The Open University of Israel
Department of Mathematics and Computer Science

IPFIM - Incremental Parallel Frequent Itemsets Mining

Thesis submitted as partial fulfillment of the requirements
towards an M.Sc. degree in Computer Science
The Open University of Israel
Department of Mathematics and Computer Science

By
Lev Kuznetsov

Prepared under the supervision of **Prof. Ehud Gudes**

March 2022

Acknowledgements

I would like to thank Prof. Gudes, my instructor, for his time and guidance in making this thesis. I would also use this acknowledgement to thanks The Open University of Israel for providing the opportunity to ... And last but certainly not least, I would like to thank my family and my wife Jenny for supporting me during that time.

Abstract

The frequent itemsets mining (FIM) problem has been around pretty much since the definition of the term 'data' in the previous century. Whenever there is a collection of data, one of the basic analysis we would like to perform, is finding relations within the data. One of those basic 'relations', is to find all the sets of data that appear together in an important frequency (usually greater than a predefined threshold). The process of finding such items is called Mining, and thus the term - Frequent Itemsets Mining (FIM). Frequent itemsets can later reveal association rules and relations between variables. This research area in data science is applied to domains such as recommender systems (e.g. what are the set of items usually ordered together), bioinformatics (e.g. what are the genes coexpressed in a given condition), decision making, clustering, website navigation and many more.

Many algorithms were developed during time to find frequent itemsets in a database, and they are mostly focused on Apriori [2] and FP-Growth [14] techniques.

As the access to online resources grew, and thus the size of the databases, the need for incremental updates (changing only states dependent on the input) and parallel algorithms grew as well.

This work focuses on using tree based structure for parallel and incremental mining. We will present 2 new algorithms for mining frequent itemsets using trees, IPFIM, IPFIM-Improved and Set-Cover IPFIM. We will compare the existing incremental and parallel tree based algorithms and evaluate them using Spark (we also discuss the differences with using Hadoop and Map-Reduce in section **paragraph 2.1.3**).

Contents

List of Figures

List of Tables

1 Introduction

1.1 Introduction - Add in to what this work contributes

Frequent itemset (FIS) mining is an important building block of machine learning. As the access to online resources grew, so does the size of the databases and today's databases' sizes go far beyond capabilities of a single machine. The need to provide better performance has grown and platforms for parallel computation, and the frameworks who support them, also became main stream. In this thesis, we will describe an approach for dealing with an incrementally updated database, while avoiding candidate generation, and performing a single DB scan.

We will discuss previous related work, describe current technology and review implementation, usage and performance.

Contribution

This work contributions are:

1. Thorough review, experiment and comparison of existing parallel and incremental algorithms for FIS mining based on tree based structures, and evaluate their performance experimentally with various datasets and parameters
2. Novel developed and implemented algorithm for mining FIS.

Thesis Structure

The structure of this work is as follows: **chapter 2** will present the relevant definitions and algorithms with examples, **chapter 3** and **chapter 4** will

present the proposed algorithms - IPFIM and IPFIM-Improved, **chapter 5** will review how we implemented and tested the different algorithms and scenarios, **chapter 6** will present the results and last are **chapter 7** and **chapter 8** that will summaries the thesis and outline future work.

2 Background And Related Work

2.1 Background

2.1.1 Frequent Itemset

Given a set L = $\{i_1, ..., i_n\}$ called items. A set $P = \{i_1, ..., i_k\} \subseteq L$, where $k \in [1, n]$ is called a pattern (or an itemset), or a k-itemsets if it contains k items.

A transaction t = (t_{id},Y) is a tuple where t_{id} is a transaction-id and Y is a pattern. If P $\subseteq$ Y, it is said that t contains P or P occurs in t.

A transaction database DB over L is a set of transactions and $|DB|$ is the size of DB, i.e. the total number of transactions in DB. The support of a pattern P in a DB, denoted as Sup(P), is the number of transactions in DB that contain P.

A pattern is called a frequent pattern if its support is no less than a user given minimum support threshold minsup ϑ, with $0 \leq \vartheta \leq |DB|$.

The frequent pattern mining problem, given a ϑ and a DB, is to discover the complete set of frequent patterns in a DB having support no less than ϑ.

Max and Closed Frequent Itemset

Later on in section 2.1.3, the used benchmarks are evaluating algorithms which perform Closed FIM. To better understand those definitions [10] provides good illustrations and explanations.

Max Frequent Itemset It is a frequent itemsets for which none of its immediate supersets are frequent. In figure 2.1, the lattice is divided into two

groups, red dashed line serves as the dermarcation, the itemsets above the line that are blank are frequent itemsets and the blue ones below the red dashed line are infrequent.

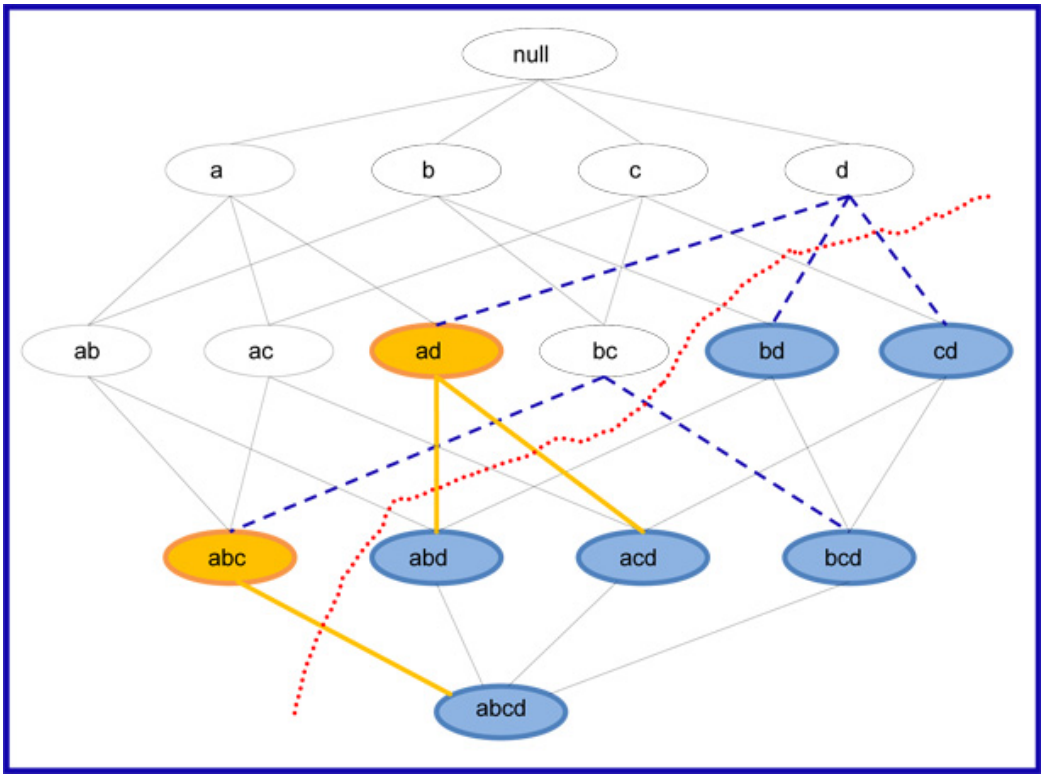


Figure 2.1: Maximal Frequent itemsets Illustration

1. In order to find the maximal frequent itemset, you first identify the frequent itemsets at the border namely **d, bc, ad** and **abc**.

2. Then identify their immediate supersets, the supersets for **d, bc** are characterized by the blue dashed line and if you trace the lattice you notice that for **d**, there are three supersets and one of them, **ad** is frequent and this can't be maximal frequent, for **bc** there are two supersets namely **abc** and **bcd, abc** is frequent and so **bc** is NOT maximal frequent.

3. The supersets for **ad** and **abc** are characterized by a solid orange line, the superset for **abc** is **abcd** and being that it is infrequent, **abc** is maximal frequent. For **ad**, there are two supersets **abd** and **acd**, both of them are infrequent and so **ad** is also maximal frequent.

Closed Frequent itemsets An itemsets is closed in a data set if there exists no superset that has the same support count as this original itemset. Figure 2.2 shows the maximal, closed and frequent itemsets. The itemsets that are circled with blue are the frequent itemsets. The itemsets that are circled with the thick blue are the closed frequent itemsets. The itemsets that are circled with the thick blue and have the yellow fill are the maximal frequent itemsets. In order to determine which of the frequent itemsets are closed, all you have to do is check to see if they have the same support as their supersets, if they do they are not closed. For example **ad** is a frequent itemsets but has the same support as **abd** so it is NOT a closed frequent itemset; **c** on the other hand is a closed frequent itemsets because all of its supersets, **ac, bc**, and **cd** have supports that are less than 3. As we can see there are a total of 9 frequent itemsets, 4 of them are closed frequent itemsets and out of these 4, 2 of them are maximal frequent itemsets. This brings us to the relationship between the three representations of frequent itemsets.

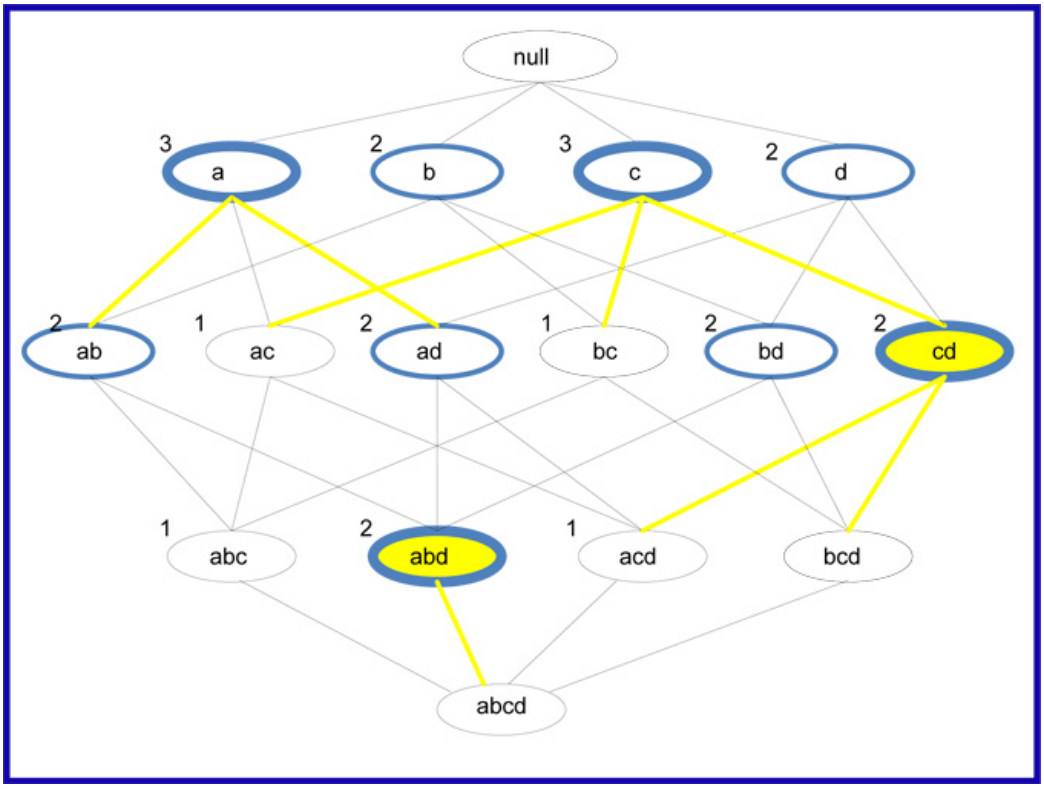


Figure 2.2: Closed and Maximal Frequent itemsets Illustration

Figure 2.3 demonstrates this relationship, between the different FIS groups.

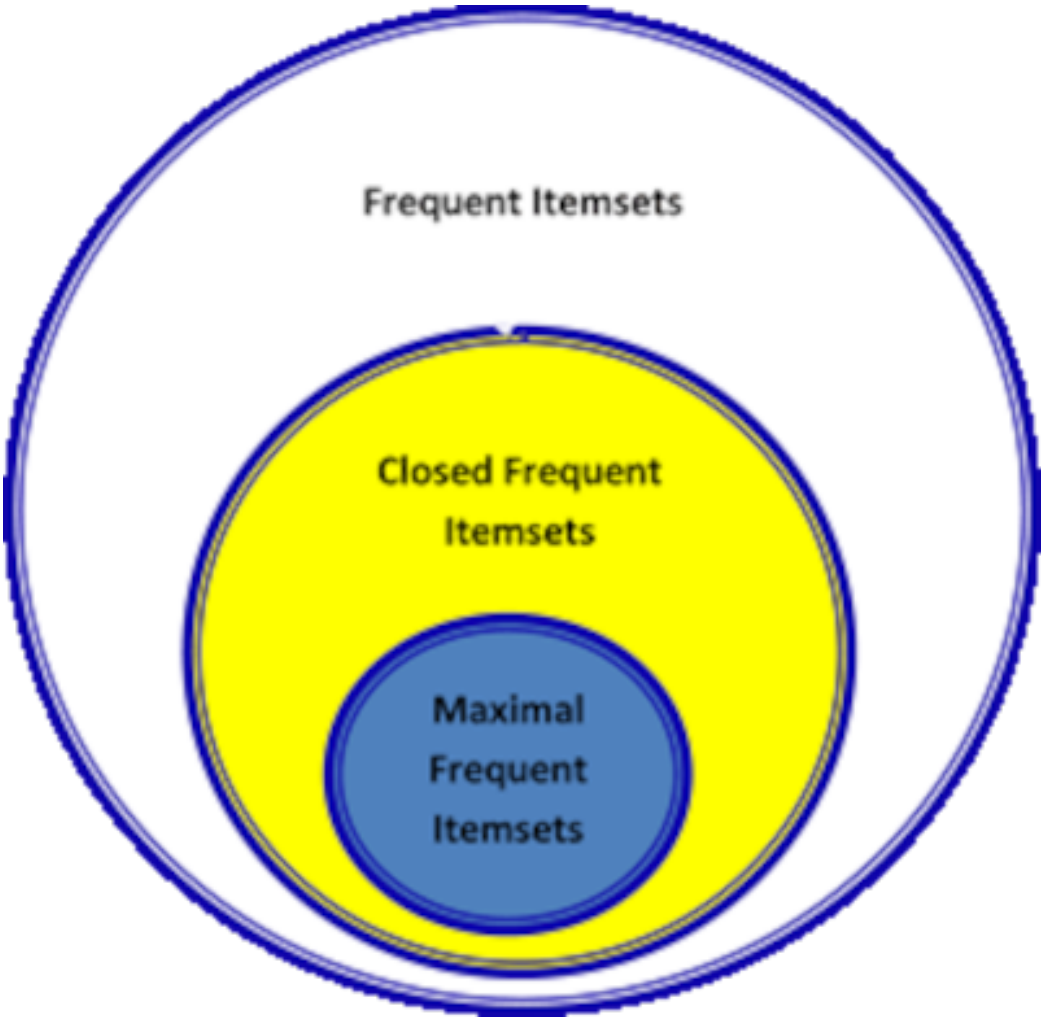


Figure 2.3: Relationship between Frequent itemsets Representations

2.1.2 Common Algorithms For Mining FIS

Apriori One of the earliest and most well known algorithms for mining association rules is the Apriori algorithm [2]. This algorithm is iteratively generating candidates and pruning items with low support at each step. The correctness of this algorithm is based on the lema that if an item of length N is frequent, then all sub patterns must be frequent as well. Using that idea, an early prune of non-frequent itemsets removes many unnecessary candidates in later iterations. An example is provided in **Figure 2.4**. We will not expand further this algorithm, but this algorithm is intuitive and widely used. We would also mention that this algorithm main limita-

tion is the candidate generation at every iteration, where many candidates may not be relevant and this information could have been used in previous stages.

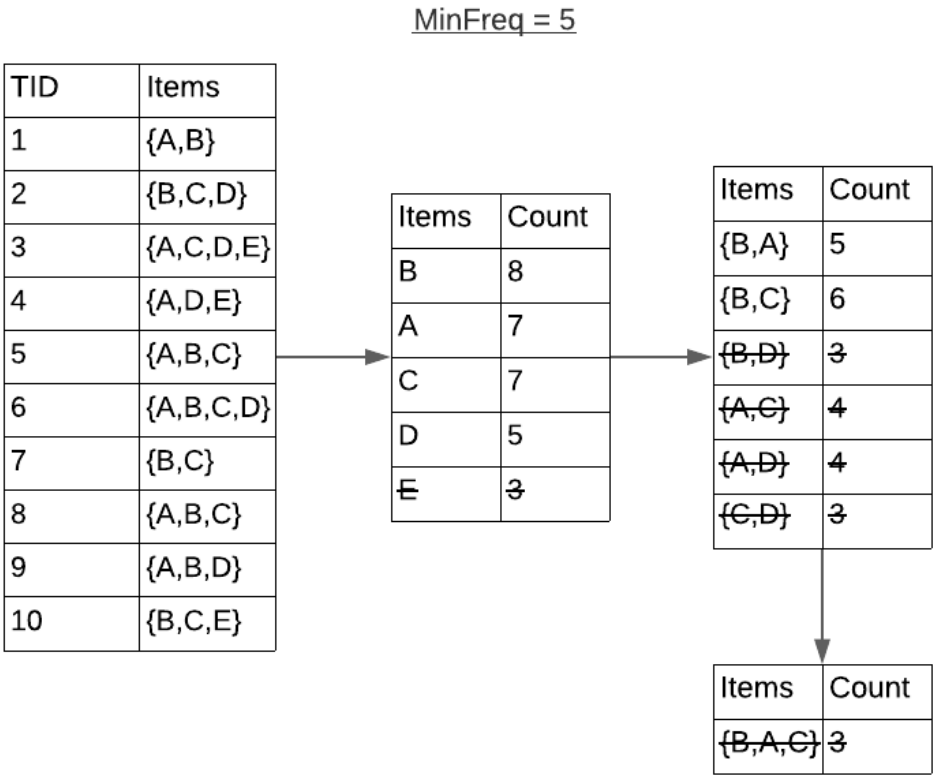


Figure 2.4: Apriori Example

FPGrowth In the year 2000, a tree based solution was introduced, FP-Growth algorithm and the FP-Tree structure [2]. This algorithm removes the need for candidate generation and yields better performance [13].

TODO: Add algo A small example is provided in **Figure 2.5**

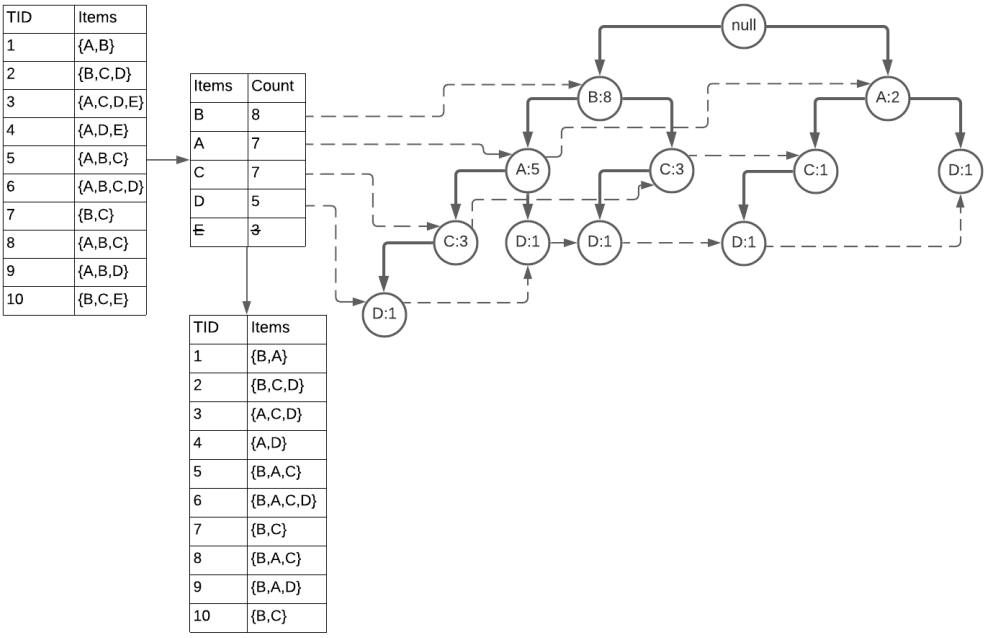


Figure 2.5: FPGrowth example

2.1.3 Parallel Algorithms For Mining FIS

As the processed databases grew, the size went beyond the capabilities of a single machine. The initial algorithms of the map-reduce platform, suggested and alternate paradigm which uses more smaller computations on the account of less larger ones (or one huge single process of the database).

Apache Hadoop Apache Hadoop [4] is a collection of open-source software utilities that facilitates using a network of many computers to solve problems involving massive amounts of data and computation. The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part which is a MapReduce programming model. Hadoop splits files into large blocks and distributes them across nodes in a cluster. It then transfers packaged code into nodes to process the data in parallel.

Apache Spark Is a data processing engine for big data sets. Like Hadoop, Spark splits up large tasks across different nodes. However, it tends to

perform faster than Hadoop and it uses random access memory (RAM) to cache and process data instead of a file system. This enables Spark to handle use cases that Hadoop cannot.

Spark is a Hadoop enhancement to MapReduce. The primary difference between Spark and MapReduce is that Spark processes and retains data in memory for subsequent steps, whereas MapReduce processes data on disk. As a result, for smaller workloads, Spark's data processing speeds are up to 100x faster than MapReduce [11].

The work by Daniele Apiletti et el. [9] is focusing on comparing different frequent itemsets mining algorithms between the Apache Hadoop [4] and Apache Spark [8].

This work is performing extensive evaluation on synthetic and real world datasets and testing execution time, load balancing, and communication costs between 4 parallel itemsets mining algorithm.

The participating algorithms are:

SparkVsHadoop.1 The Parallel FP-Growth implementation provided in Hadoop Mahout 0.9 [3]

SparkVsHadoop.2 The Parallel FP-Growth implementation provided in MLlib for Spark 1.3.0 [5]

SparkVsHadoop.3 The June 2015 implementation of BigFIM [22]

SparkVsHadoop.4 The version of DistEclat downloaded from [22] on September 2015

In our work we are using as the infrastructure, the PFP implementation of the MLLib [5] library in Spark, which is one of the evaluated algorithms in the article [9]. A detailed explanation of PFP will be described in **subsection 2.2.2**.

BigFIM and DistEclat are not relevant to this work.

On page 27, the article mentions that except the Spark MLLib PFP [5] algorithm, all other implementations are mining closed itemsets, and thus to obtain the same output, the execution times of Mahout PFP, BigFIM and DistEclat may increase with respect to MLlib PFP.

The evaluations in the paper were done using synthetic and real-world data. The synthetic data and real-world data The results of the paper tested a synthetic and real-world data. In **Figure 2.6** and **Figure 2.7** it can be seen that for low minSupport values, **item SparkVsHadoop.2**

has the better performance, so the implementation and comparison using Spark [8] is very relevant, as others were tested on classic mapReduce.

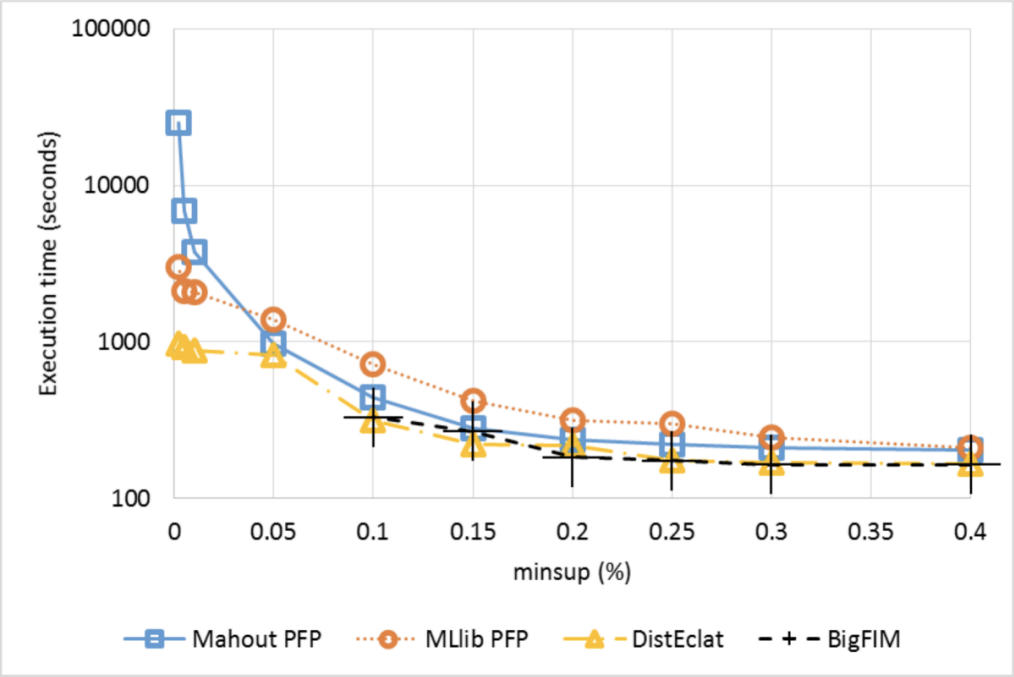


Figure 2.6: Execution time for different min-sup values, average transaction length 10

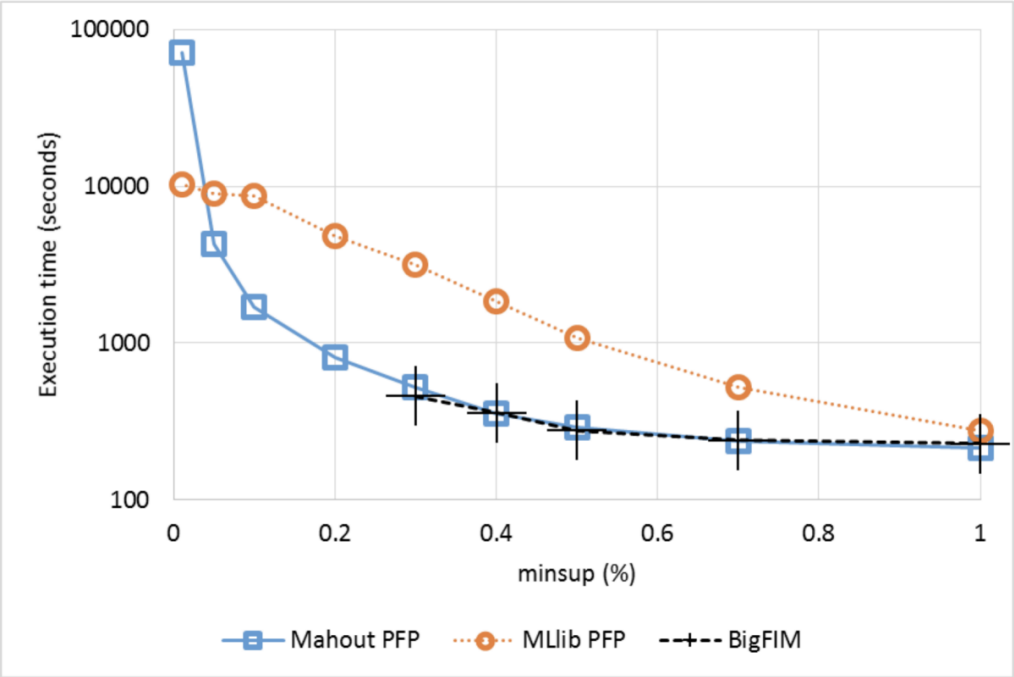


Figure 2.7: Execution time for different min-sup values , average transaction length 30

2.2 RELATED WORK

2.2.1 Incremental Frequent Itemsets Mining

The definition of an Incremental update, is to recompute outputs which depend on the incoming inputs only, without recomputing the whole data.

The challenge while performing incremental updates for frequent items mining, is a non consistent frequency order. Several algorithms such as AFPIM [15], EFPIM [18] and FUFP-tree [12] are keeping an updated frequency based trees, by reordering branches where frequency has changed. Each with its technique.

1. AFPIM [15] described in **paragraph 2.2.1**
2. EFPIM [18]
3. FUFP-tree [12]

[TODO: Describe each algo in 2-3 sentences]

Canonical-Tree The work of [16] presented a Canonical Tree (CanTree) which preserves the frequency descending structure as in FP Growth mining, by relying on a predefined order, which will not affect the tree structure and correctness.

The predefined order, creates some nice properties, as described below:

1. Items are arranged according to a canonical order, which is a fixed global ordering.
2. The ordering of items is unaffected by the changes in frequency caused by incremental updating.
3. The frequency of a node in the CanTree is at least as high as the sum of frequencies of all its children.

Since CanTree preserves the same feature as the FP-Tree for mining FIS, the mining is done in the same fasion as the original FP-Growth algorithm.

An example of a CanTree is presented in **Table 2.1** and **Figure 2.8**

Table 2.1: Consider the following database:

	TID	**Contents**
Original database (DB)	t_1	a, d, b, g, e, c
	t_2	d, f, b, a, e
	t_3	a
	t_4	d, a, b
The first group of insertions (db1)	t_5	a, c, b
	t_6	c, b, a, e
The second group of insertions (db2)	t_7	a, b, c
	t_8	a, b, c

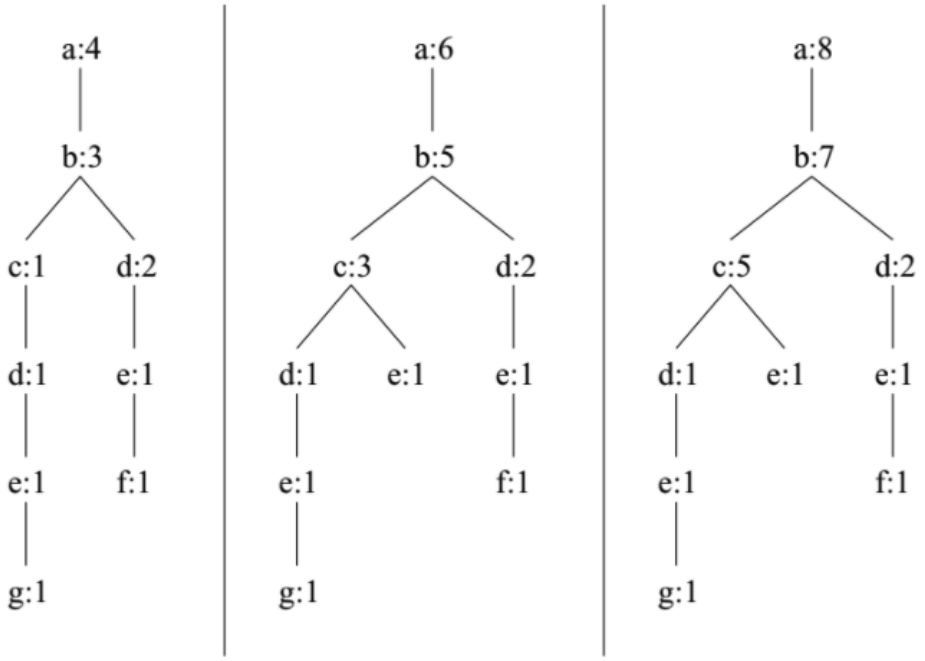


Figure 2.8: The CanTree after each group of transactions is added

CP-Tree The work of [21] proposes an improvement to the CanTree algorithm, called CompactPattern-Tree, and discusses the memory and computation limitations of CanTree for large incremental Databases. The issues are caused due to un-efficient tree structure, and CP-Tree is proposing an improvement by periodically (using a proposed guideline) updating the order of the construction literals list (l-list) and rebuilding the trees. As mentioned in the original article and as seen by our experiments, the CanTree and CP-Tree have similar tree size, and the difference for our test cases was about 10% in tree sizes. However as seen in our results, using semi-frequency based order, improves the mining results by 10X and more for smaller minSupport values [TODO: Add graphs].

AFPIM - Adjusting FP-tree Structure for Incremental Mining The work by [15], is using a technique of pre-min support threshold for defining a pre-frequent itemset. The rational behind this approach is that to identify those items within every iteration, does not require another scan of the previous data as it is already part of the intermediate tree. Table 2.2 demonstrates the different scenarios and the possibilities when new iter-

ations are added, $\cup D$ is the union of all iterations on DB. $\cup D$ needs to be scanned to re-construct the FP-tree of $\cup D$ only if there exist any item belonging to case 4. For cases 5 or 6, which are in-frequent in DB, these items can be ignored because the corresponding itemsets are not frequent in $\cup D$. For the correctness of the algorithm, 3 operations were defined:

Table 2.2: AFPIM cases

case	in DB	in $\cup D$
Case 1	X is a frequent or pre-frequent item	X is a frequent item
Case 2	X is a frequent or pre-frequent item	X is a pre-frequent item
Case 3	X is a frequent or pre-frequent item	X is an infrequent item
Case 4	X is an infrequent item	X is a frequent item
Case 5	X is an infrequent item	X is a pre-frequent item
Case 6	X is an infrequent item	X is an infrequent item

AFPIM.1 Remove nodes of infrequent items - Direct remove of the infrequent nodes and reattach sons to parent.

AFPIM.2 Adjusting the path of nodes - a bubble-sort for maintaining frequency order.

AFPIM.3 Insert or remove data from FP-tree - After the tree is scanned and re-sorted based on **AFPIM.2**, add/remove new sorted data.

AFPIM example Let the original database, DB, be illustrated in 2.3. The minimum support is 0.2 and the pre-minimum support is 0.15. Scan DB once to collect the set of frequent or pre-frequent items: A:2, B:6, C:5, D:3, E:4, F:7, G:1, and H:1 (:indicates the support count). The items with support counts no less than 2 (i.e. $13 \times 0.15 = 1.95$) are frequent or pre-frequent items in DB. Thus, A, B, C, D, E, and F are frequent items in DB. After sorting all the frequent or pre-frequent items in support descending order, the result is F:7, B:6, C:5, E:4, D:3, and A:2. Accordingly, the constructed FP-tree of DB is shown as 2.9. Then 5 transactions are inserted into and 3 transactions are removed from DB, where the transaction data are shown as 2.4 and 2.5. The support counts of all the items in db^+ and db^- are listed

in 2.6. For each item X in $\cup D$, minimum support of X can be obtained by a simple computation. The result is A:2, B:9, C:5, D:7, E:6, F:8, G:1, and H:2. In new database $\cup D$, a pre-frequent or frequent item must have support counts no less than 3 (i.e. (13+5-3)$\times$0.15=2.25). Therefore, the frequent or pre-frequent 1-itemsets in $\cup D$, shown in frequency descending order, are B:9, F:8, D:7, E:6, and C:5. As shown in table 2.6 , A is not a frequent or pre-frequent item in $\cup D$,. Thus, the nodes representing A are removed from the FP-tree. The resultant FP-tree is shown as figure 2.10. We will not discuss in detail the other operations, as in IPFIM improved algorithm we use a semi-frequency order, and no need to perform **AFPIM.2**, as for correctness of the mining, a rescan and rebuild of the FP-Tree for case 4 in 2.2 is required. However we skipped this phase for the evaluations.

Table 2.3: Original Database DB for AFPIM:

TID	Items	Frequent or pre-frequent items (ordered in frequency descending order)
1	BDEF	FBED
2	F	F
3	ABEF	FBEA
4	CH	C
5	BF	FB
6	B	B
7	ABEF	FBEA
8	CG	C
9	BF	FB
10	CDE	CED
11	F	F
12	CD	CD
13	C	C

Table 2.4: db^+

TID	Itemset
14	BCDEF
15	BDEF
16	BCDG
17	BD
18	DH

Table 2.5: db^-

TID	Itemset
5	BF
8	CG
1	CD

Table 2.6: $\cup D$ Frequency

Database	Items	Frequent or pre-frequent items (ordered in frequency descending order)
DB	A:2, B:6, C:5, D:3, E:4, F:7, G:1, H:1	F:7, B:6, C:5, E:4, D:3, A:2
db^+	B:4,C:2,D:5,E:2,F:2,G:1,H:1	-
db^-	B:1,C:2,D:1,F:1,G:1	-
$\cup D$	A:2, B:9, C:5, D:7, E:6, F:8, G:1, H:2	B:9, F:8, D:7, E:6, C:5

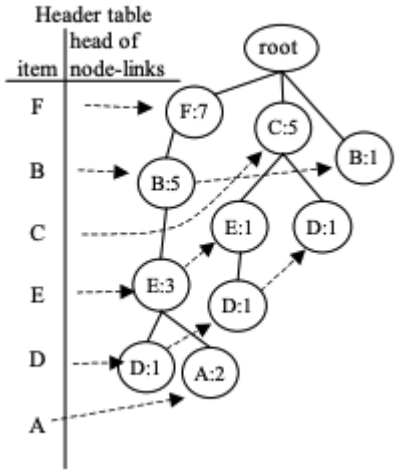


Figure 2.9: FP-tree of AFPIM DB

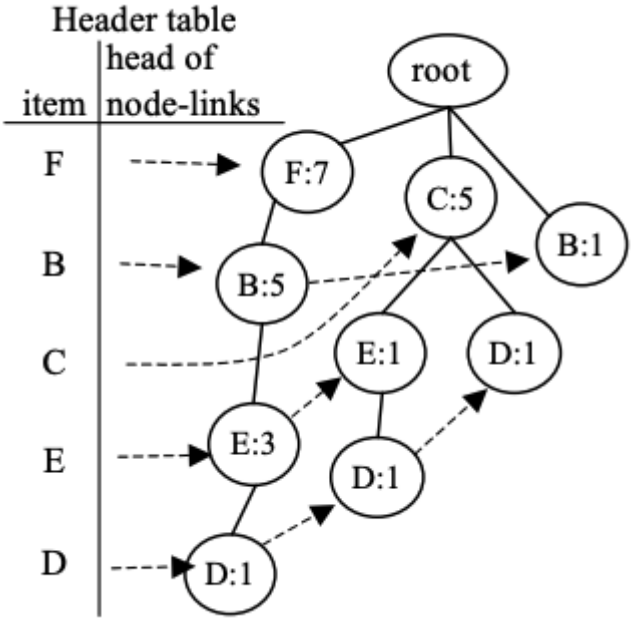


Figure 2.10: FP-tree After delete of $\cup D$

2.2.2 Parallel Frequent Itemsets mining

The difficulty in parallelizing FP-growth is to distribute iterations to parallel trees while still allowing correct mining. PFP [17] is solving this by dividing the DB transactions to independent trees using a Group-List, where every group consists of subgroup of the original items, and redistributing iterations in the DB based on this list. PFP [17] has the following MapReduce stages:

Step 1: Calculate the global frequency list F-list, by MapReduce "Work Count" manner.

Step 2: In the second job, the the map will have the following functionality:

1. Sort transactions based on F-list.
2. Replace items in a transaction with the appropriate group id mapped transactions.

The reducer here will build the trees in a parallel manner, based on the group id of the mapping stage.

Step 3: In the final mapping stage, every mapper will project the sub tree from a 1-item-length frequent itemset of the group, where the reducers will recursively mine those sub-trees. The parallelization in that case, can be at most equal to the number of items in the database.

A more detailed description of PFP [17] is described here **Algorithm 0**:

Algorithm 1 Highlevel description of the PFP-Growth algorithm

procedure PFP-GROWTH
  $F-List \leftarrow$ Find global frequency list
  $G-list \leftarrow$ Define a Group items
  for each *transaction* T_i *in* DB **do**
    $t_i \leftarrow$ order by F-List frequency
    $G-hashed-list_i \leftarrow$ replace every element a_j *in* t_i with Hash(g), where $a_j \in g$ And $g \in G-list$
    for each Hash(g) $\in G-hashed-list$ **do**
      $L \leftarrow$ find its right-most location in t_i
      Output key'=g; value'=$t_i[0] \ldots t_i[L]$
    end for
  end for
  Group by key' = g
  For each group g, build appropriate tree
  For each group g, mine the generated tree.
end procedure

To better understand the distribution of transactions between groups in PFP, an example is provided in **Table 2.7**. This example shows the initial state of raw transactions, sorting them frequency based, and distributing based on a G-List of single item per group - {C},{E},{B},{A}. Last lines demonstrate the output at line 9 **Algorithm 0** of every transaction.

Table 2.7: A simple example of distributed FP-Growth:

	TID	**Items**
Original database (DB)	t_1	A, B
	t_2	E, C
	t_3	E, A, C
	t_4	E, C
	t_5	E, C
	t_6	B
	t_7	A, C
	t_8	E, D, C
	t_9	E, B
	t_{10}	E, B
	t_{11}	E, A, C
	t_{12}	E, C
	t_{13}	B, D, C
F-List:	A: 4, E: 9, B: 5, C: 9, D: 2	
Support=4	Sorted transactions =¿ [C,E,B,A]	
Sorted and Filtered	t_1	B,A
	t_2	C,E
	t_3	C, E, A
	t_4	C,E
	t_5	C,E
	t_6	B
	t_7	C, A
	t_8	C,E
	t_9	E, B
	t_{10}	E, B
	t_{11}	C, E, A
	t_{12}	C,E
	t_{13}	C, B
G-List	{C},{E},{B},{A}	
$g \in G - List$	Transactions	FIS
{C}	t_2,t_3,t_4,t_5,t_7,t_8,t_{11},t_{12},t_{13}	[C]
{E}	t_2,t_3,t_4,t_5,t_8,t_{11},t_{12}	[C,E]
	t_9,t_{10}	[E]
{B}	t_1,t_6	[B]
	t_9,t_{10}	[E,B]
	t_{13}	[C,B]
{A}	t_1	[B,A]
	t_3,t_{11}	[C,E,A]
	t_7	[C,A]

2.2.3 Incremental and Parallel Frequent itemsets mining

Combining the previous 2 sections, yields an algorithm that does not rely on frequency order and uses parallelism advantages for computations of FIS. The drawbacks are also drawn from the 2 algorithms - large memory consumption for saving all items and recursively calculating FIS. As we will show later in the Improvments section, using an approach similar to [14] and maintaining a pre-min support, together with using a semi-freq-order as in [21], will significantly improve memory and mining runtime results.

2.2.4 Song et al.

A paper by Song et el. [20] proposes 2 techniques for building and mining frequent itemsets - IncBuildingPFP and IncMiningPFP. IncBuildingPFP presents a parallel model based on CanTree that supports incremental mining. This approach is similar to IPFIM and presented in **section 3.1**. It is important to mention, that our work was developed independently and only after implementation and testing, this work was discovered and and as comparison and evaluation.

IncMiningPFP

As discussed by Song et el. [20] (and analysed later in this article as well), using CanTree as is, will result in memory and time limitation for relatively medium datasets (¿1M) even with large clusters (¿100G RAM). IncMiningPFP solves the problem of mining by constructing FP-Growth tree for shards with new incremental items. For other shards, the data is taken from cache.

IncMiningPFP consists of the following steps:

Step 1: Group items G-list

Step 2: For the base case, for each shard, save the FIS and construct full FP-Trees to save all transactions.

Step 3: For every Shard:

1. calculate and save frequency list per shard group -¿ F-list
2. find 1-size FIS, extract paths from FP-Trees that contain only those items and build a FP-Growth tree.

3. Extract full FIS from FP-Growth tree and save items in shard cache.

Step 4: For every iteration dD, devide based on G-List and send to the appropriate shards (similar to same as the distribution stage at **Algorithm 0**).

Step 5: For every shard:

1. Recalculate F-list
2. If got new items from dD, update tree with new values
3. If this shard has added transactions, recalculate new FIS in similar way to initial stage and update shard cache.
4. Return Caches FIS.

Step 6: Reduce all previous FIS, similar to PFP.

In this paper, although the IPFIM and IncBuildingPFP are similar, in **section 4.1**, we present a different technique based on the CPTree [21] and AFPIM [15] approach.

Song et el. [20] was also tested using classic map-reduce, while here we test the performance using Spark. Spark programs iteratively run about 100 times faster than Hadoop in-memory, and 10 times faster on disk [8].

Set-Cover

The set cover problem is a classical question in combinatorics, computer science, operations research, and complexity theory. It is one of Karp's 21 NP-complete problems shown to be NP-complete in 1972 [23].

Formal Definition Given a set system $\Sigma = (X, S)$, where $S = \{s_1, ..., s_n\}$, compute a set $C \subseteq \{1, ..., n\}$ of minimum cardinality such that $X = \bigcup_{i \in C} s_i$

For Example, given a set of elements $\{1, 2, ..., n\}$ and a collection X of m sets whose union equals to S, the set cover problem is to identify the smallest sub-collection of X whose union equals to S. For example, consider the universe $S = \{1, 2, 3, 4, 5\}$ and the collection of sets $C = \{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$. Clearly the union of C is S. However, we can cover all of the elements with the following, smaller number of sets: $X = \{\{1, 2, 3\}, \{4, 5\}\}$. For our hypothesis, the Set-Cover-Ipfim algorithm [TODO:Ref set-cover-ipfim],we

wanted to check whether the use of of optimal groups based on frequent itemsets, will result in better computation time, as it is potentially to optimize the distribution. A more detailed explanation is in [TODO].

Since the classic solution is NP-Hard, we used a naive greedy implementation, which requires $O(mn^2)$ in 0, where m is the item size, and n is number of items.

Algorithm 2 Greedy set-cover implementation

```
1: procedure GREEDY-SET-COVER(X,S)
2:     U ← X // U stores the uncovered items
3:     C ← empty // C stores the sets of the cover
4:     while U is nonempty do
5:         select s[i] in S that covers the most elements of U
6:         add i to C
7:         remove the elements of s[i] from U
8:     end while
9:     return C
10: end procedure
```

3 IPFIM Algorithm

3.1 IPFIM - Incremental Parallel Frequent Itemsets Mining

The implementation of this algorithm strongly depends on PFP [17]. To support incremental tree updates, we are using a predefined comparison function to arrange the items insertion order, as used in CanTree [16].

3.1.1 IPFIM Outline

The combination of the previously mentioned algorithms will provide an incremental and parallel algorithm for mining FIS. The highlevel algorithm is presented in 0, while the inner update of every iteration is in 0.

Algorithm 3 IPFIM

```
1: procedure IPFIM(minSupport, iterationsArray[],numIterations, sortFunction,partitioner)
2:     canTrees ← RDD[|partitioner|] of empty CanTree objects
3:     fisArray ← array[numIterations]
4:     for i ← 0 ; i<numIterations ; i++ do
5:         canTrees ← IPFIMIteration(iterationsArray[i], canTrees, sortFunction, partitioner)
6:         fisArray[i] ← mineCanTrees(canTrees, minSupport)  ▷ Same as in FPGrowth
7:     end for
8:     return fisArray
9: end procedure
```

For every iteration there the following map/reduce jobs:

Algorithm 4 IPFIMIteration

```
1: procedure IPFIMITERATION(data,canTrees,sortFunction,partitioner)
2:     sortedTransactions ← order data by sortFunction
3:     partitionedTransactions ← map sortedTransactions to key'=g; value'=t_i[0]...t_i[L]
   ▷ As in PFP
4:     canTrees ← Reduce partitionedTransactions and update trees
5:     return canTrees
6: end procedure
```

1. Map: Read and sort data
2. Map: Split based on the partitioner
3. Reduce: Add to proper CanTree object in group

Followed by the mining map:

1. Map: every partition, 1-length fis to its projected recursive tree, and output FIS

IPFIM Example

Figure 3.1 shows an example for a 2 partition calculation of CanTrees based on the partition function of $\{a6, a4, a2\} \rightarrow 0$ and $\{a5, a3, a1\} \rightarrow 1$.

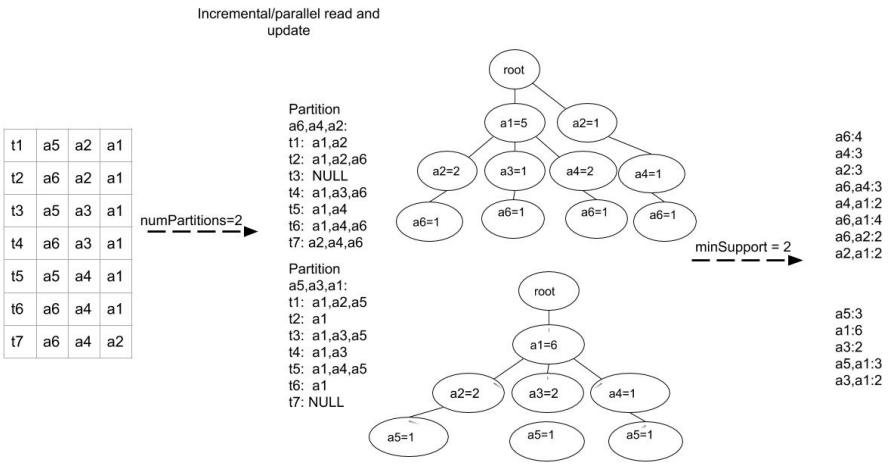


Figure 3.1: IPFIM example

3.1.2 Correctness

The correctness of IPFIM comes directly from the correctness of the 2 combined algorithms. The proof is pretty simple by using contradiction:

1. Assuming that the following item-set of length k, $\{t_i,... ,t_{i+k-1}\}$, became frequent at iteration l, from transaction T_j, but was not part of the reported output.
2. According to **??**, transaction T_j is translated to ¡key'=g; value'=$t_[0]...t_[k]$¿ and added to the CanTree of partition g.
3. If it was not added at this point, PFP is not correct, false ■
4. Otherwise it was added to the CanTree, but was not mined. CanTree is not correct, false ■.

Similar method is used as a proof of the other use cases - a false frequent itemset, a frequent itemset is no longer frequent etc. [**?**]

4 IPFIM Improvements

4.1 Improvements

While developing and testing the algorithm, 2 main obstacles prevented from using larger datasets and smaller minSupport:

1. Memory - Tree size.
2. Computation Time - Tree order.

4.1.1 IPFIM-Improved

To handle these obstacles, 2 techniques were implemented. To handle the computation time, an approach similar to CPTree 2.2.1 was tested, where we defined a semi-frequency order on 1st dataset half, and used it for the rest of the iterations. New items were sorted canonically.

To handle the memory limitation, which is caused by the construction of a large tree, a partial approach of 2.2.1 was used - We added a pre-min support to identify pre-frequent items. For the simplicity of the experiment, items which where not frequent in iteration i, and will become frequent after sum of j iterations, are waived. A trade-off between missing items and tree size can be controlled using the pre-min parameter. Also, as mentioned earlier, since we use a semi-frequent order similar to 2.2.1, no need to perform **AFPIM.2**, only **AFPIM.1** and recalculation for items that are under case 4 in 2.2.

The algorithm is presented in **Algorithm 0**, the main change is in the *IPFIMImprovedIteration* method on lines 2:3, where we first update the global frequency list and clear the items below minMinSupport before adding the transaction to the trees.

Algorithm 5 IPFIMImprovedIteration

```
1: procedure IPFIMIMPROVEDITERATION(data,canTrees,sortFunction,partitioner,minMinSu
2:     freqList ← update with data count
3:     filteredData ← filter from data where itemCount¡minMinSup
4:     sortedAndFilteredTransactions ← order filteredData by sortFunction
5:     partitionedTransactions ← map sortedAndFilteredTransactions to key'=g; value'=t_i[0].
   ▷ As in PFP
6:     canTrees ← Reduce partitionedTransactions and update trees
7:     return canTrees
8: end procedure
```

5 Experiment Preparation

5.1 Experiments Preparation

5.1.1 Datasets

For this article, we used 3 datasets:

Dataset.1 Synthetic datasets of 10M transactions, and 2 magnitudes less of items, average length of 15 items. The dataset was generated using IBM Quest Synthetic Data Generator [1].

Dataset.2 Synthetic datasets of 1M transactions, and 2 magnitudes less of items, average length of 15 items. The dataset was generated using IBM Quest Synthetic Data Generator [1].

Dataset.3 The Kosarak dataset contains 990,000 transactions with 41,270 distinct items and an average transaction length of 8.09 items (click-stream data of a hungarian on-line news portal). This dataset was the largest used by [21].

Every dataset was divided in to 5 iterations, $I_0...I_4$, where I_0 is used as a base case with 50% of the transactions, and the remaining 50% are iterations of 12.5% (e.g. base + 4 iterations).The iterations are saved accordingly as files $f_0..f_4$.

For **PFP**, at iteration i, all files of $0...i$ are re-read and used as the dataset for recalculation of FIS.

5.1.2 Implementation

[TODO: Add that we changed the original code] The implementation was done using Spark [8]. Spark contains an MLlib library, which has an im-

plementation of the PFP algorithm [6]. For our experiments, we leveraged that implementation and the edits required for IPFIM and IPFIM improved where minor:

Step 1: Added support for custom sorting

Step 2: Added support for filtering items below minMin threshold value

Step 3: Added support for logging and statistics

CanTree

For CanTree algorithm implementation, as can be seen From 3.1, running IPFIM with only one group and a lexicographical sorter function, will result in the original CanTree algorithm (adjusted to mapReduce).

Song et al.

For the algorithm developed by Song et al., we had to add a functionality to calculate the intermediate trees. This is also described in 2.2.4. The detailed implementation can be found here [7].

Set-Cover-IPFIM

To support set-cover groups, we used a greedy set-cover algorithm **Algorithm 0** to find the group distribution. This group list is later passed to the partitioning of a the transactions.

5.1.3 Logging and Statistics

To perform our evaluation of execution and memory performance, we are collecting the statistics of run time and the tree-size of trees in different partitions.

5.1.4 Infrastructure

The used hardware is 4 clusters each with 20G memory and 40 cores. The provided infrastructure uses a SparkRDMA Plugin [19]. SparkRDMA provides an improvement of 3X to compared to HDFS [TODO:Add RDMA

banchmarks]. For our experiments, since using same infrastructure, this improvement is not effecting the overall performance differences.

5.1.5 Performance evaluation

For our experiments, we will perform the evaluation of the new proposed algorithms IPFIM, improved-IPFIM and set-cover-IPFIM and compare them to the original algorithms PFP, CanTree as well as the newer Song et al.

The evaluation will review computation time at each iteration, as well as total computation improvement. We also compare the differences in tree sizes for every algorithm and test case. For every iteration we will present the median tree size to better understand the inner structures of the algorithms.

6 Experiments Results

6.1 Experiments and Results

6.1.1 Performance Evaluation

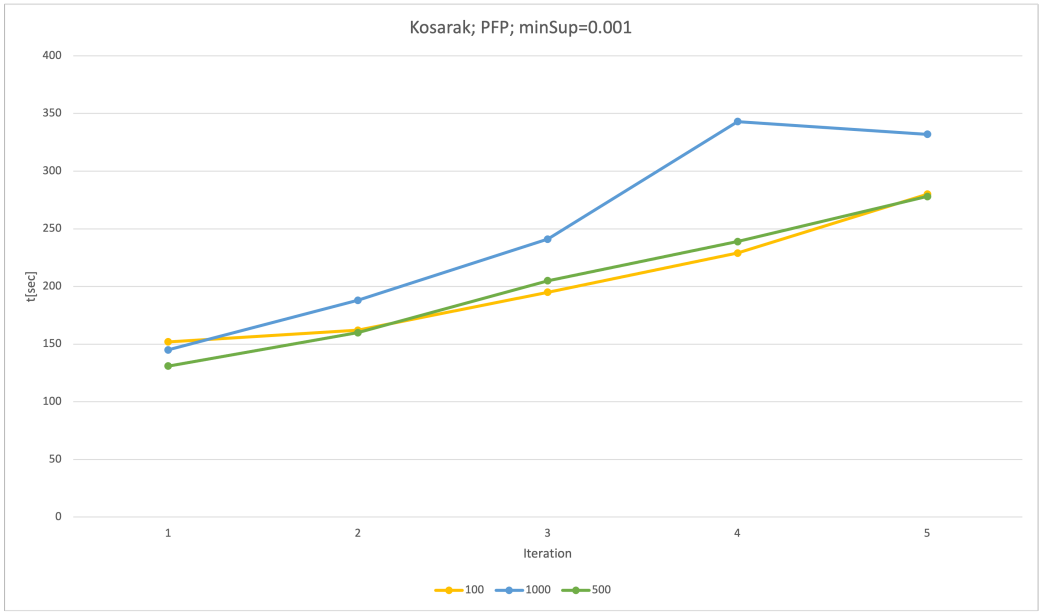
The tests were run on 100, 500 and 1000 partitions, with several support values - the values were adjusted . The following sections will present comparisons of time performance and generated tree-sizes between different scenarios.

[TODO: Add table of experiments summary - e.g. different datasets with the sizes]

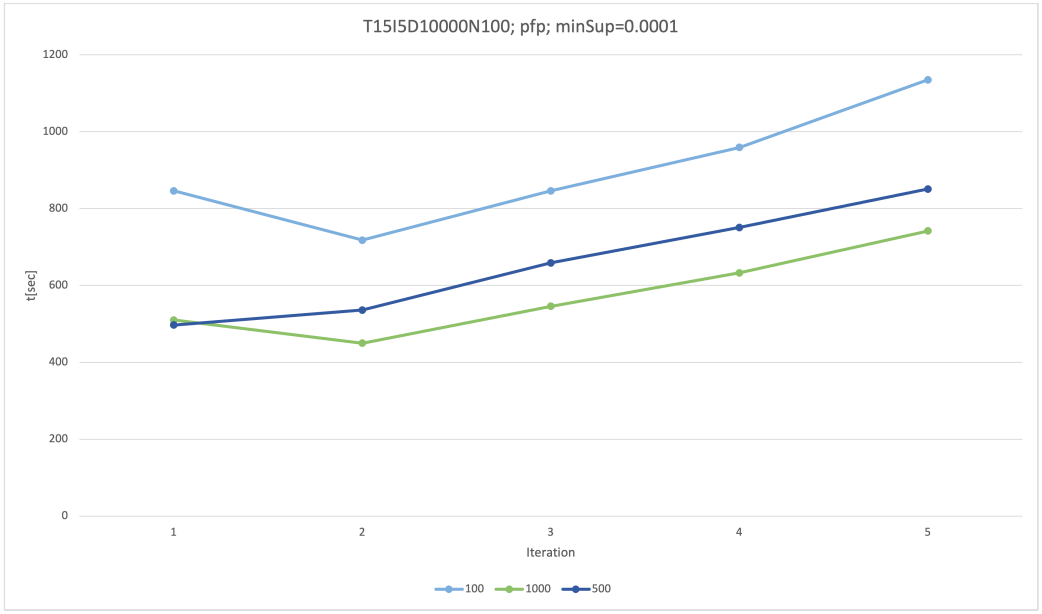
Standalone

We first evaluate each algorithm with different partition sizes to better understand its behaviour.

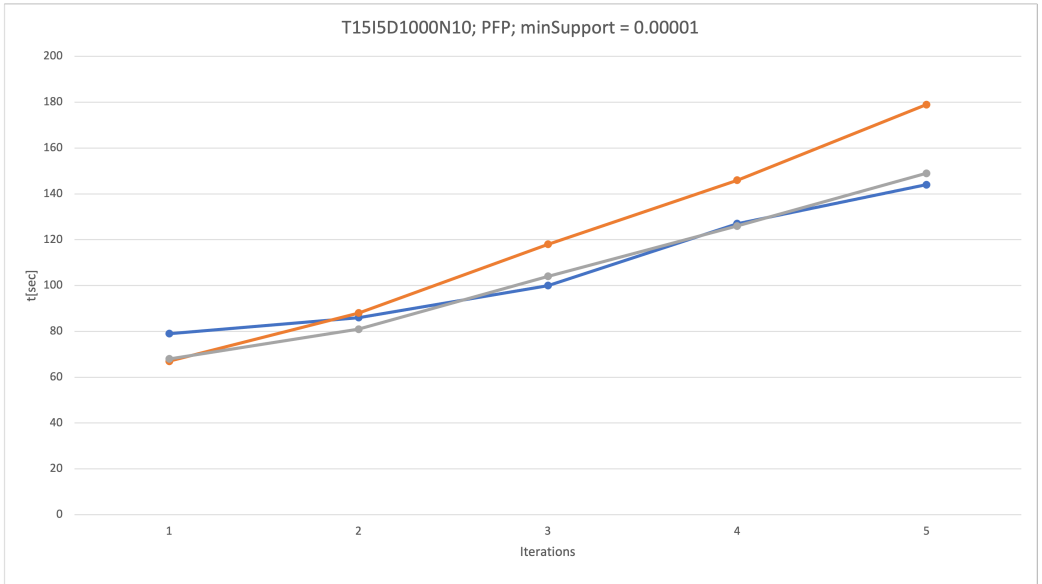
PFP **Figure 6.1a** presents PFP performance on **item Dataset.3** with different partitions sizes. Best performance achieved for 500 partitions. For **item Dataset.1** **Figure 6.1b**, best performance is achieved for 1000 partitions. And **Figure 6.1c** for **item Dataset.2** with support of 0.00001 (more than 10 items are frequent). 100 and 500 have the better performance.



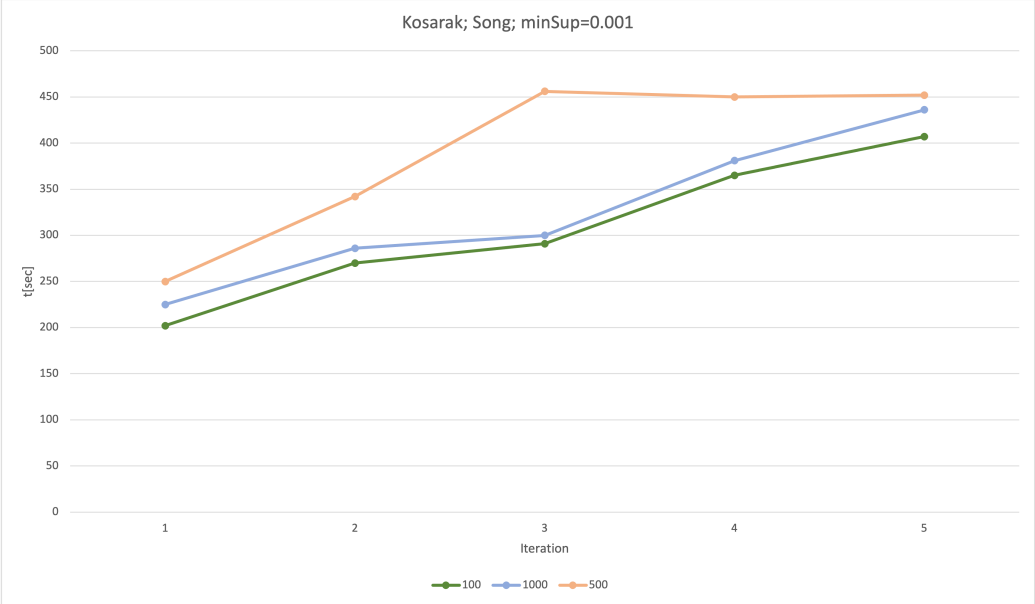
(a) kosarak, minSup = 0.001



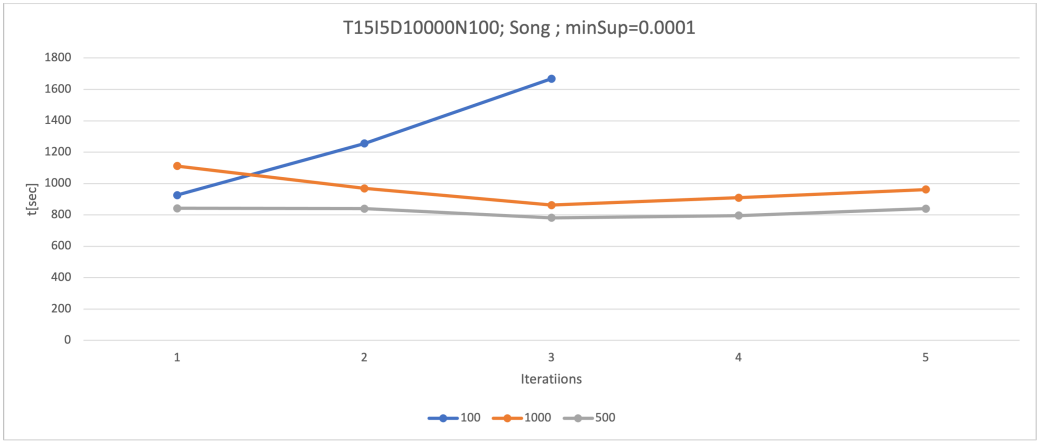
(b) T15I5D10000N100, minSup=0.0001



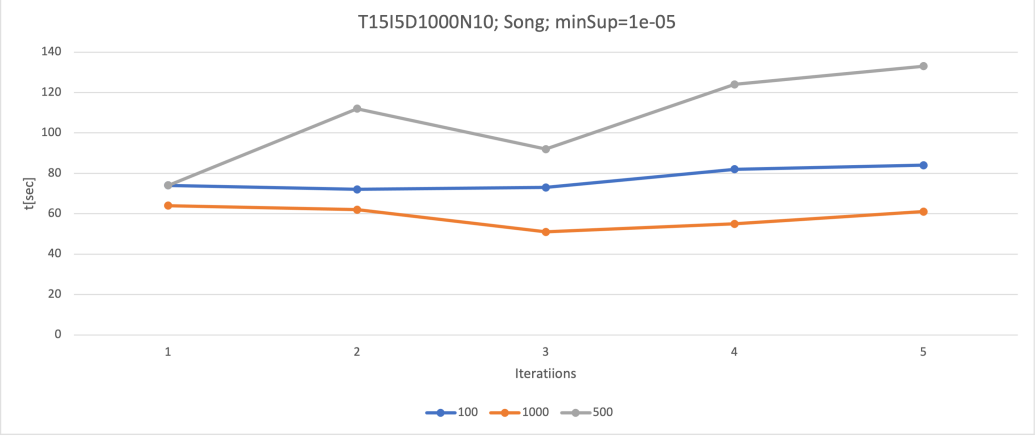
Song **Figure 6.2a** presents Song performance on **item Dataset.3**. Best performance is achieved for 100 and 1000 partitions. **Figure 6.2b** presents Song on **item Dataset.1** with minSupport = 1e-05. The run failed for 100 partitions due to memory error. Best performance achieved for 500 partitions. **Figure 6.2c** presents Song on **item Dataset.2**, best performance achieved for 1000 partitions.



(a) kosarak, minSup = 0.001



(b) T15I5D10000N100 Song, minSup = 0.0001



(c) T15I5D1000N10 Song, minSup = 1e-05

Figure 6.2: SONG 100|1000|500 partitions

IPFIM **Figure 6.3** presents IPFIM on **item Dataset.1**. Best performance achieved on 1000 partitions.

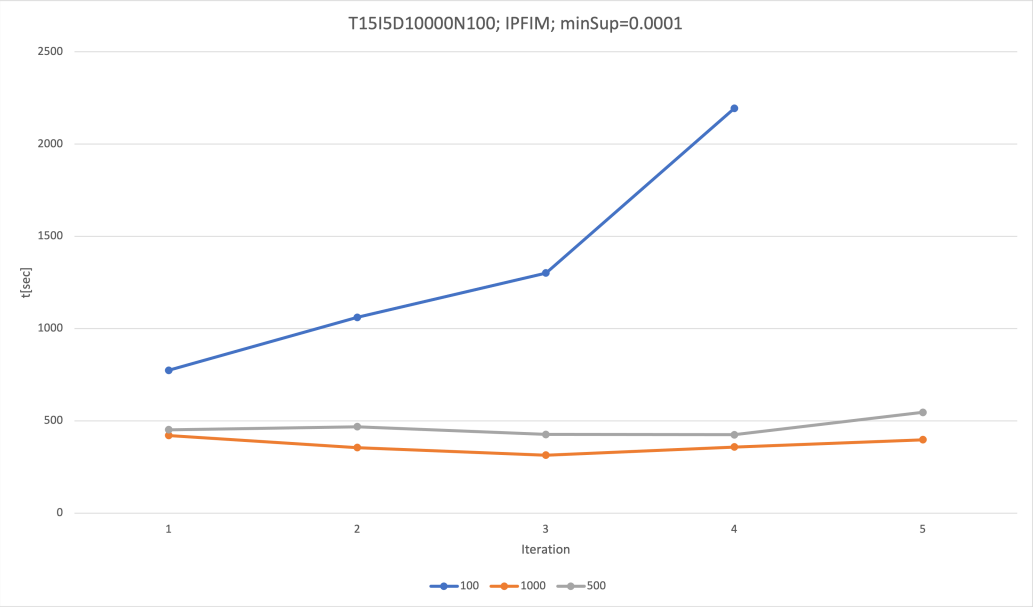
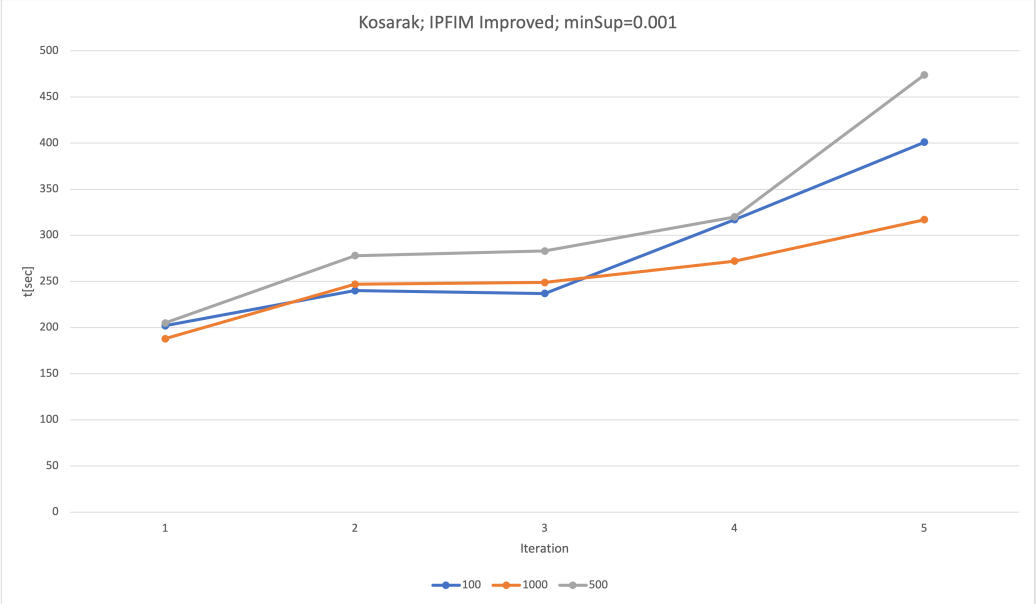
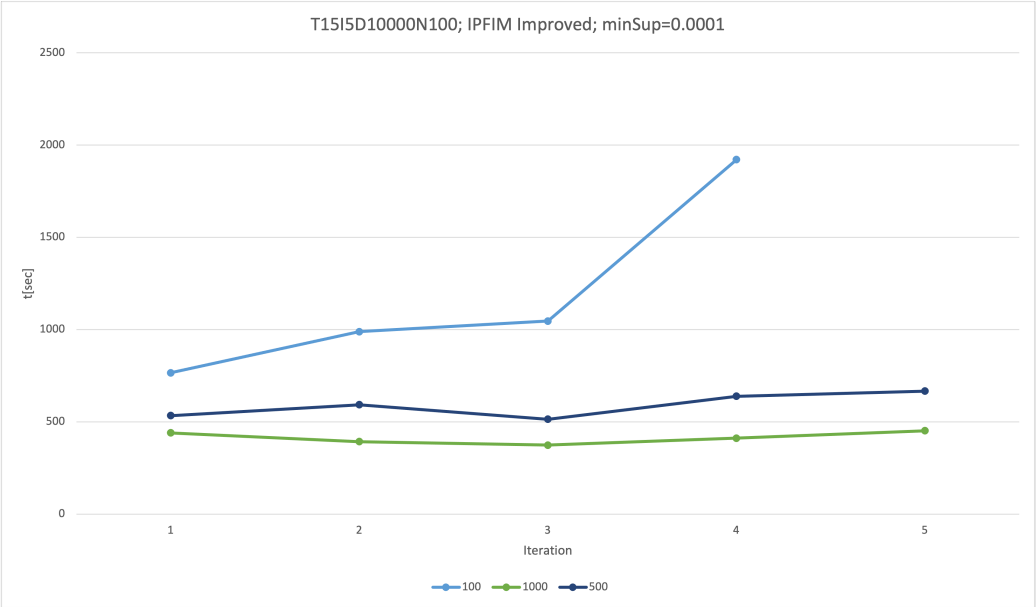


Figure 6.3: T15I5D10000N100, minSup = 0.0001, IPFIM 100|1000|500 partitions

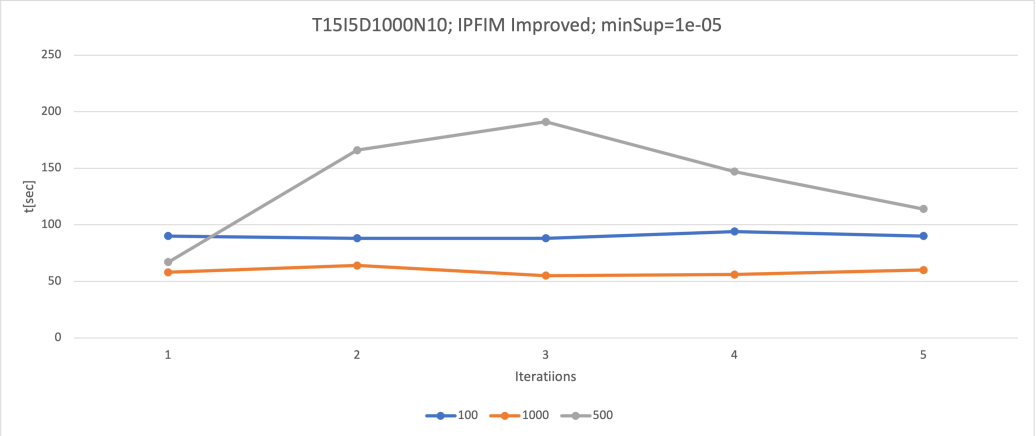
IPFIM Improved **Figure 6.4a** presents IPFIM improved on **item Dataset.3**. Best performance achieved on 1000 partitions. **Figure 6.4b** presents IPFIM improved **item Dataset.1**. Best performance achieved on 1000 partitions. **Figure 6.4c** presents IPFIM improved **item Dataset.2**. Best performance achieved on 1000 partitions.



(a) kosarak, minSup = 0.001, minMinSup = 0.1



(b) T15I5D10000N100, minSup = 0.0001, minMinSup = 0.1

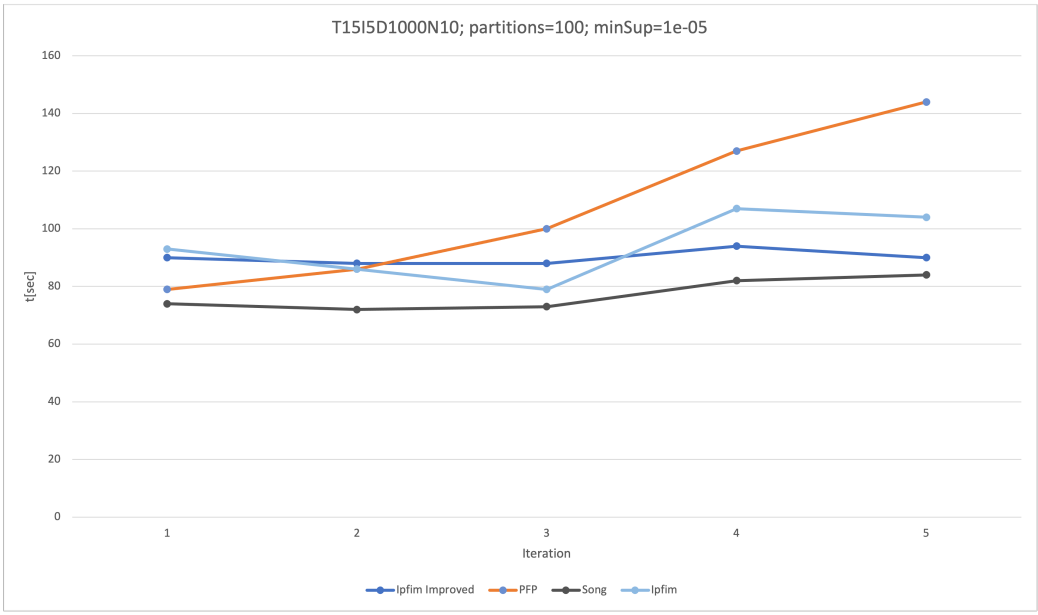


(c) T15I5D1000N10,minSup = 1e-05, minMinSup = 0.1

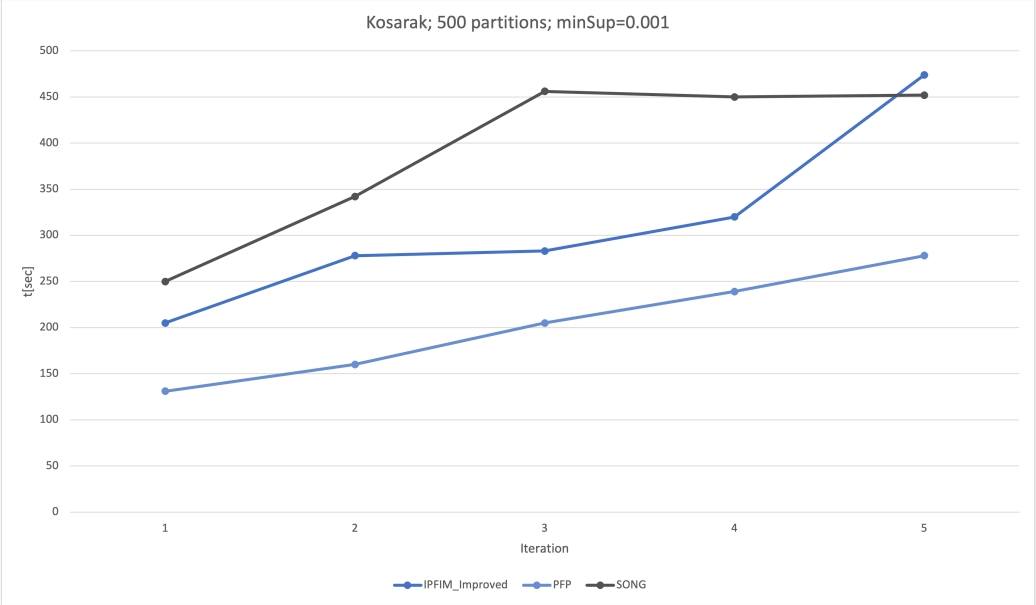
Figure 6.4: IPFIM Improved 100|1000|500 partitions

6.1.2 Comparing performance

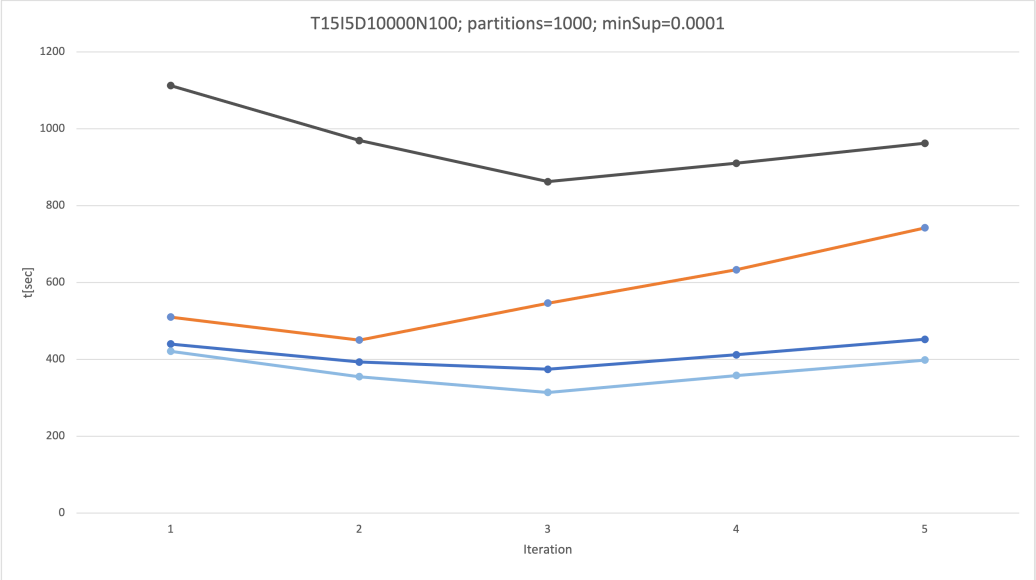
To better compare evaluation, we will determine the best partition for a specific data from the results in **subsubsection 6.1.1**, and appropriate minSupport. **Figure 6.5a** shows that Song has the best performance, while PFP the worst. **Figure 6.5b** shows best performance for PFP and worse for Song and **Figure 6.5c** IPFIM and IPFIM improved has best performance while Song has the worst.



(a) T15I5D1000N10 100 partitions



(b) kosarak, minSup = 0.001, partitions = 500



For the Set-Cover algorithm, **Figure 6.6** shows that the reshuffleing of the partitions using map hash is extremely slow compared to a rand hash function, used in the original algorithm.

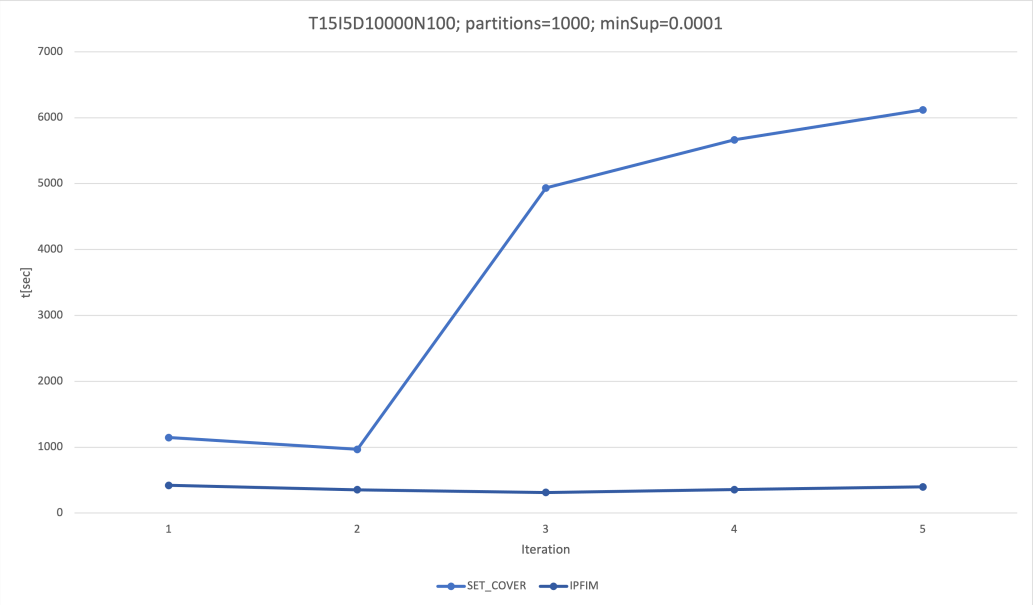
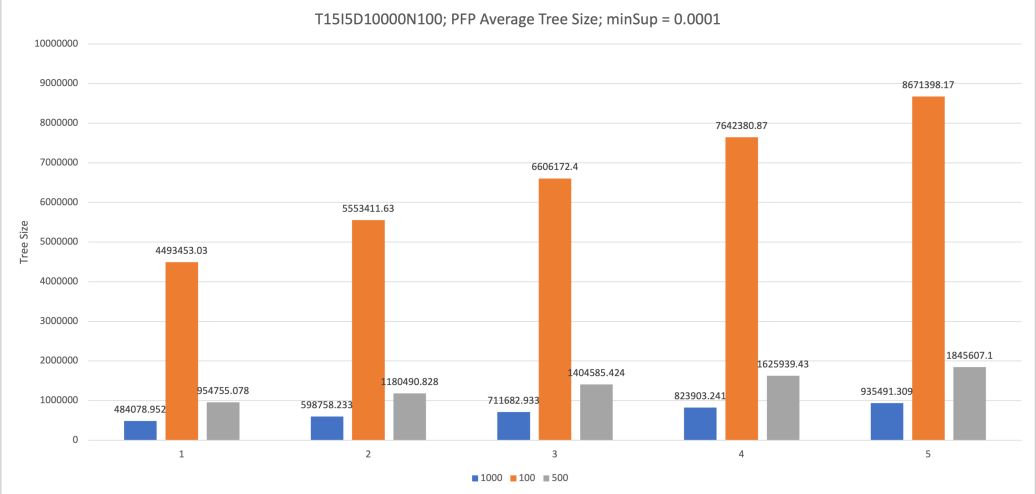


Figure 6.6: T15I5D10000N100, minSup = 0.0001, 1000 partitions, Set cover vs IPFIM

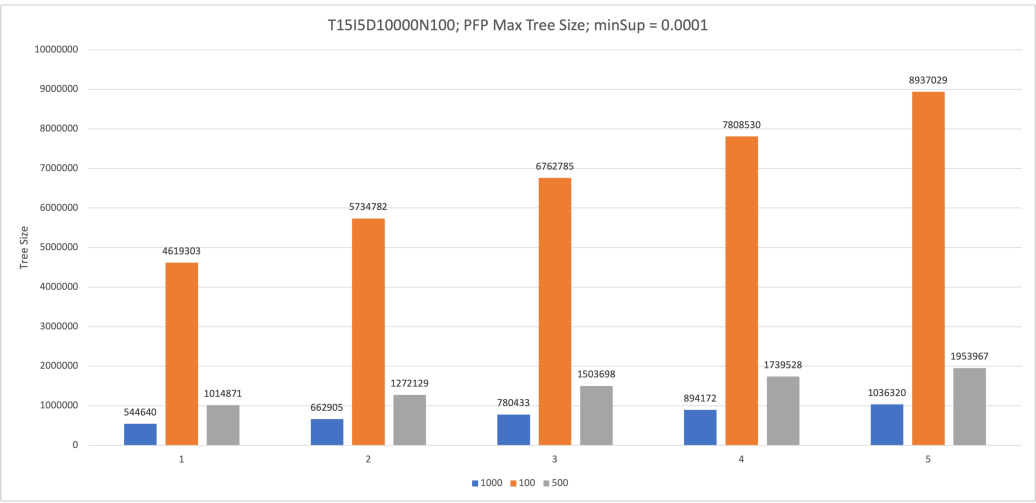
6.1.3 Tree size evaluation - Standalone

For the tree size evaluation, we collected the statistics of maximum, average and minimum tree size in each iteration.

PFP For **Figure 6.7b** the ratio is 0.21 and 0.116 between the different partitions, however for **Figure 6.7a** it is the partitions ration - 0.20 and 0.1 (similar to partitions ratio) .



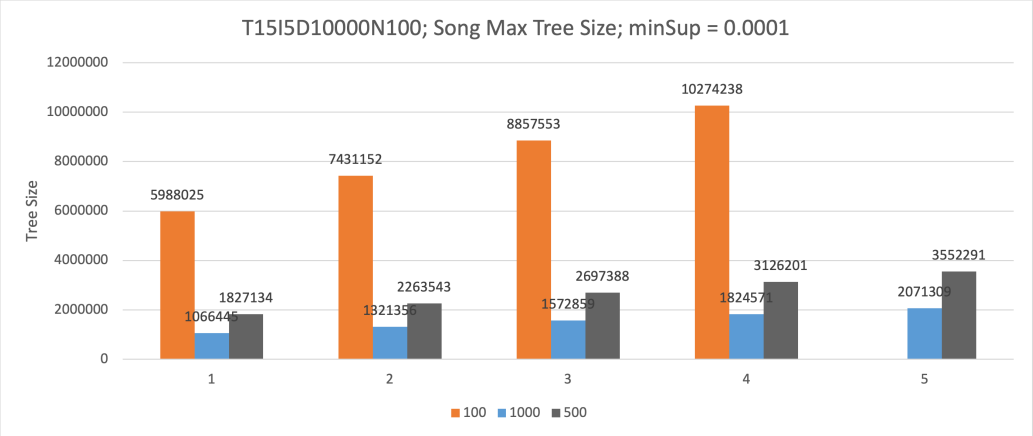
(a) T15I5D10000N100 PFP Average tree size



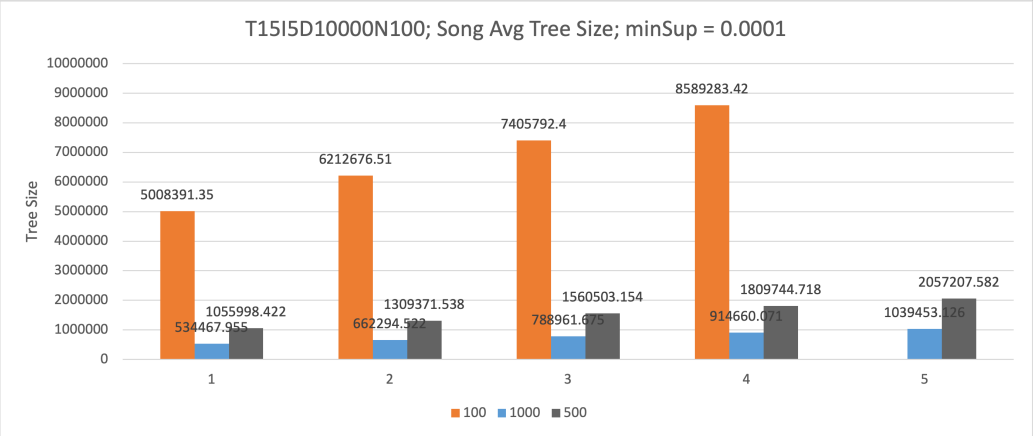
(b) T15I5D10000N100 PFP Max tree size

Figure 6.7: T15I5D10000N100 PFP Average and Max tree size for 100|1000|500 partitions

Song For **Figure 6.8a** the ratio is 0.3 and 0.17 between the different partitions, however for **Figure 6.8b** it is the partitions ration - 0.2 and 0.1 (same as partitions ratio).



(a) T15I5D10000N100 Song Max tree size



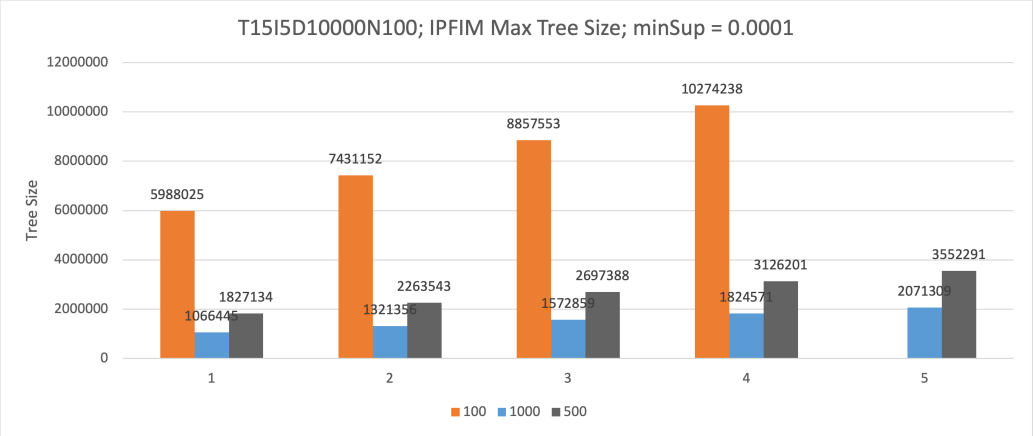
(b) T15I5D10000N100 Song Average tree size

Figure 6.8: T15I5D10000N100 Song Average and Max tree size for 100|1000|500 partitions

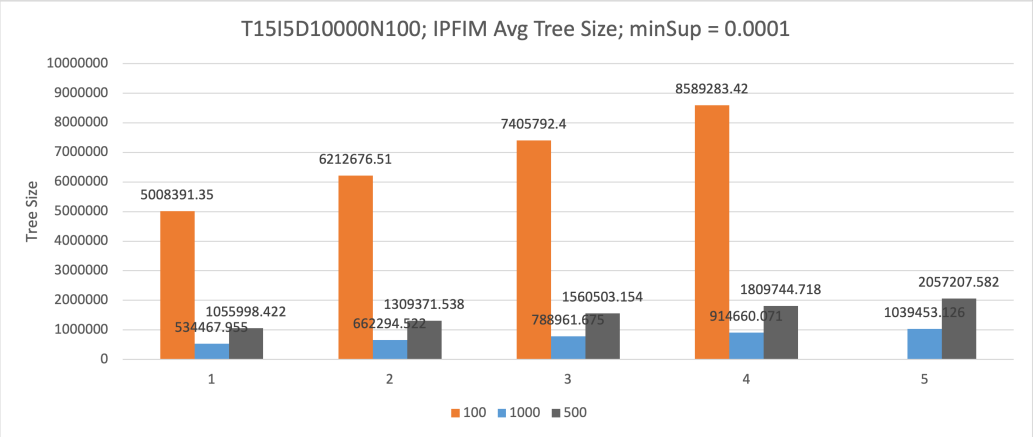
IPFIM For **Figure 6.9a** the ratio is 0.3 and 0.177 between the different partitions, however for **Figure 6.9b** it is the partitions ration - 0.2 and 0.1 (same as partitions ratio).

IPFIM-Improved For **Figure 6.10a** the ratio is 0.277 and 0.16 between the different partitions, however for **Figure 6.10b** it is the partitions ration - 0.2 and 0.1 (same as partitions ratio).

Set-Cover IPFIM For **Figure 6.11a** the ratio is 0.27 and 0.16 between the different partitions, after the repartition, the max size are the same as the

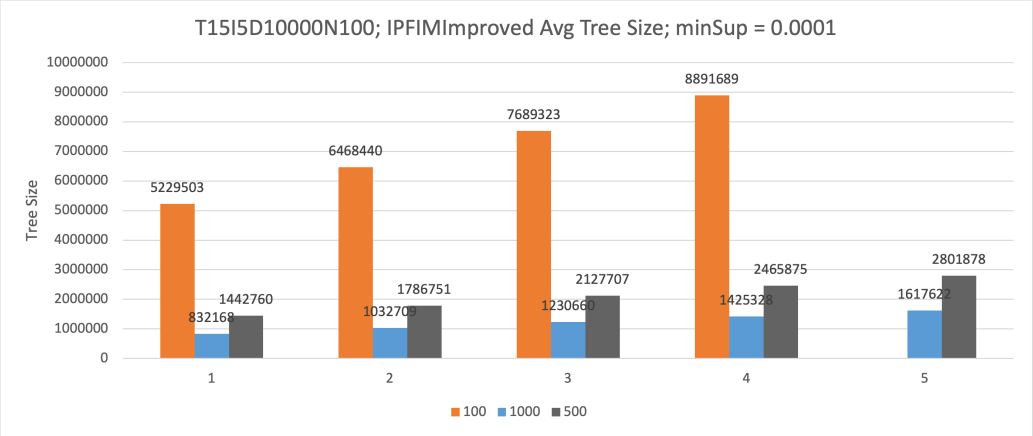


(a) T15I5D10000N100 IPFIM Max tree size

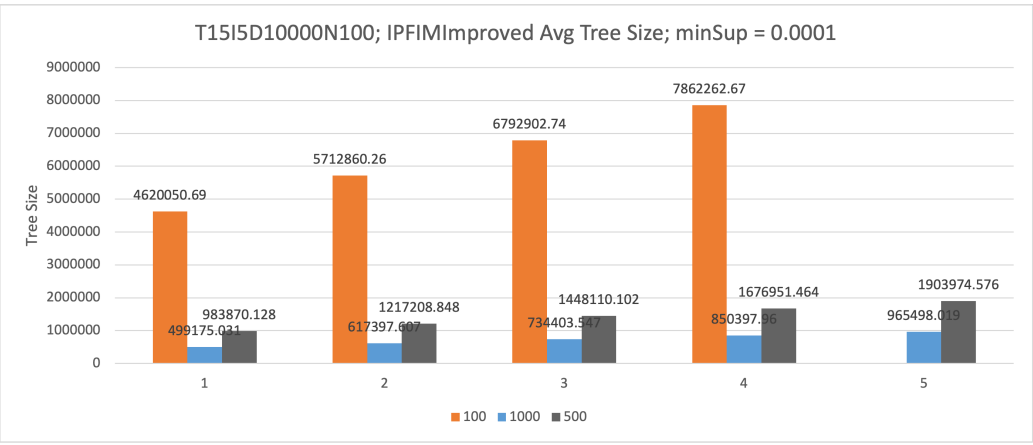


(b) T15I5D10000N100 IPFIM Average tree size

Figure 6.9: T15I5D10000N100 IPFIM Average and Max tree size for 100|1000|500 partitions

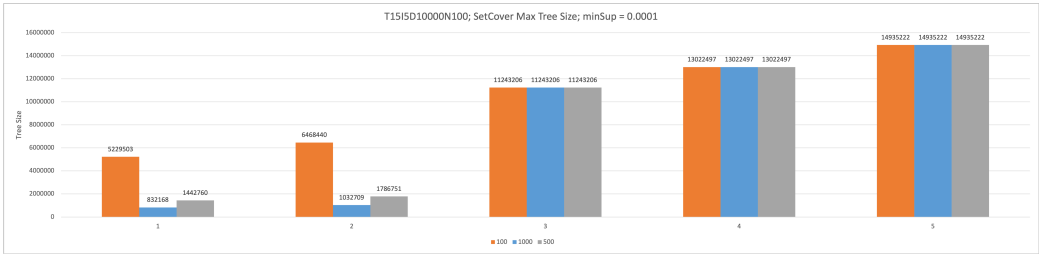


(a) T15I5D10000N100 IPFIM-Improved Max tree size



(b) T15I5D10000N100 IPFIM-Improved Average tree size

Figure 6.10: T15I5D10000N100 IPFIM-Improved Average and Max tree size for 100|1000|500 partitions, minSup = 0.0001, minMinSup = 0.1



(a) T15I5D10000N100 Set Cover IPFIM Max tree size

(b) T15I5D10000N100 Set Cover IPFIM Average tree size

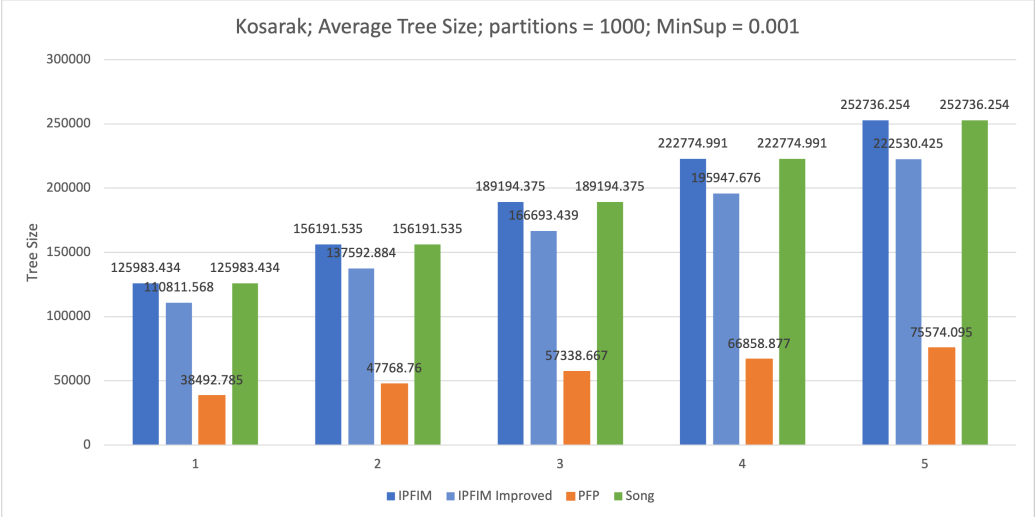
Figure 6.11: T15I5D10000N100 Set Cover IPFIM Average and Max tree size for 100|1000|500 partitions

algorithm sets the same partition size. However for **Figure 6.11b** although partitions ration - 0.2 and 0.1 (same as partitions ratio), the average size of the partitions is much more balanced,. The reason for not having same sizes, is that some partitions remained as is, and where not affected by the recalculation.

6.1.4 Tree size evaluation - Comparison

Figure 6.12a and **Figure 6.12b** present the comparison of average tree size for **item Dataset.1** and **item Dataset.3**.

(a) T15I5D10000N100 tree size comparison



(b) Kosarak tree size comparison

Figure 6.12: Tree size comparison for 1000 partitions

7 Discussion

7.1 Discussion

The total computation time consists from 2 main section:

1. Build FIS model.
2. Mine FIS from model.

IPFIM proposes improvement for the 1st item. When the 2nd item computation time is by far larger than 1st one, the improvement is negligible in total. The drawbacks of the CanTree algorithm are even more evident for small minSup values and large dataset - large memory consumption and slow minning time due to inefficient tree traversal. For this reason an improvement to the original IPFIM algorithm is proposed in details in section **section 4.1**.

IPFIM

For stressed results, large DB e.g. **Figure 6.3**, IPFIM failed for 100 due to memory error as the tree was the largest and not frequency optimized **Figure 6.9a**, 10M.

Similar to the conclusion of [20], IPFIM has advantage for moderate databases and support values, and its main advantage is the simplicity. It can be used to calculate different support values as the data is independent of the order nor the support values. An example can be seen in **Figure 6.5c** as IPFIM has the best performance, while **subsubsection 2.2.4** has the worst as it requires more processing.

IPFIM-Improved

Although in this work we haven't implemented all the corner cases from **paragraph 2.2.1**, the results show that maintaining a min-min-support improve the memory and mining when low resources are available **Figure 6.5a**. From **Figure 6.5a** we can see since the minSup is relatively low (item count of more then 10 is frequent), PFP is increasingly rising, while **subsubsection 2.2.4** is steady and has best performance, slightly better then **section 4.1**.

Set-Cover

As can be seen from **Figure 6.6**, the overhead of using a map for grouping, vs a simple hash functions, is very performance heavy. The potential in memory is evident from **Figure 6.11b**, as the trees are balanced and provide a good ratio of 50% less of an average tree size in comparison to **Figure 6.10b**.

8 Conclusion

8.1 Conclusions

In this thesis we have tested and compared 3 new techniques for incremental parallel mining of frequent itemsets.A major achievement of this thesis, is that IPFIM-Improved proved to have good results and a simple solution and implementation by combining 4 techniques - AFPIM, CPTree, CanTree and PFP. The Set-Cover algorithm, although has rational for optimized groups, the overhead of using groups as map instead of int-hash proved to be very resource dependent and inefficient. IPFIM is mostly useful for moderate min-support and database size.

For future work, we would suggest to test IPFIM-Improved with full implementation using CPTree **paragraph 2.2.1** and AFPIM **paragraph 2.2.1** suggested techniques and adjustments.

Bibliography

[1] Rakesh Agrawal and Ramakrishnan Srikant. Quest synthetic data generator. *IBM Almaden Research Center*, 1994.

[2] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[3] Apache. The apache mahout machine learning library,. URL: `http://mahout.apache.org/`.

[4] Apache. The apache™ hadoop® project develops open-source software for reliable, scalable, distributed computing. URL: `https://hadoop.apache.org/`.

[5] Apache. Mllib is apache spark's scalable machine learning library. URL: `https://spark.apache.org/mllib/`.

[6] Apache. Pfp mllib implementation. URL: `https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/fpm/FPGrowth.scala`.

[7] Apache. Song implementation. URL: `https://github.com/1level1/incrementalParallelCanTree/blob/564fcbd7fdd3662910531c5cc7a12274b5e16218/src/main/scala/levko/cantree/utils/cantreeutils.scala#L228`.

[8] Apache. Apache spark™ is a unified analytics engine for large-scale data processing, 2020. URL: `https://spark.apache.org/`.

[9] Daniele Apiletti, Elena Baralis, Tania Cerquitelli, Paolo Garza, Fabio Pulvirenti, and Luca Venturini. Frequent itemsets mining for big data: A comparative analysis. *Big Data Research*, 9:67–83, 2017. URL: `https://www.sciencedirect.com/science/article/`

pii/S2214579616300193, doi:https://doi.org/10.1016/j.bdr.2017.06.006.

[10] Rakesh M. Verma Ping Chen Computer Science Department Computer and Math Science Dept. University of Houston. The uh data mining hypertextbook, version 1. URL: https://www.hypertextbookshop.com/dataminingbook/public_version/contents/contents.html.

[11] Databricks. Apache spark, 2020. URL: https://databricks.com/spark/about.

[12] Tzung-Pei Hong, Chun-Wei Lin, and Yu-Lung Wu. Incrementally fast updated frequent pattern trees. *Expert Systems with Applications*, 34(4):2424–2435, 2008.

[13] Daniel Hunyadi. Performance comparison of apriori and fp-growth algorithms in generating association rules. In *Proceedings of the European computing conference*, pages 376–381, 2011.

[14] Jia-Ling Koh and Shui-Feng Shieh. An efficient approach for maintaining association rules - afpim.

[15] Jia-Ling Koh and Shui-Feng Shieh. An efficient approach for maintaining association rules based on adjusting fp-tree structures. In *International Conference on Database Systems for Advanced Applications*, pages 417–424. Springer, 2004.

[16] CK-S Leung, Quamrul I Khan, and Tariqul Hoque. Cantree: a tree structure for efficient incremental mining of frequent patterns. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 8–pp. IEEE, 2005.

[17] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y Chang. Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems*, pages 107–114, 2008.

[18] Xin Li, Zhi-Hong Deng, and Shiwei Tang. A fast algorithm for maintenance of association rules in incremental databases. In *International Conference on Advanced Data Mining and Applications*, pages 56–63. Springer, 2006.

[19] Mellanox. Sparkrdma plugin is a high-performance, scalable and efficient shufflemanager open-source plugin for apache spark., 2020. URL: `https://www.mellanox.com/products/SparkRDMA`.

[20] Yu-Geng Song, Hui-Min Cui, and Xiao-Bing Feng. Parallel incremental frequent itemset mining for large data. *Journal of Computer Science and Technology*, 32(2):368–385, 2017.

[21] Syed Khairuzzaman Tanbeer, Chowdhury Farhan Ahmed, Byeong-Soo Jeong, and Young-Koo Lee. Efficient single-pass frequent pattern mining using a prefix-tree. *Information Sciences*, 179(5):559–583, 2009.

[22] ua adrem. Dist-eclat and bigfim implementation. URL: `https://github.com/ua-adrem/bigfim`.

[23] Wikipedia. Set-cover algorithm, 1972. URL: `https://en.wikipedia.org/wiki/Set_cover_problem`.