets, tracking user activity patterns on website, and more recently sequencing in DNA.

1.1 Motivation

Frequent itemset mining (FIM) problem is defined as: Suppose $I = \{a1, a2, ..., an\}$ is a set of n items, $T =< id, P >$ is a transaction with id as transaction id and P is subset of I. The transaction database D denoted by $\{t1, t2, ..., tm\}$ is a collection of m transactions. Then the support of some itemset Q is equal to the number of transactions containing Q. Formally,

$$Support(Q) = |\{id | Q \subseteq P \wedge (id, P) \in D\}| \tag{1.1}$$

An itemset can be termed frequent if the support of the itemset is greater than or equal to a desired *minimum threshold* value which can be specified by the analyst.

Apriori like algorithms has to scan the entire database multiple times. Also it requires exchange of large number of candidate itemsets. To remove the need for scanning databases multiple times, various techniques of based on FP-growth were proposed using the extended prefix tree structure, which averts the generation candidate itemsets. But still they are not appropriate for large scale dataset where single scan of dataset is quite expensive. Hence there is need to address the scalability problem of interval pattern mining.

1.2 Need for Interval Pattern Mining

Much of the existing work focuses on discovering frequent patterns among point-based time events and not with interval events. Mining temporal patterns is much arduous task compared to simple events because of the complex relation among two interval events [1]. This results in huge search space and exponential increase in possible temporal patterns as the support reduces. For Apriori based algorithms the complex relation among intervals may lead to large number of candidate patterns followed by tedious work of support calculation. On the other hand for the pattern growth algorithm complexity within the long pattern increases complexity.

Application. The various fields where interval pattern mining is crucial includes library management for lending, stocks analysis, relationships arising among different diseases etc. For example, in library we can find that a certain book is always lent along with one particular book. In the medical application based on different symptoms like frequent coughing occurring during the flu, the prediction can be made that fever might be followed.

1.3 Need for Scalable Pattern Mining

In 2013, NSA publicly released a seven page document in which they proclaimed that they processed more than 29 petabytes of data every day that year. Facebook stores and analyses over 500+ terabytes of data every day.[1]

With this exponential growth in data volume, it has become infeasible to perform mining on one system. The scalable distributed algorithms can solve these problems dealing with big data. Parallel mining can be easily achieved once the ample storage and numerous computing resources is made available in distributed fashion. To the best of my knowledge, neither the task parallelism, nor the data parallelism has been addressed for interval pattern mining. In this work, techniques for incorporating both of them is presented.

[1]Report at `www.theguardian.com/commentisfree/2013/aug/13/nsa-internet-traffic-surveillance`

1.4 Problem Statement

To develop a scalable pattern mining algorithm for large scale interval based data that can run in distributed means on multiple computing nodes using big data technologies. Also depicting the mechanism to store, maintain and process the data in distributed way. Furthermore incorporating both task and data parallelism techniques in the computation which are yet unaddressed with the traditional computing techniques.

1.5 Report Organization

This report demonstrates procedure of pattern mining on large scale interval data. The report contains four chapters. The first chapter introduces the topic pattern mining. The other chapters are in the report are as follows:

- Chapter 2 presents the related work carried on addressing various relationships among different types of events. It also manifests the techniques for pattern mining on interval based data. It finally presents the work done on distributed sequential mining till date.
- Chapter 3 gives the proposed model. It describes the details of the input temporal dataset and the representation used for expressing temporal relationships.
- Chapter 4 demonstrates the experimental setup for carrying out the proposed model with details of framework used and system details.
- Chapter 5 exhibits the experimental results with the performance as well as scalability tests of the proposed algorithm used for interval mining. It also depicts the details of generating synthetic datasets and results on both real and synthetic datasets.
- Chapter 6 contains the conclusion of the conducted work and the future scope on scalable pattern mining on large scale interval data.

BACKGROUND AND RELATED WORK

2

Mining Algorithms, of late, have focused only on finding frequent patterns from immediate events i.e. events with zero duration. In real world, various events, that instead of being immediate, persist for some time period. The proposed sequential algorithms for time point cannot be applied to these sequence of events having start and finish time. The immediate events can only find sequence of pattern like fever followed by indigestion followed by vertigo. However, this sequence of pattern is not appropriate to evince the complex relations in medical field or financial market where duration of events plays a crucial role. Similarly power meter in house that logs interval based data i.e. when each appliance is on or off.

2.1 Interval Events

A temporal database can handle data with time. It stores all the interval sequences in which each interval denoting an interval-based event can have starting and finish times. Multiple sequences are present in the database each with such interval-based events which can be present in any fashion including overlapping each other.

TABLE 2.1: Database with Interval Sequences

SID	event symbol	start time	finish time	interval duration
1	A	1	4	3
	B	3	6	3
	C	8	12	4
	D	8	12	4
2	B	3	6	3
	C	8	12	4
	D	8	12	4
3	C	6	8	2
	D	6	8	2
4	A	1	3	2
	C	6	8	2
	D	6	8	2
	E	12	14	1

2.1.1 Interval Sequence Database

An interval sequence is a collection of several intervals. Table 2.1 shows the example database with four interval sequences 1, 2 3 and 4 as sequence id containing subset of four interval events A, B, C and D. Suppose with minimum support 3, it can be observed that temporal pattern (C equals D), explained later, is classified as frequent because the support of this pattern is 4 present in all the sequences. Similarly the support of pattern (A overlaps B) is 1 as it is present only in first sequence and therefore is not frequent.

2.1.2 Temporal Relations

All of the work related to temporal pattern mining is based on the 13 relations among temporal events given in original James F. Allen's work [1]. These various relations between two interval events say X and Y is depicted in Table 2.2. The support of a temporal pattern, which is a composite of two events with a relation, is the number of interval sequences in database in which it occurs. If the support is greater than minimum threshold it is a frequent temporal pattern. The length of temporal pattern is the number of events in the pattern.

TABLE 2.2: Relations among Temporal Events

Relation	Symbol	Symbol for Inverse	Pictorial Example
X before Y	$<$	$>$	XXX YYY
X equal Y	=	=	XXX YYY
X meets Y	m	mi	XXXYYY
X overlaps Y	o	oi	XXX YYY
X during Y	d	di	XXX YYYYY
X starts Y	s	si	XXX YYYYY
X finishes Y	f	fi	XXX YYYYY

The Allen 13 relations can be reduced to 7 as expect equal relation 6 relations are just inverse of other 6, so identifying one set implicitly expresses the rest. For example, the relation "A before B" implicitly implies the inverse relation "B after A".

Two simple representations, endpoint [21] and endtime [3] can be used for making temporal pattern mining more feasible. Endpoint representation in a sequence uses the arrangements of endpoints to express an interval sequence. Endtime representation includes time information also to describe interval sequence losslessly. For example the endpoint representation of the first interval sequence in Table 2.1 with events A, B, C, and D is $A^+B^+A^-B^-(C^+D^+)(C^-D^-)$. In endtime along with starting and finishing point time of occurrence is also stored. Later in Chapter 3, we illustrate the use of endtime representation in our proposed algorithm model.

2.2 Representation Methods

Almost all the studies on interval pattern mining are based on Allen's relations [1]. An appropriate representation of the temporal pattern is crucial for further mining techniques. In this section we discuss various representations used for same.

2.2.1 *Hierarchical Representation*

One of the earliest representation to express temporal pattern was hierarchical as proposed by Fu and Kam [7]. The temporal pattern was formed by combining frequent temporal pattern to original long pattern. However this relationship was ambiguous. Since same relation can be mapped into different temporal pattern based on the candidate selection.

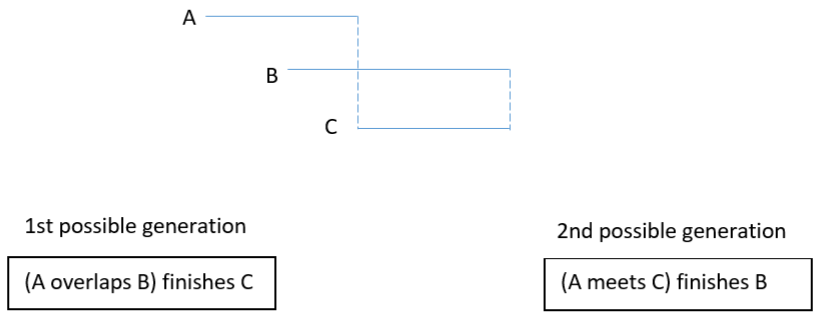


FIGURE 2.1: Ambiguity in Hierarchical Representation

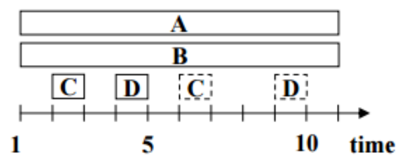
For example consider three temporal events A, B and C in Figure 2.1. This ambiguity arises because of the order of pattern generation in first case, (A overlaps B) pattern is frequent and C is identified as the candidate for further pattern generation while in the second (A meets C) pattern is identified as frequent followed by B as a candidate.

2.2.2 *Relation Matrix*

The ambiguity was addressed by Hoppner [5] by using a matrix to list exhaustively all the relations and using graph connectivity to mention a temporal pattern only once if it is connected. The sample relation matrix is shown in Figure 2.2.

In matrix representation upper diagonal triangle can be used to generate graph as other lower half would result in same graph with only direction reversed. Initially single node is selected in the graph and is grown with respect to different possible candidates. The connected component can thus contain small length patterns within it as subgraphs.

a)



b)

	A	B	C	D
A	e	***e***	***c***	***c***
B	e	e	***c***	***c***
C	d	d	e	***b***
D	d	d	a	e

FIGURE 2.2: a) Example Interval Sequence b) Relation Matrix [5]

2.2.3 Augmented Hierarchical Representation

The ambiguity of same pattern inferring possible different relations was addressed by Patel [13] by including the count of each relation as shown in Figure 2.3.[1] Consider the pattern ”(A overlaps B) overlaps C” of three interval events this can be interpreted in different ways as accounting the relation of C with respect to A.

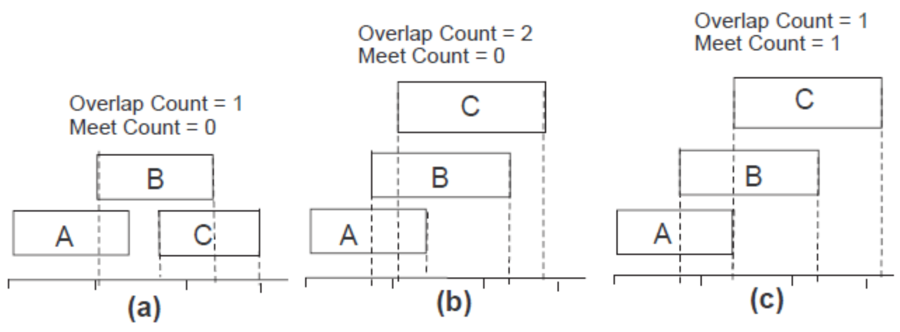


FIGURE 2.3: Different Interpretations of (A overlaps B) overlaps C [13]

Then, the three different representation would be,

- (A overlaps[0,0,0,1,0] B) overlaps[0,0,0,1,0] C
- (A overlaps[0,0,0,1,0] B) overlaps[0,0,0,2,0] C
- (A overlaps[0,0,0,1,0] B) overlaps[0,0,1,1,0] C

[1]as illustrated in [13]

Where the order in square brackets is count of all relations in order contains, finishes, meets, overlaps and starts. This way the pattern depicted is lossless representation of the initial configuration.

2.2.4 Endpoints Representation

Wu and Chen proposed endpoint representation [21] to unambiguously depict the pattern where starting and finishing points of each interval event.

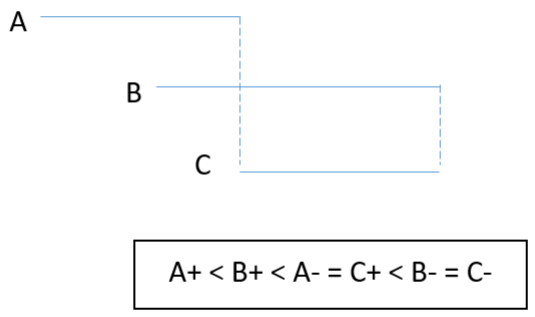


FIGURE 2.4: Endpoints Representation

In Figure 2.4, the endpoints representation for three events is shown. The starting point is denoted by event symbol followed by the + symbol and ending point as event symbol followed by – symbol. So the sequence can be represented by ordering the starting and ending points in occurrence order with only two possible occurrences either after previous point or coinciding with previous point.

2.3 Interval Pattern Mining Algorithms

Interval pattern mining is based on identifying the aforementioned Allen's relation [1] in the given temporal database. Only recently the work is carried out on interval pattern mining. To the best of my knowledge all the interval pattern mining algorithms are serial in order and no prior work is done on addressing the parallel techniques for interval pattern mining because of the sheer complexity involved in the same.

TABLE 2.3: Candidate Generation from Previous Iteration

L(2) {frequent}
B before C {3}
B before D {3}
C equals D {4}

Itemset	**C(3) {due}**
B before C	D {B, C}
B before D	C {B, D}
C equals D	B {C, D}

2.3.1 Apriori-like Algorithms

In early days, most of the algorithms for discovering temporal patterns were Apriori-like. Villifane [20] proposed a mining method by converting interval sequences into containment graphs. Kam and Fu [7] used Apriori like approach on hierarchical representation. Cooper et al. proposed Recent Temporal Pattern (RTP) [2] mining that mines frequent time-interval patterns backward in time starting from patterns related to the most recent observations, which can be typically most important for prediction in certain situations. The method involves two phases, namely,

1. candidate generation, and
2. k-length pattern generation

This method uses multiple passes over the dataset with each iteration. In first phase potential atomic event is found for candidate C(k) using the pattern generated in previous iteration. Next, in the second phase potential temporal pattern L(k) is formed using the atomic candidate C(k) and the temporal pattern L(k-1). For example, for the database in Table 2.1 the L(2) and subsequent C(2) candidates is shown in the Table 2.3 for the minimum support 3. The candidate generation for the frequent pattern of length 2 is used for finding the candidate for next iteration.

An interval event in an iteration is identified as candidate if that event is frequent with any of the events present in the frequent temporal pattern found in previous iteration. If it does not have any frequent relation with any of the events present in the pattern in previous iteration then the candidate set for that pattern is empty.

From this example we can easily see, in this way ambiguity might arise resulting in same pattern being carried forward in next iteration represented in different way in hierarchical manner.

HDFS [12] converts the temporal database into a list of sequence ids for each event. Then merging the sequence ids using enumeration tree. The enumeration tree is generated one by one for each event. For the database given in the Table 2.1 enumeration tree is shown in Figure 2.5 for minimum support 3.

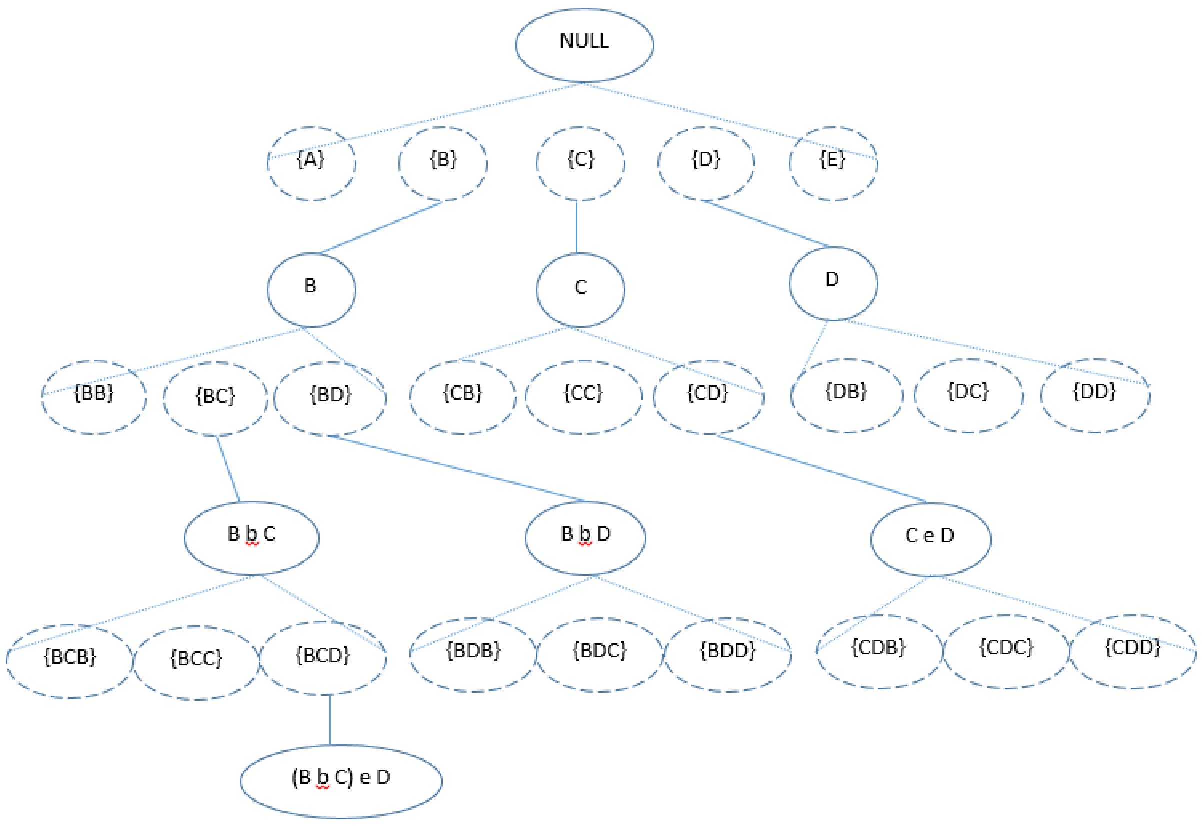


FIGURE 2.5: Arrangement of Enumeration Tree

In Figure 2.5 the arrangement of enumeration tree, the dotted ovals are the possible candidates and the solid ovals are the actual pattern found. Only one set of relation is used to expand tree as inverse is implicit and redundant. The equal relation can be addressed by lexicographically maintaining the event id.

IEMiner [13] is one other Apriori-based algorithm. First, it scans the entire database to obtain all the frequent single events. Then generates the higher level candidate set until the candidate set is empty. The candidate generation follows that a (m+1)-pattern is a candidate for frequent itemset given it is formed using a frequent m-pattern and 2-pattern which is present in at least m-1 of the frequent m-patterns. While the support count is found by keeping the track of active and passive events at particular time. This way the support count can be calculated in the single pass.

Proof. Let TP(m+1) be frequent pattern, then there are m+1 frequent m-patterns. Also, ei and ej be two events in TP(m+1) then there exists m frequent m-patterns containing ei. Similarly there exist m frequent m-patterns that can be formed by ej. Therefore, ei and ej together must be present at least in m-1 patterns.

2.3.2 Projection based Algorithms

Projection based interval mining uses divide and conquer strategy to find frequent temporal patterns. If α is a sequential pattern, then let $DB_{|\alpha}$ denotes α projected database, which is a collection of sequences' suffixes with regard to prefix α. Then α+1 pattern is generated using the projected database and extending the α sequential pattern. *TPrefixSpan* [21] uses projection based method along with the endpoint representation of interval sequence to extend the patterns. It recursively scans the projected database to find longer sequential patterns. Sadasivam [16] refined TPrefixSpan so that number of database scans is reduced. HTPM [22] was developed to mine hybrid patterns i.e. both point-based and temporal patterns from the sequence database.

2.3.3 TPMiner Algorithm

The most recent *TPMiner* [3] extends the concept of aforementioned database projection. Events with same sequence ID are clubbed in single interval sequence. The procedure for *TPMiner* follows the steps mentioned below. Julia et al. developed *Inc_TPMiner* (Incremental Temporal Pattern Miner) [24] to incrementally discover temporal patterns from interval-based data.

I Temporal database is transformed to endpoint representation. TPMiner counts the frequency of each endpoint i.e. the starting and ending points.

II The infrequent endpoints are removed. Then for each of the remaining starting endpoint, the projected database is built and TPSpan is called recursively.

III If the sequence is a valid temporal pattern then output.

TPMiner incorporates three pruning strategies to reduce the search space as following.

1. To find support of all endpoints, it is not necessary to scan each sequence from beginning to end. Instead start at each sequence and stop at the starting point of the desired endpoint. This is referred to as scan pruning.

2. If the ending endpoint in the projected database has no corresponding starting endpoint in the prefix it is removed. It is used to prune non applicable endpoints before creating its projected database. This is referred to as point pruning.

3. During construction projected database, the pattern is essential only if its starting endpoint is in the prefix sequence. All non-essential patterns are ignored in discovery of frequent patterns. This is referred to as postfix pruning.

2.4 Parallel Frequent Itemset Mining

Although parallel itemset mining is not addressed for finding temporal patterns due to complexity of the various interval pattern mining algorithms. But there is some work being carried out to find frequent patterns in sequential pattern mining in parallel fashion which in turn can provide the insight to manifesting the similar techniques for interval pattern mining. Most of the previously developed parallel algorithms [17][10] were Apriori based which suffer from high I/O overhead and synchronization.

2.4.1 Parallel FP-growth

A small number of FP-growth-like implementation [6][8][11] address parallel FIM problem. *PFP* [8] is a popular parallel frequent itemset mining algorithm that parallelize the classic FP-Growth algorithm. Recent work in parallel Apriori-like algorithm, such as sequence-growth [9] that uses lexicographical tree, has to scan database multiple times and exchange large number of candidate itemset. *PFP* uses the MapReduce approach which address the problem of dataset not fitting in memory. *PFP* uses three MapReduce phases to parallelize FP-Growth namely.

1. Sharding the input data and parallel support counting
2. Parallel and self-adaptive version of FP-growth
3. Aggregation

The detailed flow design of PFP is shown in Figure 2.6 [8] below with three MapReduce jobs and total 5 steps.

Sharding divides the database in consecutive parts and stores them in different machines. While the **Parallel Counting** does a MapReduce task for counting the support of the items in database. Each mapper works on single shard. The result of items and their count is stored in F-list. Further the item list is divided into groups with group id in G-list. **Grouping** items step converts the transactions into group-dependent transactions such that the FP-tree of these groups are independent.

Mapper- Generate intermediate key value pair with transaction id as key and group ids as value for each itemset in that transaction. Generate the final key value pair with key as group id and value as transaction ids for each of the group id. Here the group-dependent shard is the value associated with key group id.

Reducer- Constructs local FP-tree for each group-dependent shard parallel and stores the found patterns with their support count into a max heap.

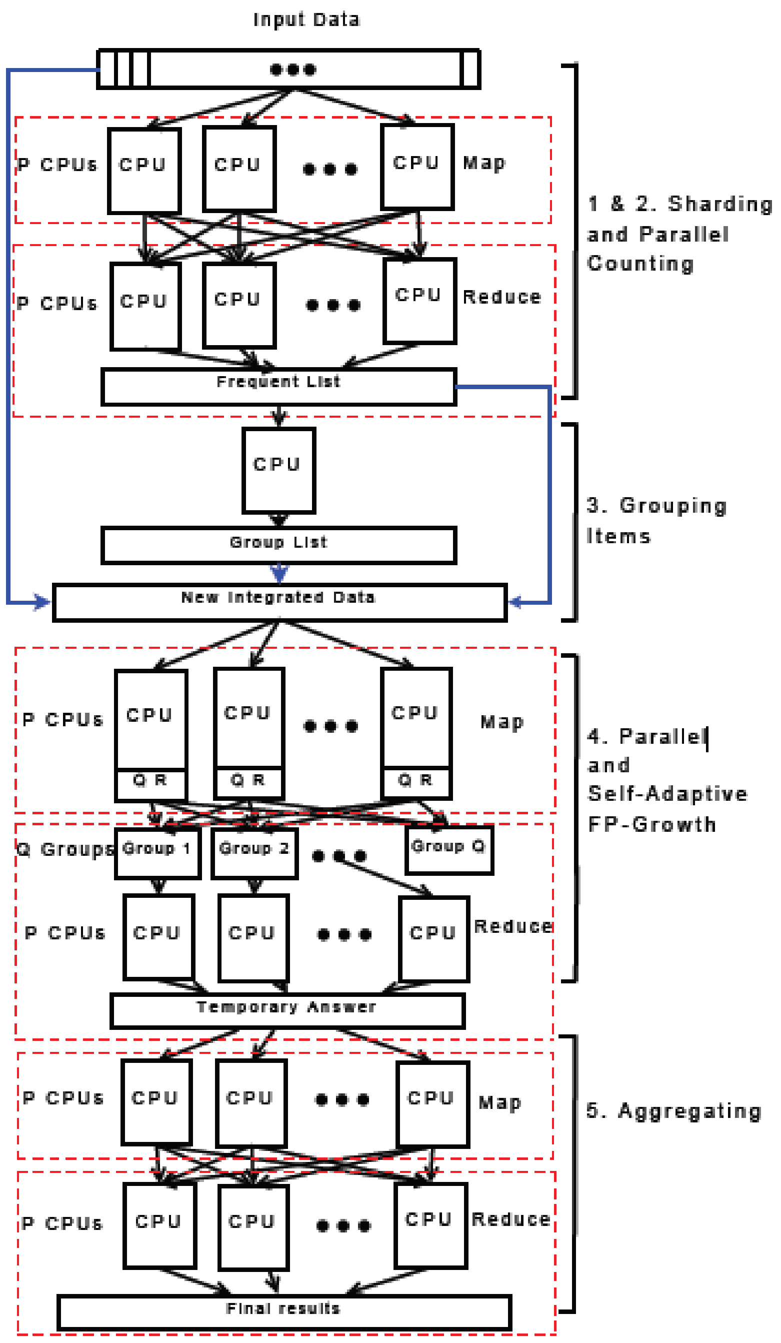


FIGURE 2.6: PFP Computation Stages [8]

Aggregation. For all the items the corresponding frequent patterns are listed out of which required number of mostly supported patterns are reported.

2.4.2 *Frequent Itemset Ultrametric Method*

FiDoop [23] is the most recent technique that incorporated *FIU-tree* [19] to mine frequent itemset mining. FIU-tree has certain advantages over traditional FP-tree. FIU-tree provides reduced I/O overhead, offers an integral way to partition the dataset, compressed storage, and avoids the recursive traversal of tree.

FIU-tree

An ultrametric tree is a tree diverging from a single node i.e. root in which distance from the root to leaves remains same. The frequent item ultrametric tree (FIU-tree) can be constructed by following steps below.

1. The root of tree be *'null'*. The frequent itemset $p_1, p_2, ..., p_k$ is placed as a skewed path of length *(*k) composed of edges as $(p_1, p_2), (p_2, p_3), ..., (p_{k-1}, p_k)$, starting with child p_1 of the root and ending with leaf p_k.

2. The k-FIU-tree is built by placing all frequent itemsets of length k starting from root to last item in itemset in a single path. Hence, all the leaves are at the same distance k from root of *k-FIU-tree* e.g. Figure 2.7 shows the tree for k = 5.

3. The leaves in the FIU-tree contain two variables namely item-label and its count. The count of the item in leaf is equal to the number of transactions that contain the item as the end of itemset sequence path. While the non-leaf nodes contains two fields namely item-name and node-link which is a pointer to child nodes in the FIU-tree. For illustration there are two leaf nodes i.e. m:1 and p:3 in Figure 2.7 with their count.

FiDoop Methodology

Three following three MapReduce Jobs in FiDoop is explained briefly.

A. First MapReduce Job

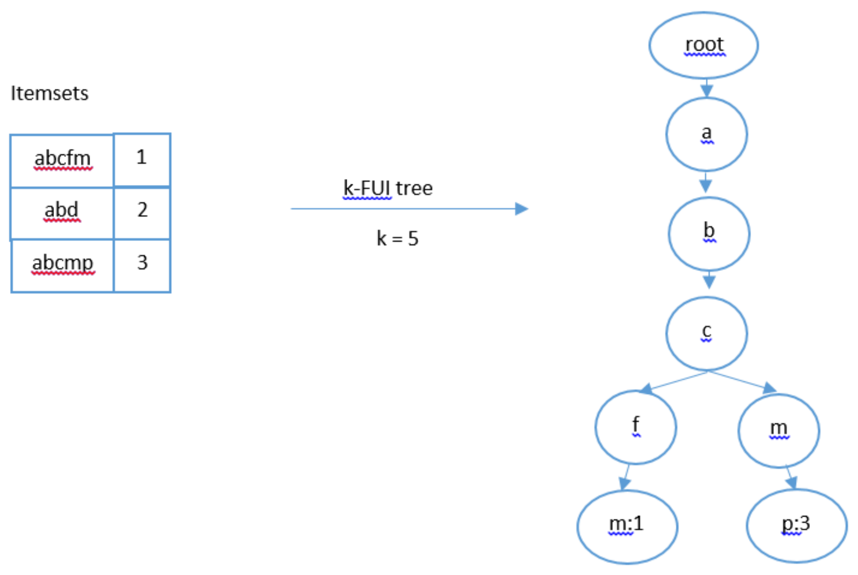


FIGURE 2.7: k-FUI-tree example

The first MapReduce job creates all the frequent one-itemset. Each mapper reads all the transactions from its local files sequentially and generates local one-itemsets. The reducer then combines all the local one-itemset and creates a global frequent one-itemset.

B. Second MapReduce Job

The second MapReduce job scans the database second time to prune out the infrequent items from each transaction. The mapper here for each transaction outputs the k-itemset ($2 \leq k \leq M$ where M is the maximum pruned value) with pruned out items as key and count one as value. The reducer then outputs the itemset id as key and value contains the itemset and the count as similar pruned out itemset may be generated.

C. Third MapReduce Job

The final MapReduce job decompose the k-itemset, construct the k-FIU-tree, and mines the frequent itemset. Each mapper decomposes the k-itemset into small sized itemset of length between 2 to k-1 in parallel. Then mapper constructs FIU-tree from local decomposed itemsets of same length. The mapper finally outputs the number of items in itemset and the FIU-tree so itemsets of same length are passed to single reducer. The reducer then creates the h-FIU-tree and mines the frequent itemset by probing the count of leaves in h-FIU-tree.

Most Recent Parallel Study on Interval Patterns. All the current pattern mining algorithms on interval-based events are sequential in nature. They cannot scale to large data set which cannot be stored in single memory. The only parallel approach [15] addressed regarding interval data does quantitative analysis of sequential pattern which is out of the scope of this work. Although, the various parallel techniques in mining frequent patterns in instantaneous events can provide insights into parallelizing techniques.

2.5 Research Gap

It is observed that Apriori-based interval pattern mining algorithms may lead to huge number of candidate patterns because of the complex relation that can exist between two intervals. This is then followed by tedious work of calculating support of those candidate patterns. Only recently, projection-based interval pattern mining algorithms are devised in which complexity only increases with the lenght of pattern. In Section 1.3 we percieve the need for scalable pattern mining algorithms. These serial projection-based algorithms become infeasible when the dataset size increases too much. Since the pattern mining on interval data is itself an arduous task, to the best of my knowledge no prior work has been done to address the scalability issue on interval pattern mining. Therefore, there is a growing need to address the scalability issue in this area.

2.6 Summary

In this Chapter we discussed various representation schemes in section 2.2 for efficient storing and access of interval data. We also illustrated several serial Apriori-based pattern mining algorithms [20][7][12][13] for interval events. Then we presented several projection-based pattern mining algorithms [21][16][22] for interval events which avoids the generation of large number of candidate patterns. The most recent one *TPMiner* [3] as the author claims performs better than previous interval pattern mining algorithms. Then we illustrated the methodologies adopted for parallelizing point-based pattern mining algorithms specifically [8] and [23] in detail. This provided us with the insight on how to approach a parallel interval pattern mining algorithm. In Chapter 3 we will present the proposed methodology to achieve the same.

PROPOSED METHODOLOGY

3

Temporal Pattern Mining is much computationally expensive compared to point-based events because of the complex number of relations between two interval events. This can be illustrated in terms of search space for both types of mining.

Definition. Event Interval. Let the complete list of temporal events symbols in the given temporal database be $E = e_1, e_2, .., e_k$. Then the temporal event is denoted by the triplet (e_i, s_i, f_i) where event $e_i \in E$ and $s_i < f_i$. The symbols s_i and f_i are the starting and finishing time of event e_i.

Definition. Maximal Property on Interval Sequence. The interval sequence contains series of triplets $< (e_1, s_1, f_1), (e_2, s_2, f_2), .., (e_n, s_n, f_n) >$ where $s_i \leq s_{i+1}$. Every interval (e_i, s_i, f_i) is maximal such that same event is not overlapped. If the maximal property is not met then concerned event can be merged as $(e_i, min(s_i, s_j), max(f_i, f_j))$.

Definition. Temporal Database. The database $DB = \{t_1, t_2, .., t_m\}$ consists of record t_i which consist of sequence id and the interval event.

3.1 Complexity Analysis

Point-based Events. For a point based database with two frequent events A and B, the top three levels of sequential mining search space is illustrated in Figure 3.1. Note that for a combination other event can either coincide with previous or just occurs after the previous event. So if the number of frequent events is e and the longest pattern is of length is l, the order of the search space would be,

$$e + e*(2*e) + e*(2*e)^2 + ... + e*(2*e)^{l-1} = \frac{e((2e)^l - 1)}{(2e) - 1} = O((2e)^l) \qquad (3.1)$$

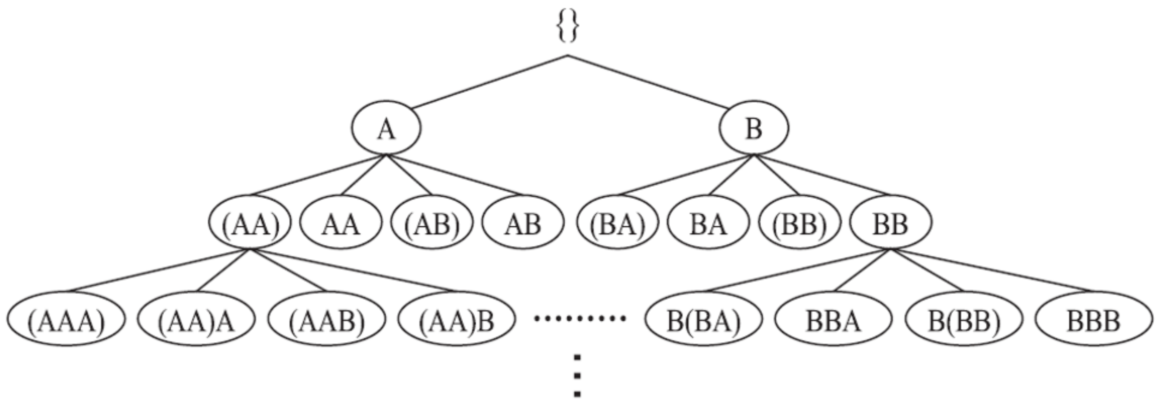


FIGURE 3.1: Search space for 2 frequent events in point based database [3]

Interval-based Events. Although for a temporal database containing two frequent interval events A and B, the search space is illustrated in Figure 3.2. The search space for temporal database is highly explosive. Note that the Allen's 13 relations is reduced in 7 relations without loss of generality. So again, if the number of frequent interval events is e and the longest temporal pattern is of length is *l*, the order of the search space would be,

$$e + e(7*e) + e(7*e)(7^2*e) + ... + e(7*e)(7^2*e)..(7^{l-1}*e) = O(7^{l^2}e^l) \qquad (3.2)$$

Obviously this create a large search space and make temporal pattern mining relatively more complex compared to point based sequential mining.

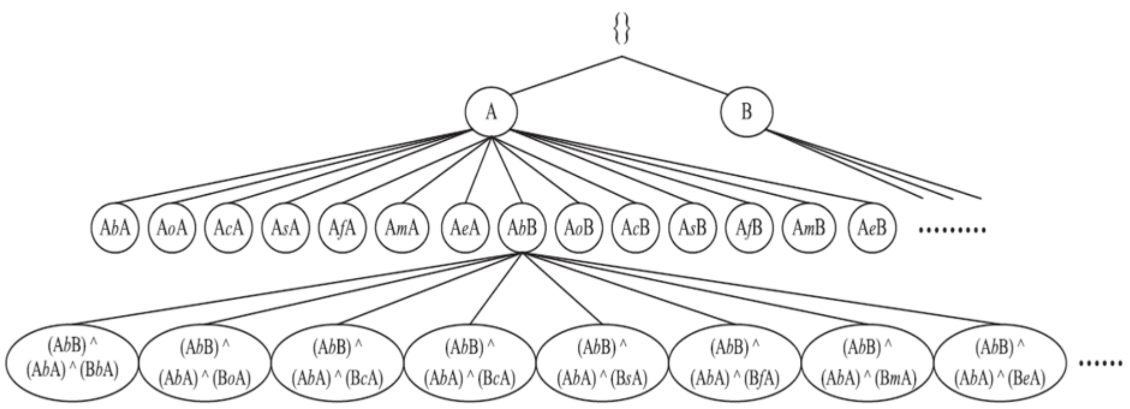


FIGURE 3.2: Search space for 2 frequent interval events in temporal database [3]

From the search space analysis, it is shown that complex relationships of temporal events is critical to designing an efficient algorithm. In **Apriori-like algorithms**, for each iteration this complex relationships leads to generation of huge number of candidate sequence which ultimately leads to the arduous task of finding support of it, although most them might not be present in the temporal database itself. Furthermore, in **Pattern-growth algorithms**, the complexity of temporal relation within the pattern itself leads to expensive computation. The low level tasks is expensive in terms of space and time complexity and are potential candidate for parallel techniques. So the motivation is to devise a complete, efficient, and scalable pattern-growth algorithm.

3.2 Representation Scheme

We used *endtime representation* in implementation which is an extension to the *endpoint representation* used in previous work for expressing temporal patterns. The major bottleneck for finding temporal patterns is the complex possible relations among interval events. The endpoint representation uses the arrangement of endpoints of interval event to express the relation among interval events in the sequence.

Endpoint Sequence. For given interval sequence $Q =< (e_1, s_1, f_1), (e_2, s_2, f_2), ..., (e_n, s_n, f_n) >$ the corresponding $ES_Q =< p_1, p_2, ..., p_{2n} >$ where each triplet (e_i, s_i, f_i) maps into two symbols

p_i and p_j corresponding to the endpoints e_i^+ and e_i^- with time point s_i and f_i respectively. The endpoints occurring together at the same time are collected in brackets. Furthermore, to deal with multiple occurrences of the same event in an interval sequence, the occurrence number is appended with the event symbol.

Then the temporal database DB is transformed into list of tuples $< ID, ES_Q >$ where ID is the sequence ID in the database and the endpoint representation of Q is ES_Q. The proposed endpoint sequence is maintained in sorted order base of time so that relation between two endpoints is reduced from Allen's 13 relations to just 2 namely *before* and *equal*.

Endtime Representation. The endtime representation also incorporates the actual time of each endpoint that is used in some applications for the logical extension of algorithm to other quantitative analysis. For example, including the time of endpoints can help predicting the time of the next occurrence of the event.

For a given interval sequence Q = $< (e_1, s_1, f_1)$, (e_2, s_2, f_2), ..., $(e_n, s_n, f_n) >$ and the corresponding endpoint sequence $ES_Q =< p_1, p_2, ..., p_{2n} >$ the time sequence would be $TS_Q =< t_1, t_2, ..., t_{2n} >$ where t_i is the occurrence time of endpoint p_i. Then the endtime representation would be a tuple of the corresponding items of endpoint sequence ES_Q and temporal sequence TS_Q i.e.

$$(ES_Q, TS_Q) =< (p_1, t_1), (p_2, t_2), ..., (p_{2n}, t_{2n}) > \tag{3.3}$$

The endpoint and endtime representation of the temporal database presented in Table 2.1 is depicted in Table 3.1. For example for the first sequence with sequence id 1, the endpoint representation would be $A+B+A-B-(C+D+)(C-D-)$ maintaining the order of time points i.e. starting and ending time while the endtime representation would be $\binom{A+}{1}\binom{B+}{3}\binom{A-}{4}$ $\binom{B-}{6}\binom{C+\ D+}{8\ \ 8}\binom{C-\ D-}{12\ \ 12}$ inclusive of the time of each endpoint. Note that endpoints occurring at same time are ordered in lexicographical fashion of the event symbol.

Temporal Pattern. Suppose two endpoint sequences $x =< p_1, p_2, ..., p_n >$ and $y =< q_1, q_2, .., q_m >$, here the p_i and q_j are pointsets such that $n \leq m$. Then x is the subsequence of y denoted by $x \subseteq y$ if there exist n integers $1 \leq g_1 \leq g_2 ... \leq g_n \leq m$ such that $p_1 \subseteq q_{g_1}, p_2 \subseteq q_{g_2}, ..., p_n \subseteq q_{g_n}$.

TABLE 3.1: Sample Proposed Endpoint and Endtime Representation

SID	**Event symbol**	**Start time**	**Finish time**	**Endpoint representation**	**Endtime representation**
1	A	1	4	$A+B+A-B-(C+D+)(C-D-)$	$\binom{A+}{1}\binom{B+}{3}\binom{A-}{4}\binom{B-}{6}\binom{C+\ D+}{8\ \ 8}\binom{C-\ D-}{12\ \ 12}$
	B	3	6		
	C	8	12		
	D	8	12		
2	B	3	6	$B+B-(C+D+)(C-D-)$	$\binom{B+}{3}\binom{B-}{6}\binom{C+\ D+}{8\ \ 8}\binom{C-\ D-}{12\ \ 12}$
	C	8	12		
	D	8	12		
3	C	6	8	$(C+D+)(C-D-)$	$\binom{C+\ D+}{6\ \ 6}\binom{C-\ D-}{8\ \ 8}$
	D	6	8		
4	A	1	3	$A+A-(C+D+)(C-D-)E+E-$	$\binom{A+}{1}\binom{A-}{3}\binom{C+\ D+}{6\ \ 6}\binom{C-\ D-}{8\ \ 8}\binom{E+}{12}\binom{E-}{14}$
	C	6	8		
	D	6	8		
	E	12	14		

Also, the y is termed as the supersequence x. For a temporal database sequence $< ID, Q >$ where id is sequence id and Q is endpoint sequence, Q is said to contain sequence x if x is subsequence of Q. Then the support of x would be number of sequences containing x i.e.

$$support(x) = |\{< id, Q > \mid (< id, Q > \subseteq DB) \wedge (x \subseteq Q)\}| \tag{3.4}$$

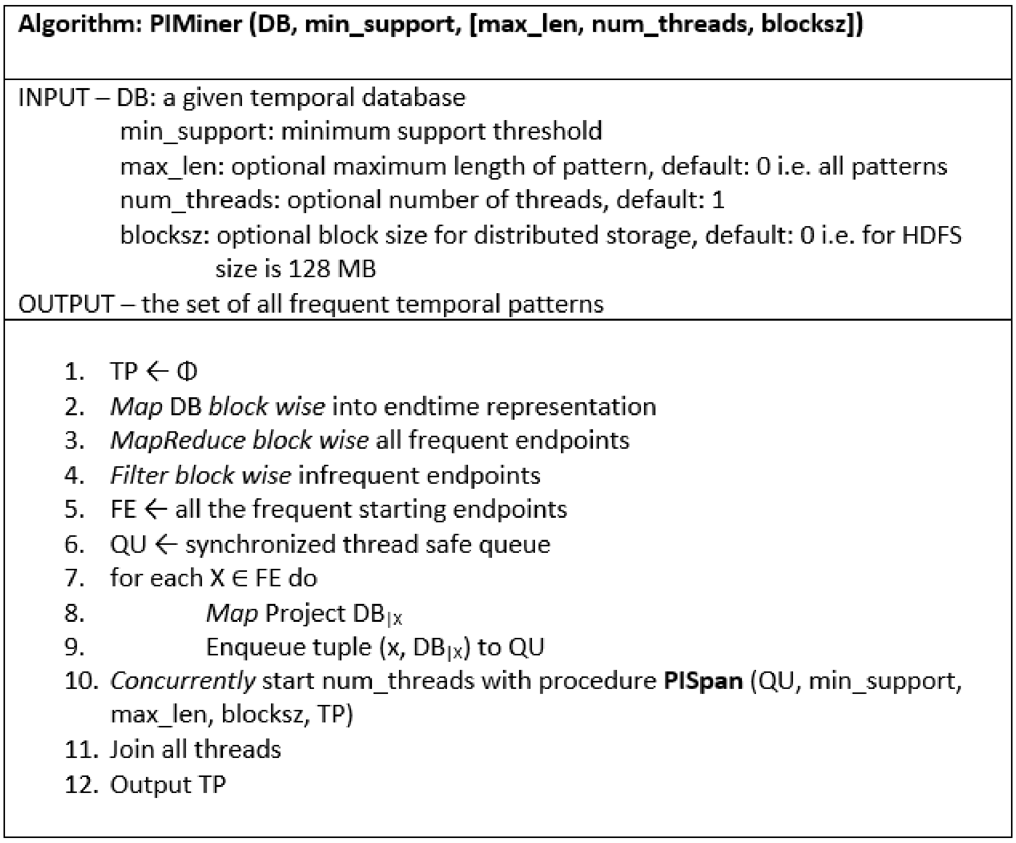
Therefore, the temporal pattern is a frequent endpoint sequence, having support greater than specified minimum support threshold, where every starting endpoint as a corresponding finishing endpoint and vice versa in the sequence that is endpoints exist in pairs.

Consider the database in Table 3.1 and minimum support threshold as 3, then the support of the sequence $< B+B-C+ >$ is 3 and support of $< (C+C-)(D+D-) >$is 4. Therefore both the sequences are frequent as their support is greater than threshold, but $< B+B-C+ >$ is not a temporal pattern because starting endpoint C+ has no corresponding finishing endpoint. Although, $< (C+C-)(D+D-) >$ is a temporal pattern as all the endpoints occur in pairs.

3.3 Proposed PIMiner Algorithm

In this section, proposed Parallel Interval Miner (PIMiner) algorithm is depicted based on the endtime representation described in previous section. It illustrates how to incorporated data and task parallelism in the proposed interval pattern mining algorithm. The pseudocode of PIMiner is shown in Figure 3.3. PIMiner manifests divide and conquer technique using projection database.

Projected Database. For a given endpoint sequence α in the database DB represented in endpoint representation. Then, the α-projected database, can be denoted by $DB_{|\alpha}$, is the collection of all the suffixes with respect to prefix α in all the sequences in DB.



Algorithm: PIMiner (DB, min_support, [max_len, num_threads, blocksz])

```
INPUT – DB: a given temporal database
        min_support: minimum support threshold
        max_len: optional maximum length of pattern, default: 0 i.e. all patterns
        num_threads: optional number of threads, default: 1
        blocksz: optional block size for distributed storage, default: 0 i.e. for HDFS
            size is 128 MB
OUTPUT – the set of all frequent temporal patterns

1.  TP ← Φ
2.  Map DB block wise into endtime representation
3.  MapReduce block wise all frequent endpoints
4.  Filter block wise infrequent endpoints
5.  FE ← all the frequent starting endpoints
6.  QU ← synchronized thread safe queue
7.  for each X ∈ FE do
8.          Map Project $DB_{|X}$
9.          Enqueue tuple (x, $DB_{|X}$) to QU
10. Concurrently start num_threads with procedure PISpan (QU, min_support,
    max_len, blocksz, TP)
11. Join all threads
12. Output TP
```

FIGURE 3.3: PIMiner Algorithm Pseudocode

3.3.1 PIMiner Framework

In the main framework of algorithm PIMiner in Figure 3.3, the input is the temporal database which contains the sequences of temporal events with starting and ending time. If the database is given in tuple format with sequence id and temporal event, then extra pre-processing step takes care of it by merging all the temporal events of same sequence in one list. The first step is to transform the database into endtime representation, detailed implementation would be followed in next section. The endpoints in the endtime representation is sorted by time of each point-event. The frequency of all the endpoints is calculated concurrently (Line 3, Figure 3.3).

The infrequent endpoints are removed partition wise in parallel (Line 4, Figure 3.3). Then all the frequent starting endpoints is collected as a starting point of temporal pattern. A thread safe queue is maintained so that different thread can access the queue concurrently. For each starting endpoint the projected database is generated, which is ultimately enqueued as a tuple of starting endpoint and the projected database (Line 9, Figure 3.3).

Then task parallelism is achieved as the mentioned number of threads are started which concurrently executes the PISpan procedure for extending the frequent sequence. All the temporal patterns are listed at last after the queue is empty and all threads finishes there execution.

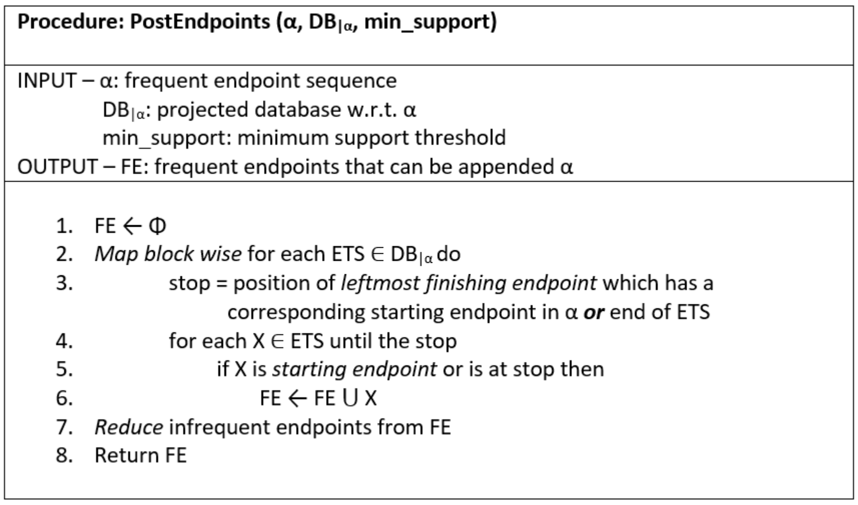
```
Procedure: PISpan (QU, min_support, max_len, blocksz, TP)

INPUT – QU: synchronized queue containing tuple of sequence and projected db
          min_support: minimum support threshold
          max_len: maximum length of pattern, default: 0
          blocksz: block size for distributed storage, default: 0
          TP: thread safe queue for frequent temporal patterns
OUTPUT – append found temporal patterns to TP

1. $(\alpha, DB_{|\alpha}) \leftarrow$ Dequeue QU
2. FE $\leftarrow$ PostEndpoints $(\alpha, DB_{|\alpha}$, min_support)
3. for each $X \in$ FE do
4.         Append X to $\alpha$ to generate sequence $\beta$
5.         if $\beta$ is a temporal pattern:
6.                 TP $\leftarrow$ TP $\cup$ $\beta$
7.         $DB_{|\beta} \leftarrow$ ProjectDB $(\beta, DB_{|\alpha})$
8.         if $DB_{|\beta} \neq \Phi$ and (max_len = 0 or $\beta$.len < 2*max_len) then
9.                 Enqueue tuple $(\beta, DB_{|\beta})$ to QU
```

FIGURE 3.4: PISpan Procedure Pseudocode

3.3.2 *PISpan Procedure*

The *PISpan* procedure, as shown in Figure 3.4 extends the frequent sequence to find longer frequent sequence. The endtime sequence and the respective projected database is dequeued (Line 1, Figure 3.4). The pruning is done by borrowing idea from PrefixSpan [14].The possible extension to the sequence is generated by procedure PostEndpoints collected as a list of frequent endpoints (Line 2, Figure 3.4). For each of the frequent endpoint, new sequence is formed by appending the endpoint (Line 4, Figure 3.4). If the new sequence is a valid temporal pattern, where all the starting endpoints have a respective finishing endpoints and vice versa, then the sequence is appended to output queue of temporal patterns (Line 6, Figure 3.4).

The projected database is generated with respect to the new sequence. The tuple of this projected database and new sequence is enqueued if the projected database is not empty and either *max_len* is not specified, which is the case for finding all the frequent temporal patterns, or the current length of the pattern is less than the *max_len* (Line 9, Figure 3.4).



Procedure: PostEndpoints (α, $DB_{|\alpha}$, min_support)

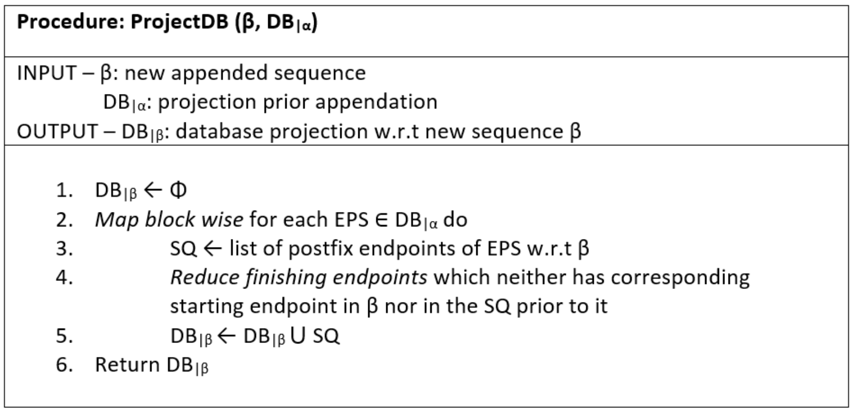
```
INPUT – α: frequent endpoint sequence
        DB_{|α}: projected database w.r.t. α
        min_support: minimum support threshold
OUTPUT – FE: frequent endpoints that can be appended α

1. FE ← Φ
2. Map block wise for each ETS ∈ DB_{|α} do
3.     stop = position of leftmost finishing endpoint which has a
           corresponding starting endpoint in α or end of ETS
4.     for each X ∈ ETS until the stop
5.         if X is starting endpoint or is at stop then
6.             FE ← FE ∪ X
7. Reduce infrequent endpoints from FE
8. Return FE
```

FIGURE 3.5: PostEndpoints Procedure Pseudocode

3.3.3 *PostEndpoints Procedure*

The *PostEndpoints* procedure as depicted in Figure 3.5 finds the frequent endpoints that can be appended to the α sequence and the resulting sequence is also frequent. We need to collect the endpoints only till the leftmost endpoint which has a corresponding starting endpoint in α, because beyond it collecting the endpoints would not lead to valid temporal pattern on concatenation.

If no such finishing endpoint is found all starting endpoints until the end of the sequence is collected (Line 3, Figure 3.5). The frequency of each endpoint is calculated concurrently (Line 7, Figure 3.5). Finally, the list of frequent endpoints is returned.



```
Procedure: ProjectDB (β, DB|α)

INPUT – β: new appended sequence
           DB|α: projection prior appendation
OUTPUT – DB|β: database projection w.r.t new sequence β

1.  DB|β ← Φ
2.  Map block wise for each EPS ∈ DB|α do
3.          SQ ← list of postfix endpoints of EPS w.r.t β
4.          Reduce finishing endpoints which neither has corresponding
            starting endpoint in β nor in the SQ prior to it
5.          DB|β ← DB|β U SQ
6.  Return DB|β
```

FIGURE 3.6: ProjectDB Procedure Pseudocode

3.3.4 ProjectDB Procedure

The *projectDB* procedure is used of generating the projected database with respect to the end-time sequence β. The framework of the procedure with pseudocode is shown in Figure 3.6. The list of suffix sequence SQ is collected with respect to the prefix β. Note that we can prune finishing endpoints *ev* which has no corresponding starting endpoints in β nor in the prefix of *ev* in SQ (Line 4, Figure 3.6). The projected database is returned in the last step.

3.4 MapReduce Job

The input database is first converted to endtime representation (ETS). Each sequence in ETS is a list of tuple of endpoint and the time of occurrence. So for example second sequence in Table 3.1 will be transformed to ETS [[A_1+,1][B_1+,3][A_1-,4][B_1-,6][C_1+,8][D_1+,8][C_1-,12][D_1-,12]] where occurrence number is also appended.

The *MapReduce* job is used to find the frequency of such endpoints in block partition way. This way, in PIMiner the frequency of all the endpoints is reduced which is even faster than single serial scan of the database and maintaining a dictionary as in TPMiner. All the endpoints in Map job is converted to tuple of endpoint and count one in a single sequence.

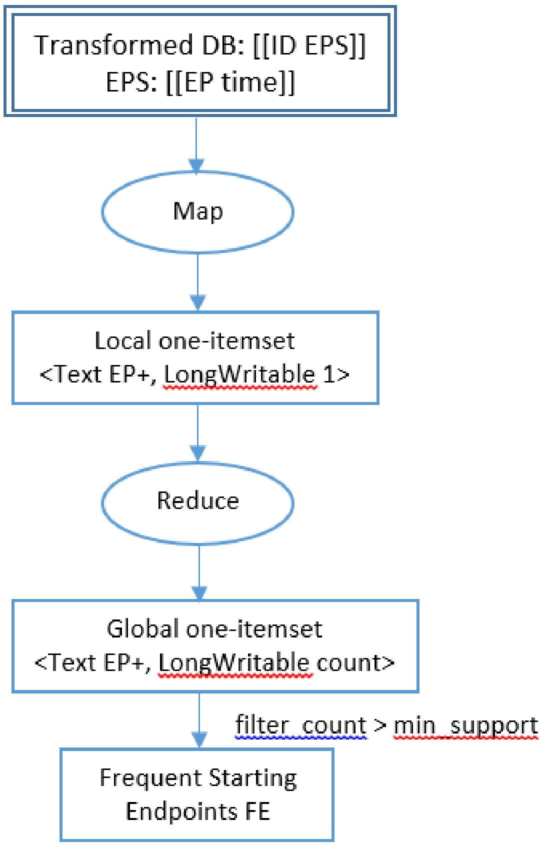

Transformed DB: [[ID EPS]]
EPS: [[EP time]]
Map
Local one-itemset
<Text EP+, LongWritable 1>
Reduce
Global one-itemset
<Text EP+, LongWritable count>
filter_count > min_support
Frequent Starting
Endpoints FE

FIGURE 3.7: PIMiner MapReduce Job Flowchart

The three phases involved in finding frequent endpoints and pruing infrequent ones while achieving block parallelism are:

1. Map Phase
2. Reduce Phase
3. Filter Phase

All the three phases are involved in achieving the task parallelism at block level. The number of partitions in the actual implementation can be determined using the optional input parameter *blocksz*. The number of partitions are determined at run time based on the variable size of the projected database.

So, the first sequence in Table 3.1 results in the tuples [A+, 1], [B+, 1], [A-, 1], [B-, 1], [C+, 1], [D+, 1], [C-, 1], and [D-, 1]. In this way all the sequences are flat mapped to these tuples as shown in Figure 3.8. Note that even an endpoint is present multiple times in the sequence it is counted as one and hence in key-value pair value is always one.

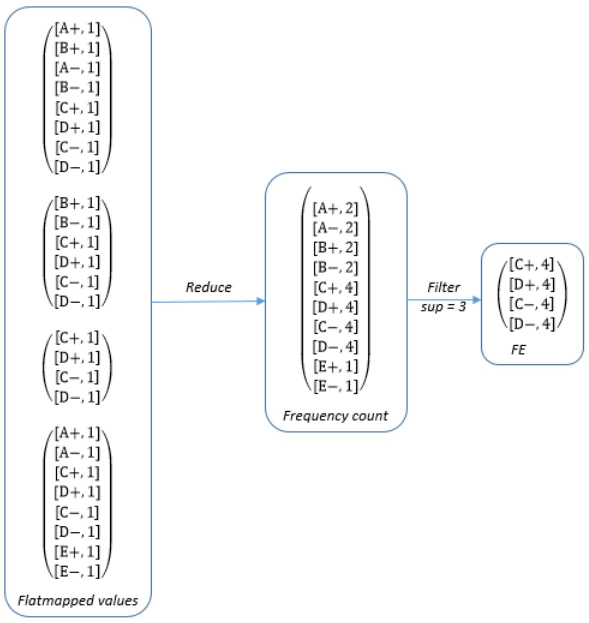


FIGURE 3.8: Frequent Endpoints Generation

In *reduce* step all the values are merged by specified addition operator based on the key value. Finally, the frequency count can be determined of all the endpoints involved in the whole projected database. All the endpoints without occurrence consideration are obtained with their count.

In *filter* step the endpoints with frequency less than the specified minimum threshold value are filtered as they cannot generate frequent endpoint sequence on concatenation with previous sequence.

For given example in Table 3.1 and minimum support threshold 3 then number of generated frequent endpoints are 4 as shown in Figure 3.8. The frequent endpoints now can be concatenated as suffix to already frequent sequence as already illustrated in PISpan procedure.

3.5 Summary

In this Chapter we presented the *PIMiner* algorithm for parallel mining of interval events. The algorithm incorporates task and data level parallelism. It generally improves over [3] by first identifying the portion of code that can be executed concurrently on different cores on the same chip. This way task level parallelism is achieved. Furthermore, the algorithm explicitly specify where data parallelism can be applicable. Finally the MapReduce job describes how using key-value pair block level parallelism is achieved while calculating support of a temporal pattern. In Chapter 4 we present the experimental setup for implementing the proposed algorithm.

EXPERIMENTAL SETUP

4

Data nowadays is so large that traditional data processing techniques are infeasible, then big data technologies are adopted. Datasets are growing rapidly as generated by numerous mobile devices, software logs, radio-frequency identification (RFID) chips and wireless sensors. Relation database cannot handle size of such magnitude. One of the most famous approach to handling big data was MapReduce model [4] given by Google in 2004. Apache open source framework Hadoop adopted the same MapReduce programming model.

4.1 Apache Spark Framework

Apache Spark is a lightning-fast cluster computing framework designed for fast computation. It was built on top of MapReduce and extends MapReduce model to other types of computations like stream processing and interactive queries. The distributed file used by Spark is HDFS which is nothing but Hadoop Distributed File System only. Spark is faster than Hadoop because the computation is carried out in-memory. Spark comes with APIs for Scala, Java and Python. Sparks uses Resilient Distributed Data storage (RDDs) which have fault tolerant implicitly. Spark also provide support for SQL through SparkSQL, machine learning through MLib and graph processing modules through GraphX library.

Spark has its own cluster management system and is dependent on Hadoop only for storage. There are three ways in which Spark can be built alongside Hadoop. The most important feature of Spark is in-memory clustering cluster computing that increases the computation speed of an application as the intermittent time between two MapReduce steps the intermediate result is saved in memory and not on disk.

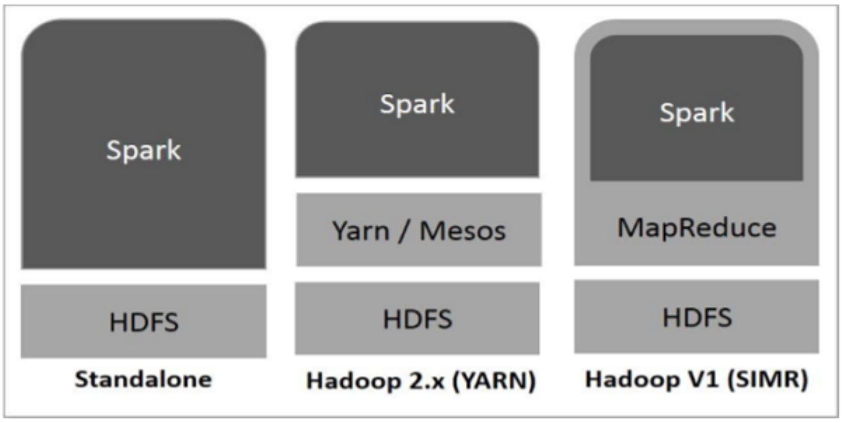


FIGURE 4.1: Apache Spark Deployment

There are three methods in which Spark can be built alongside Hadoop. The three Spark deployment methods[1] is shown in Figure 4.1. They are,

1. **Standalone.** In this deployment Spark sits above the HDFS and space for HDFS is though allocated explicitly. In this way both MapReduce and Spark and MapReduce run concurrently to complete all the Spark jobs in the cluster.

2. **Hadoop Yarn.** In Hadoop Yarn deployment, Spark execute over Yarn without any prior installation and no root privileges is necessary. This way the Spark is integrated with already existing Hadoop ecosystem.

3. **Spark in MapReduce.** In this deployment method, in addition to standalone deployment, the spark jobs can be launched separately. In SIMR the Spark shell can be used without any administrative rights.

[1]Spark 2.2.1 deployment: `www.spark.apache.org/docs/latest/cluster-overview.html`

4.2 Cloudera Distribution of Hadoop Deployment

CDH (Cloudera Distribution of Hadoop) is an open source Hadoop distribution provided by Cloudera Inc, an American enterprise software company. It is the most complete, robust and widely deployed Apache Hadoop distribution. More enterprises have used CHD than all the other distributions combined together. The services provided by CDH can be summarised in Figure 4.2.[2] It is the only distribution which provides solution to interactive queries.

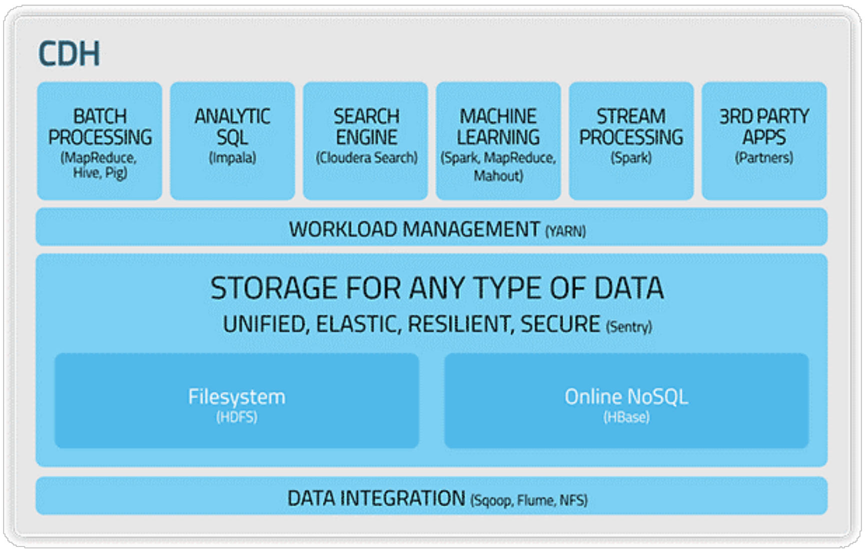

CDH
BATCH PROCESSING (MapReduce, Hive, Pig)
ANALYTIC SQL (Impala)
SEARCH ENGINE (Cloudera Search)
MACHINE LEARNING (Spark, MapReduce, Mahout)
STREAM PROCESSING (Spark)
3RD PARTY APPS (Partners)
WORKLOAD MANAGEMENT (YARN)
STORAGE FOR ANY TYPE OF DATA
UNIFIED, ELASTIC, RESILIENT, SECURE (Sentry)
Filesystem (HDFS)
Online NoSQL (HBase)
DATA INTEGRATION (Sqoop, Flume, NFS)

FIGURE 4.2: Cloudera Taxonomy

4.3 System Details

The details of the CDH used is mentioned as following. The CDH is installed over the Dell rack server.

Version: Cloudera Express 5.6.0

Java VM Name: Java HotSpot(TM) 64-Bit Server VM

Java VM Vendor: Oracle Corporation

Java Version: 1.7.0_67

[2]CDH overview at `www.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_intro.html`

TABLE 4.1: Dell Rack Server UGPC Lab

Name	Local IP	Roles	Cores	Memory	Disk
blade1.cloud.org	10.0.0.1	HDFS Balancer, HDFS NameNode, Hive Server, Oozie Server, Hue Server, YARN Resource Manager, YARN JobHistory Server, Spark (standalone) Master	24	128 GiB	2 TiB
blade2.cloud.org	10.0.0.2	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB
blade3.cloud.org	10.0.0.3	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB
blade4.cloud.org	10.0.0.4	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB
blade5.cloud.org	10.0.0.5	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB
blade6.cloud.org	10.0.0.6	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB
blade7.cloud.org	10.0.0.7	HDFS DataNode, YARN NodeManager, Spark (standalone) Worker	12	32 GiB	1 TiB

The Cloudera Manager has the following specifications. The Cloudera Manager system is used to send commands over the server over which the CDH is installed.

OS: Ubuntu 12.04 LTS 64-bit

Memory: 11.6 GiB

Processor: Intel® Xeon® CPU E5-16500 @3.20GHz X12

Disk: 357.4 GB

The details of rack server installed in UGPC Lab to run the job is illustrated in Table 4.1. The data is distributed stored across HDFS DataNode and there are 6 standalone Spark Workers for carrying out the computation.

4.4 Summary

The proposed *PIMiner* algorithm is implemented on spark using python on Dell Rack Server. We also implemented the *TPMiner* algorithm in python on blade1. In PIMiner algorithm implementation blade 1 acts as the spark standalone master and manages the spark jobs while in the second case it is the only computing node. PIMiner uses HDFS for storing and managing the temporal database. The database is divided into number of blocks depending upon the optionally specified block size. These blocks are stored in distributed fashion over the HDFS data nodes i.e. blade[2-7] while maintaining the replication factor of 3. In Chapter 4 we will observe the performance of our algorithm against serial version. Also we will present the effects of different parameters on our algorithm.

5 PERFORMANCE STUDY

PIMiner algorithm can efficiently mine temporal patterns. Since, to the best of our knowledge no work is done on parallel interval mining, we implement *TPMiner* algorithm, the most recent interval pattern mining algorithm, along with scalability tests to evaluate the performance of *PIMiner*. The system specifications are already specified in previous chapter. The performance study is carried out on both synthetic as well as real world datasets.

We conduct several tests including execution time, distribution of patterns, number of sparks, effects of threads, and scalability tests among others. When size of the dataset is not huge in kilo bytes the task parallelism is evident and only when size of dataset is in several hundred mega bytes, then the effects of data parallelism is evident.

TABLE 5.1: Synthetic Data Generation Parameters

Parameters	**Description**
S	Number of temporal sequences
C	Average length of sequence
E	Total number of interval events
N_f	Average length of frequent pattern
N_e	List of possible duration of an interval

5.1 Synthetic Data Generation

The synthetic datasets for experiments are generated as proposed by Srikant and Agarwal [18]. This proposed program was meant for time point-based events and hence needed to be modified accordingly. The several parameters that were used for synthetic temporal data generation are shown in Table 5.1

The sequences generated are maximal as no same temporal event meets or overlaps within the same sequence. The total number of sequences generated is S. The event symbol is randomly selected while generating the sequence from the list of integers from 0 to E-1 where E is the total number of interval events across the temporal database.

Since the interval events have a duration, the program requires additional tuning for interval data generation. We used the method as proposed by Wu. The duration of the events are chosen from three categories viz. short, medium and long events. The duration of short, medium and long events are 4, 8 and 12 units respectively. Hence, the list N_e contains three elements i.e. $N_e = [4, 8, 12]$ in the program. The duration of the interval event is randomly selected from the list.

Finally, the temporal relations among the consecutive interval events is randomly selected to generate maximal sequence. The temporal relation is chosen from the set {*before, equal, meets, overlaps, starts, finishes*}. We generate S such sequences. The size of the sequence is determined prior to generation with from the Poisson distribution with mean equal to C. Additionally, a list of frequent sequences are generated that is distributed for the support P = 0.5 in the

TABLE 5.2: S10kC20E1k Dataset Aspects

Parameters	**Values**
Number of sequences (S)	10000
Average length of sequence (C)	20
Number of events (E)	1000
Average length of frequent pattern (N_f)	7
Duration of the event (N_e)	[4 8 12]
Average duration of interval (D)	7
Support for pattern generation (P)	0.5
Total number of intervals (I)	200326

sequences randomly selected. If the length of the sequence is smaller than the randomly selected length than extra events are added with the randomly selected relations, or if the length of sequence is longer then the events are pruned at the end. The length of the generated frequent sequence is determined from the Poisson distribution with mean equal to N_f.

5.2 Performance on Synthetic Datasets

In the dataset generation several parameters are fixed. For the dataset S10kC20E1k the number of sequences is S = 10000, the total number of events is E = 1000 and the average length of the sequence is C = 20 as shown in Table 5.2.

In Figure 5.1 the execution time of our algorithm is compared to the serial TPMiner algorithm for the dataset S10kC20E1k. Maximum length of the patterns to be mined is set to L = 5. Since the size of the dataset is only 2.2MB the block size is set to default and there is no need for data parallelism as the completer dataset would fit in a single block.

Here the task parallelism is achieved by increasing the number of threads i.e. in Figure 5.1 the number of threads is set to 24 and 96 and execution time is determined for different support values at 0.2, 0.3, 0.4, and 0.5. The execution time graph in clearly shows that PIMiner takes less time than TPMiner as the support is reduced. This is due to the increased task parallelism by increasing number of threads.

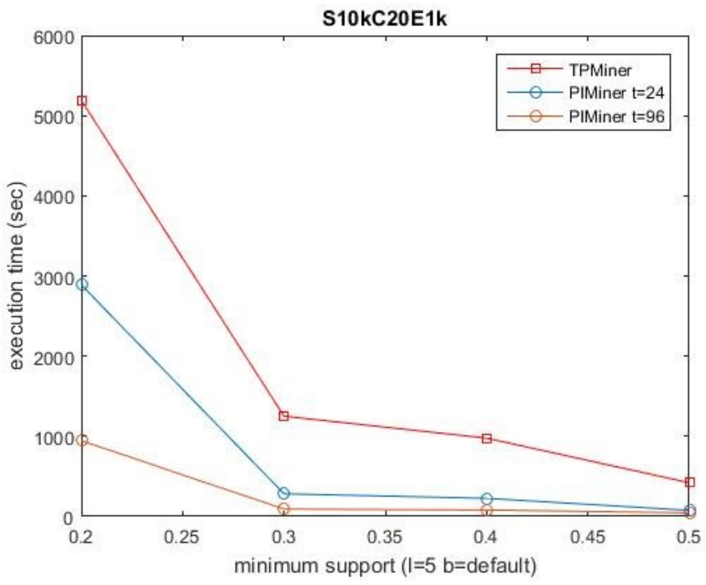


FIGURE 5.1: Execution Time Analysis for S10kC20E1k

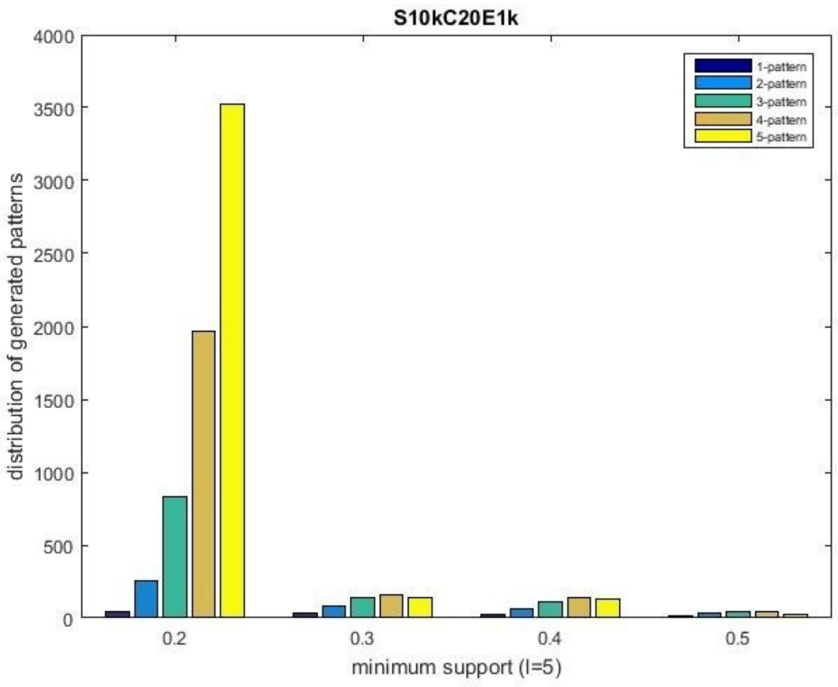


FIGURE 5.2: Distribution of Generated Patterns for S10kC20E1k

The distribution of the patterns for the dataset S10kC20E1k as 1-length, 2-length, 3-length, 4-length and 5-length is shown in Figure 5.2 for different values for support. It is evident that the number of patterns of shorter length hardly increases compared to the patterns of longer length.

This is because the dataset is generated for frequent patterns with average size of $N_f = 7$.

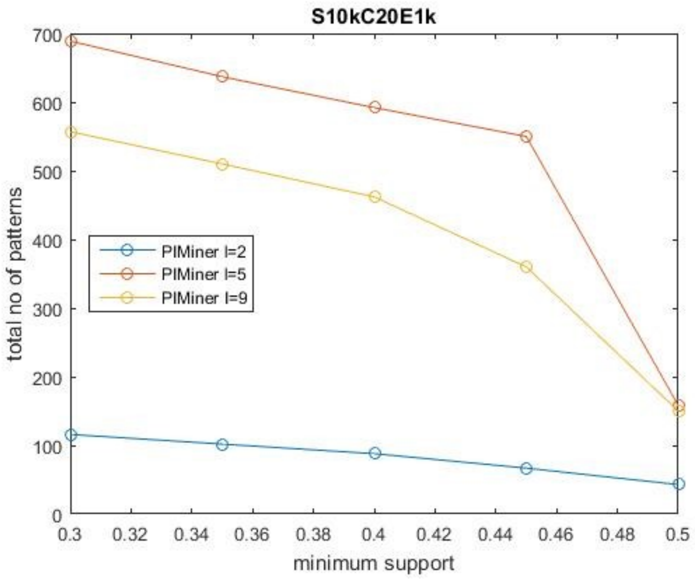


FIGURE 5.3: Number of Patterns on Maximum Length of Patterns for S10kC20E1k

We can observe the effects of changing the maximum length allowed for mining frequent pattern for different support values in Figure 5.3. The different values of maximum length of frequent pattern is used as L=2, 5 and 9. The support values used for comparison are 0.3, 0.35, 0.4, 0.45, and 0.5. As the maximum length allowed is increased more number of patterns is generated as expected. Again for low value of L = 2 the number of patterns tends to increase linearly but as the length is increase and support is decreased, longer patterns are more frequently mined as observed in distribution of found patterns in Figure 5.2. Hence, the total number of patterns is more steeper for the high value of the maximal length of frequent value allowed.

We can also observe the effects of changing the maximum length allowed on execution time. In Figure 5.4 the execution is plotted against different values for maximum allowed pattern length. The support value is fixed to sup = 0.2 and number of threads to t = 96 to so that program takes reasonably less time.

It can be illustrated that as the maximum length is increased the execution time increases exponentially. The total number of generated patterns is also shown in the figure prefixed with '#'. The total number of new x-length pattern can be determined by subtracting the total patterns at (x-1)-length by x-length. For example the total number of 4-length patterns generated for the support value 0.2 is 1960 (3082 – 1122) as the total number of frequent patterns up to length 4 are 3082 and total number of frequent patterns up to length 3 are 1122 respectively.

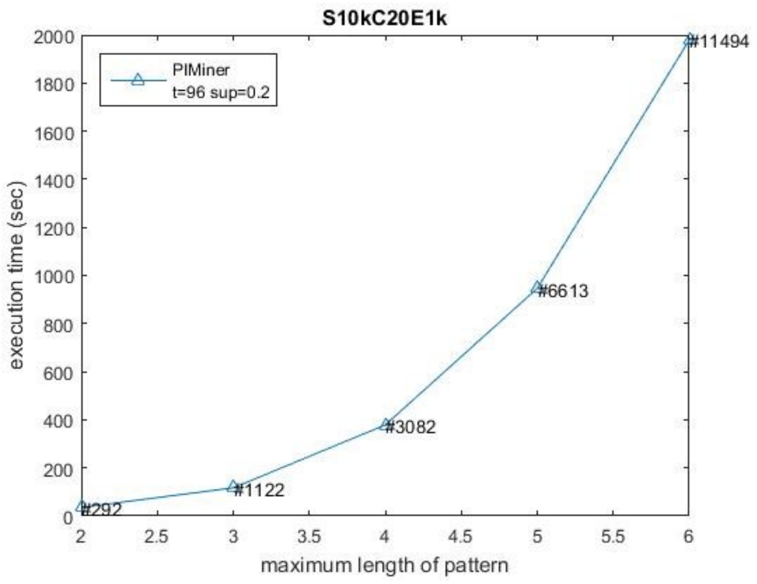


FIGURE 5.4: Performance Analysis on Maximum Length of Pattern for S10kC20E1k

5.3 Real World Dataset Analysis

In addition to the experiments on synthetic dataset, we have conducted the performance study on a real dataset to show the applicability of the algorithm and efficiency compared to traditional serial algorithms. The dataset we have used is of American Sign Language (ASL) dataset.

In ASL dataset we can determine the relationship between gesture field used and the grammatical structure. The dataset contains total of 730 utterances. Each of the utterance contains the gestural as well as the grammatical field. The complete specification of ASL dataset is shown in Table 5.3.

The size of the dataset is 213.6KB. So task parallelism is enough to observe the performance of the PIMiner compared to serial TPMiner. The block size is again default and the whole dataset is in the single block. The execution time for the both the algorithms is shown in the Figure

TABLE 5.3: ASL Dataset Aspects

Parameters	Values
Number of sequences (S)	730
Average length of sequence (C)	31
Number of events (E)	234
Average duration of interval (D)	13
Total number of intervals (I)	22727

5.5. The number of threads is set to 96 and different support values used are 0.4, 0.3, 0.2, 0.2, 0.1 and 0.05. It can be observed that the execution is exponentially increasing as the support decreases.

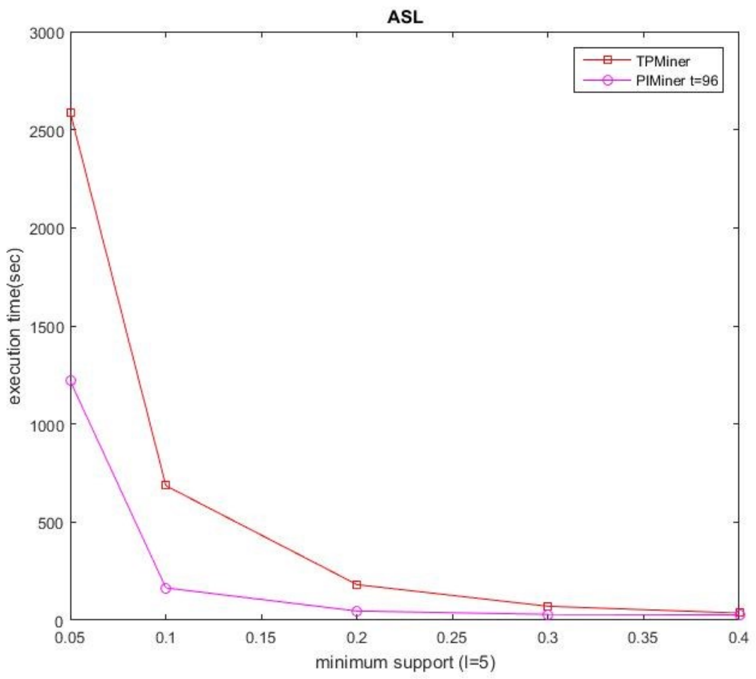


FIGURE 5.5: Execution Time Analysis for ASL Dataset

The distribution of generated patterns is shown in Figure 5.6. It can be observe that medium length patterns are among the most frequent that is of 3-pattern is one of the most frequent which holds truth as verified against Papapetrou et al.

The number of spark jobs are increased drastically as the support is decreased as evident in Figure 5.7. The number of spark jobs includes mapping, reducing and collection among other

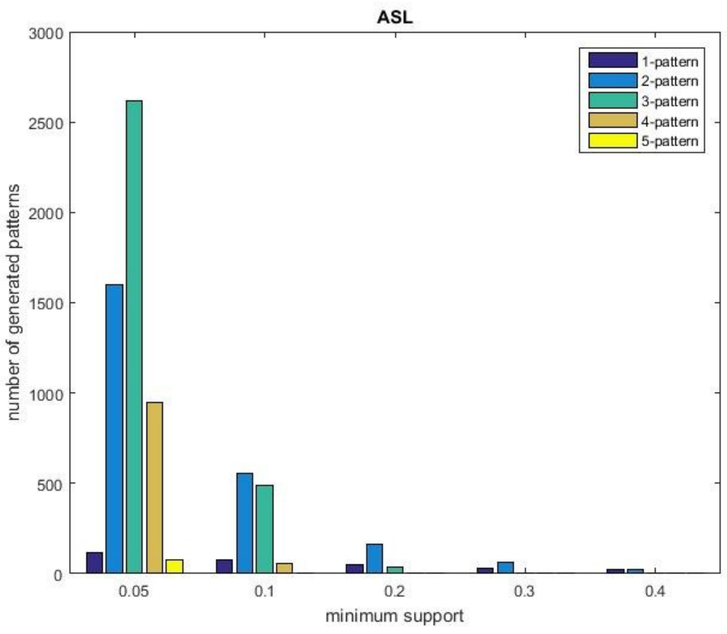


FIGURE 5.6: Distribution of Generated Patterns for ASL Dataset

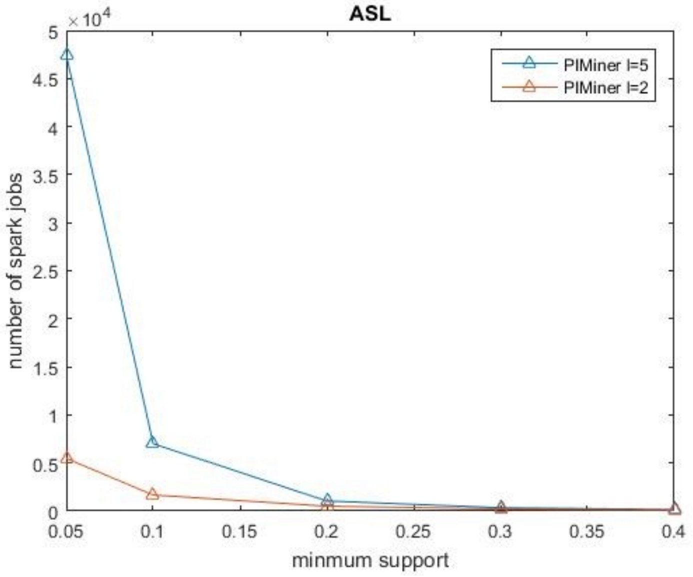


FIGURE 5.7: Spark Jobs Analysis for ASL Dataset

RDD jobs. The number of spark jobs is plotted against support values 0.4, 0.3, 0.2, 0.1 and 0.05. The two different values of the maximum length of the frequent temporal pattern allowed i.e. L = 2 and L = 5.

When the maximum length of the frequent temporal patter is increased the number of spark jobs are increased by large factors for small support which is more dependent on the dataset. The number of spark jobs is proportional to the total frequent patterns mined. As evident from Figure 5.6 as the number of frequent patterns is increased exponentially, so is the rate of the number of spark jobs increased.

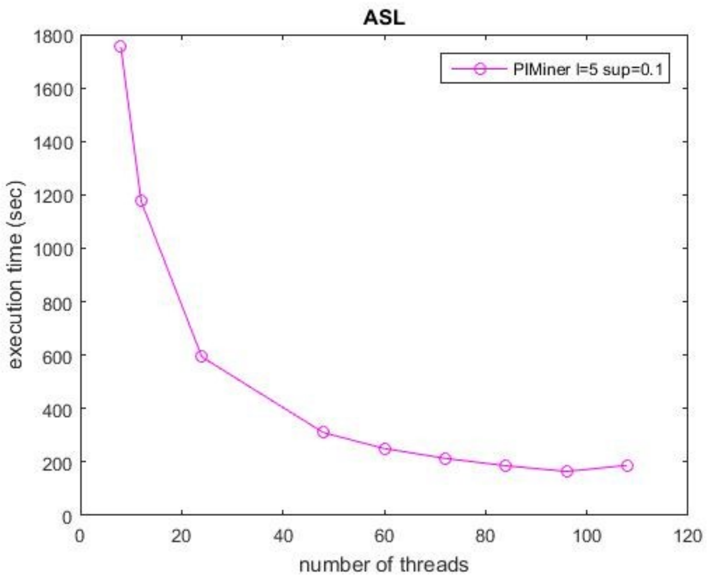


FIGURE 5.8: Threads Influence on ASL Dataset

The effect of the number of threads to execution time is shown in Figure 5.8. The permissible maximum value of the length of the frequent pattern is L = 5. The execution time is evaluated for the support value of sup = 0.1. When the number of threads are increased the execution time is reduced exponentially. Also observe that when number of threads are increased beyond threshold value the execution time again starts increasing. This is because as the number of threads grow the considerable amount of time is spent managing and scheduling all the threads. This threshold value is determined based on the system configuration. This value approximately converges to the total number of cores available across the whole distributed system.

5.4 Scalibility Tests

The real scalability is observed when the size of the dataset is considerably larger. Then the data parallelism is more evident. The first synthetic dataset used for this purpose is S100kC40E1k.

TABLE 5.4: S100kC40E1k Dataset Aspects

Parameters	Values
Number of sequences (S)	730
Average length of sequence (C)	31
Number of events (E)	234
Average duration of interval (D)	13
Total number of intervals (I)	22727

This dataset contains 100000 sequences, the average length of the sequence is 40 and number of events is 1000. The size of dataset S100kC40E1k is 45.4MB. The detailed specifications of the dataset is shown in the Table 5.4.

The execution time of our PIMiner algorithm is stacked against the serial algorithm TPMiner in Figure 5.9. Here to observe how data parallelism is achieved we even use a single thread for comparison. Here the parameter *b* is the block size in MBs in which the database is divided. As the block size value is decreased the number of blocks is increased and more parallelism is evident. The different parameters for comparison are:

1. TPMiner
2. PIMiner with number of threads (t) = 1 and block size (b) = 4MB
3. PIMiner with number of threads (t) = 1 and block size (b) = 2MB
4. PIMiner with number of threads (t) = 2 and block size (b) = 2MB

Analogous to the task parallelism using number of threads in Section 5.2, as the block size decreases the time of execution is decreases, but beyond certain threshold time starts to increase. This threshold value can be obtained by hit and trial, and is dependent on the dataset. Beyond this threshold the cost of maintaining the number of blocks per computing node would be more than the data parallelism achieve by partitioning the database in first place.

The different support values used for different configurations of our algorithm are 0.3, 0.35, 0.4, 0.45 and 0.5. The maximum length of the frequent pattern L is set to L = 1. Hence this

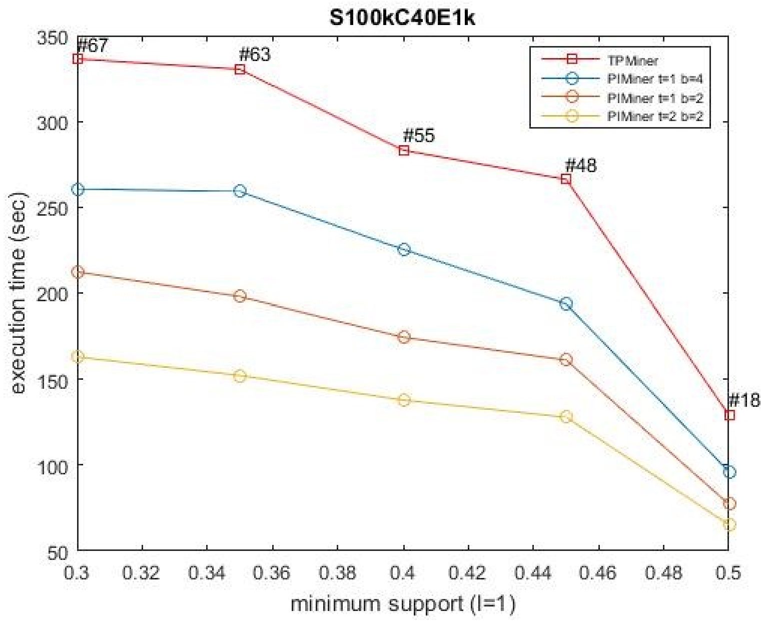


FIGURE 5.9: Execution Time Analysis for S100kC40E1k

eventually finds the number of frequent temporal events in the database. As the value of L is increased the difference between our algorithm and serial algorithm would be such drastic that it would be possible to plot the values in same graph.

It can be deduced from Figure 5.9 that when the value of block size is 4MB, PIMiner takes order of hundreds of seconds less time when support value is small. Also when the block size is further reduced to 2MB the execution time graph is further reduced. Finally, when the number of threads is increased by just 1 for the same block size of 2MB, the execution time is further reduced. The number of frequent interval events is show with prefix '#' before the value in the graph for reference.

Therefore, from this observation it can be safely inferred that to achieve the maximum possible performance we need to tweak both the parameters the number of threads to control the task parallelism as well as the block size to control the data parallelism. This sweet spot can be obtained by some hit and trial and is more dependent on the dataset as well as the distributed system on which the algorithm is executed.

The total number of patterns generated for different support values for dataset S100kC40E1k

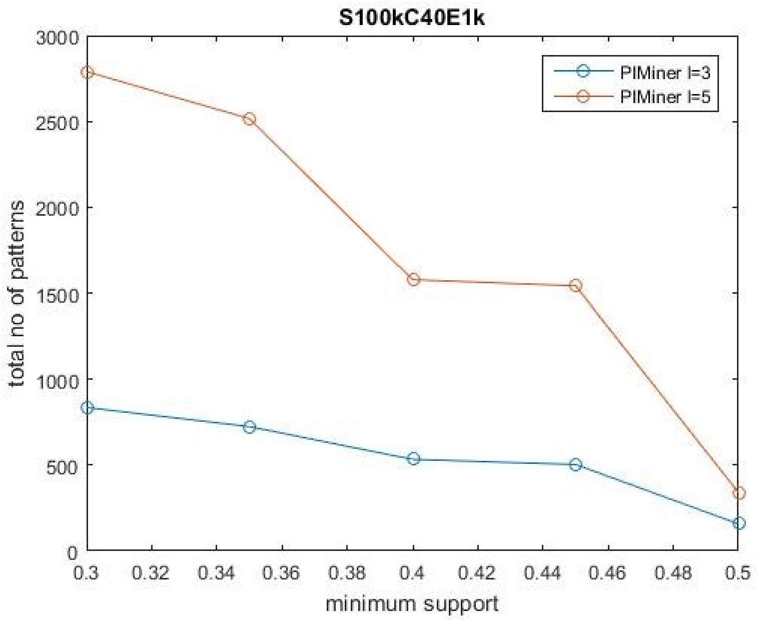


FIGURE 5.10: Patterns Generated on Maximum Length of Pattern on S100kC40E1k

is shown in Figure 5.10. We have used two different values of maximum length permissible for frequent temporal pattern viz. L = 3 and L = 5.

The values of support for which the total number of patterns are shown is 0.3, 0.35, 0.4, 0.45 and 0.5. As expected with decreasing support the total number of patterns are increasing. Similarly when maximum length of the allowed frequent pattern is increased from to 5, there is again a surge in number of temporal patterns as expected, as new 4-length and 5-length patterns are reported.

The large scale dataset used for the final evaluation are:

1. S100000C40E1k
2. S200000C40E1k
3. S300000C40E1k
4. S400000C40E1k
5. S500000C40E1k

TABLE 5.5: C40E1k Dataset Aspects

Parameters	**S100k**	**S200k**	**S300k**	**S400k**	**S500k**
Number of sequences (S)	100000	200000	300000	400000	500000
Average length of sequence (C)	40	40	40	40	40
Number of events (E)	1000	1000	1000	1000	1000
Total number of intervals (I)	3997520	7999442	12004982	16002474	20002495

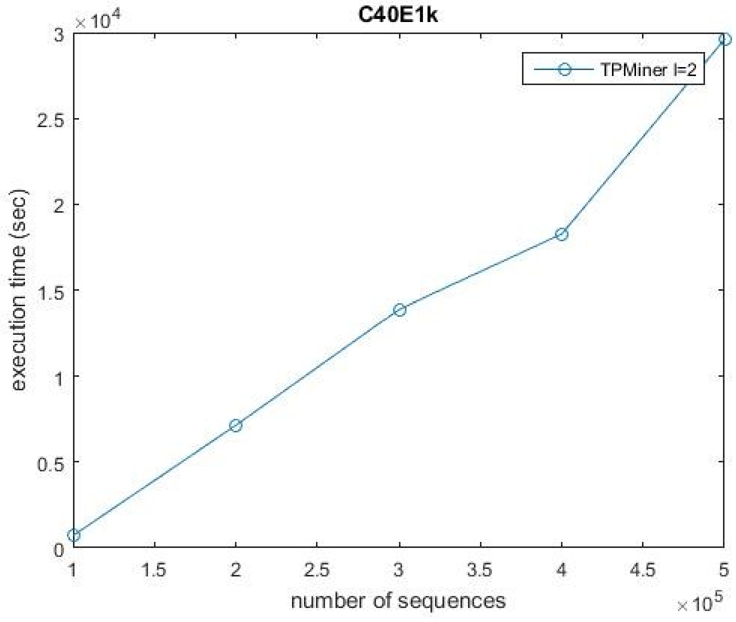


FIGURE 5.11: TPMiner Performance Analysis on C40E1k

These five datasets vary in number of total sequences. The average length of sequence is same for all the datasets C = 40 and number of total temporal events is also same for all datasets E = 1000. The number of sequences are 100000, 200000, 300000, 400000, and 500000. The size of the databases are 45.4MB, 91.0MB, 136.9MB, 185.5MB and 232.4MB respectively. The detailed specification of the datasets is shown in Table 5.5 mainly illustrating the differences among the datasets.

The execution time for the TPMiner algorithm is shown in Figure 5.11 with maximum allowed length of temporal pattern is L = 2 for the number of sequences for the five datasets.

The execution time of PIMiner for same maximum length of frequent pattern I = 2 is plotted in Figure 5.12. Note that here we have tried to achieve maximum performance by tweaking

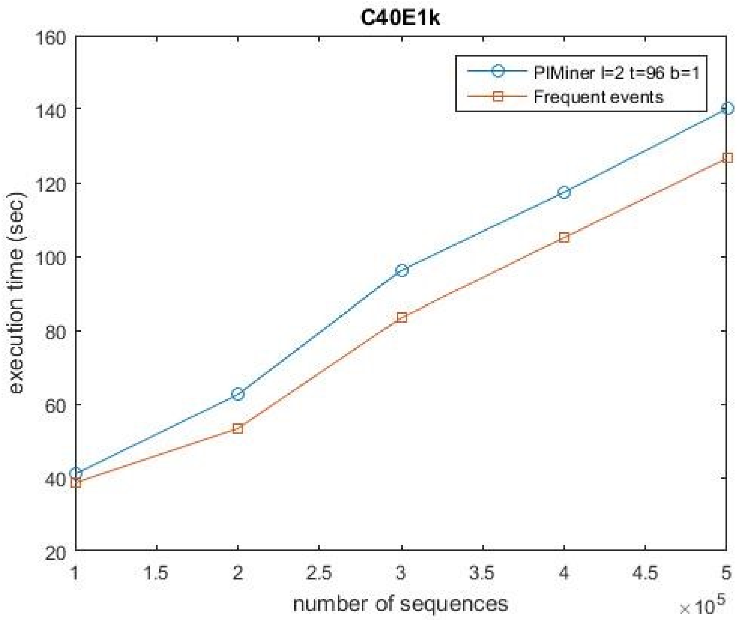


FIGURE 5.12: PIMiner Performance Analysis on C40E1k

both the parameters several number of threads and block size to t = 96 and b = 1MB respectively. Notice the considerable execution time improvement in two cases where TPMiner is taking time in order of several thousand seconds and PIMiner is taking only in few hundred seconds. For example, when tuned with optimal parameters PIMiner is 211 times faster than TPMiner for dataset S500kC40E1k. This is because the size of dataset is considerably large that data parallelism is quite influential. The only limiting factor is now the underlying distributed system.

5.5 Inference

In this Chapter, we conducted several experiments to demonstrate the effectiveness of the *PIMiner* algorithm. We generated synthetic dataset for this purpose and apply the algorithm on ASL real dataset to validate the practicability of mined patterns. Both task parallelism and data data parallelism is demonstrated separately as well as together. The task parallelism is achieved by maintaining a set number of user threads while data parallelism is controlled by the block size for the dataset. It can be inferred that both techniques individually performs better than traditional serial approach. Also, we observed that there is a threshold to how many user threads we are maintaining in the implementation and how small can the block size be. After this threshold controlling operations are increased so much that scheduling processes become more expensive than computation processes. The last scalability tests show that *PIMiner* is very much effective when both approaches employed jointly.

CONCLUSION AND FUTURE WORK

6

Mining of frequent temporal patterns on interval based data is a crucial subfield of data mining. The complex relations among intervals make it inherently arduous to design efficient distributed temporal pattern mining algorithm. All the current pattern mining algorithms on interval-based events are sequential in nature. They cannot scale to large data set which cannot be stored in single memory. The various parallel techniques proposed in mining frequent patterns are carried out on instantaneous events, i.e. point-based events and not on interval events. In this work, we have developed a new algorithm *PIMiner* which runs in distributed fashion, which is a first such attempt on interval pattern mining. *PIMiner* employs both task and data parallelism techniques and is therefore highly scalable. The performance study of *PIMiner* indicate that the algorithm is much efficient compared to serial temporal pattern mining algorithms as it is evident when stacked against the state-of-the-art TPMiner algorithm for mining temporal patterns.

Future Work

There are several possible interesting issues that could be addressed in future. To the best of my knowledge, two main issues that are not addressed anywhere regarding parallel interval pattern mining are as follows:

1. Creating a hybrid of the proposed PIMiner and serial temporal pattern mining algorithms may even perform better. For instance when the size of the projected database is small enough then switch to serial algorithms, that is, when the inherent cost of incorporating data and task parallelism is expensive enough.

2. No techniques is developed to utilize GPU computations in finding the frequent patterns in interval data. Although GPU based techniques for instantaneous events are presented which can help in tackling this issue. In order to achieve this an entirely new techniques need to be devised so that there is some portion of algorithm which can run in parallel on GPU cores.

BIBLIOGRAPHY

[1] Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843.

[2] Batal, I., Cooper, G. F., Fradkin, D., Harrison Jr, J., Moerchen, F., and Hauskrecht, M. (2016). An efficient pattern mining approach for event detection in multivariate temporal data. *Knowledge and information systems*, 46(1):115–150.

[3] Chen, Y. C., Peng, W. C., and Lee, S. Y. (2015). Mining temporal patterns in time interval-based data. *IEEE Transactions on Knowledge and Data Engineering*, 27(12):3318–3331.

[4] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[5] Höppner, F. and Klawonn, F. (2001). Finding informative rules in interval sequences. In *International Symposium on Intelligent Data Analysis*, pages 125–134. Springer.

[6] Javed, A. and Khokhar, A. (2004). Frequent pattern mining on message passing multiprocessor systems. *Distributed and Parallel Databases*, 16(3):321–334.

[7] Kam, P.-s. and Fu, A. W.-C. (2000). Discovering temporal patterns for interval-based events. In *International Conference on Data Warehousing and Knowledge discovery*, pages 317–326. Springer.

[8] Li, H., Wang, Y., Zhang, D., Zhang, M., and Chang, E. Y. (2008). Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems*, pages 107–114. ACM.

[9] Liang, Y.-h. and Wu, S.-y. (2015). Sequence-growth: A scalable and effective frequent itemset mining algorithm for big data based on mapreduce framework. In *Big Data (BigData Congress), 2015 IEEE International Congress on*, pages 393–400. IEEE.

[10] Lin, M.-Y., Lee, P.-Y., and Hsueh, S.-C. (2012). Apriori-based frequent itemset mining algorithms on mapreduce. In *Proceedings of the 6th international conference on ubiquitous information management and communication*, page 76. ACM.

[11] Liu, L., Li, E., Zhang, Y., and Tang, Z. (2007). Optimization of frequent itemset mining on multiple-core processor. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1275–1285. VLDB Endowment.

[12] Papapetrou, P., Kollios, G., Sclaroff, S., and Gunopulos, D. (2005). Discovering frequent arrangements of temporal intervals. In *Data Mining, Fifth IEEE International Conference on*, pages 8–pp. IEEE.

[13] Patel, D., Hsu, W., and Lee, M. L. (2008). Mining relationships among interval-based events for classification. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 393–404. ACM.

[14] Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., and Hsu, M.-C. (2004). Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on knowledge and data engineering*, 16(11):1424–1440.

[15] Ruan, G., Zhang, H., and Plale, B. (2014). Parallel and quantitative sequential pattern mining for large-scale interval-based temporal data. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 32–39. IEEE.

[16] Sadasivam, R. and Duraiswamy, K. (2013). Efficient approach to discover interval-based sequential patterns. *Journal of Computer Science*, 9(2):225.

[17] Schuster, A. and Wolff, R. (2004). Communication-efficient distributed mining of association rules. *Data mining and knowledge discovery*, 8(2):171–196.

[18] Srikant, R. and Agrawal, R. (1996). Mining sequential patterns: Generalizations and performance improvements. *Advances in Database Technology—EDBT'96*, pages 1–17.

[19] Tsay, Y.-J., Hsu, T.-J., and Yu, J.-R. (2009). Fiut: A new method for mining frequent itemsets. *Information Sciences*, 179(11):1724–1737.

[20] Villafane, R., Hua, K. A., Tran, D., and Maulik, B. (2000). Knowledge discovery from series of interval events. *Journal of Intelligent Information Systems*, 15(1):71–89.

[21] Wu, S. Y. and Chen, Y. L. (2007). Mining nonambiguous temporal patterns for interval-based events. *IEEE Transactions on Knowledge and Data Engineering*, 19(6):742–758.

[22] Wu, S.-Y. and Chen, Y.-L. (2009). Discovering hybrid temporal patterns from sequences consisting of point-and interval-based events. *Data & Knowledge Engineering*, 68(11):1309–1330.

[23] Xun, Y., Zhang, J., and Qin, X. (2016). Fidoop: Parallel mining of frequent itemsets using mapreduce. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 46(3):313–325.

[24] Yi-Cheng, L. and Chen, Julia Tzu-Ya, S.-Y. (2016). Incremental mining of temporal patterns in interval-based database. *Knowledge and Information Systems*, 46(2):423–448.