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**

December 2020

Abstract

Frequent itemsets are collections of items which appear in a data set at an important frequency (usually greater than a predefined threshold) and can thus reveal association rules and relations between variables. Frequent pattern mining is a research area in data science applied to many domains such as recommender systems (what are the set of items usually ordered together), bioinformatics (what are the genes coexpressed in a given condition), decision making, clustering, website navigation.

The study of frequent itemsets has been studied since the 1990's. Many algorithms were developed during time and they are mostly focused on Apriori [**?**]—- and FP-Growth [**?**]. With years, size of databases has grown and today's databases' sizes go far beyond capabilities of a single machine. Recent studies show how to adopt classical algorithms for frequent itemsets mining for parallel execution and frameworks such as MapReduce. Even then, in case of a slight database update, like add-on of new transactions to the DB, it is required to re-run the MapReduce mining algorithm from the beginning on the whole data. Such events happen with high velocity in current databases and making full recalculation from the beginning may be far from optimal. Thus, a variation of these algorithms for incremental database update is desired. This work focuses on using tree based structure for parallel and incremental mining. To achieve this, the proposed algorithm is using a combination of two techniques. As a high-level overview, the PFP [**?**] algorithm is the base algorithm for parallel mining and CanTree [**?**] as the base structure for incremental updates. For the parallel computations, Spark was chosen [**?**], as it's mllib package already implements the PFP [**?**] algorithm.

A similar approach was already developed by [**?**] at 2017. In this article, we will extend this technique to use other incremental approaches and one we came up (SetCover), discuss the pros and cons of this approach and propose practical usages.

Acknowledgements

This work was supported by the [...] Research Fund of [...] (Number [...]). Additional funding was provided by [...] and [...]. We also thank [...] for contributing [...].

Contents

List of Figures

List of Tables

1 Introduction

1.1 Introduction

Mining of frequent items and association rules is a well known and studied field in Computer Science. The algorithms and solutions in this field can be roughly divided into two types - Apriori [**?**] and tree based solutions [**?** **?** **?**] Each type has benefits and limitations such as simplicity, performance, memory consumptions and scaling.

In this paper, we will describe an approach for dealing with an incrementally updated database, while avoiding candidate generation, and only a single DB scan.

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

For this article, we used 2 types of datasets:

1. Synthetic datasets of 100M, 10M and 1M transactions, and 2 magnitudes less of items, average length of 20 and 15 items. The datasets were generated using IBM Quest Synthetic Data Generator [**?**].
2. The Kosarak dataset contains 990,000 transactions with 41,270 distinct items and an average transaction length of 8.09 items (clickstream data of a hungarian on-line news portal). This dataset was the largest used by [**?**].

2 Previous Work

2.1 PRELIMINARY AND RELATED WORK

This paper will focus on the use case of an incremental mining, such as streaming data, while reading the full DB only once.

2.1.1 Related Work

One of the most well known algorithms for mining association rules is the Apriori algorithm [**?**]. This algorithm is iteratively generating candidates and pruning items with low support at each step. 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.

In the year 2000, a tree based solution was introduced, FPGrowth algorithm and structure [**?**]. This algorithm removes the need for candidate generation and yields better performance [**?**]. A small example is provided in Figure 2.2

2.1.2 Incremental Frequent Itemsets Mining

Incremental updates is to recompute outputs which depend on the incoming inputs only, without recomputing the whole data.

The basic challenge in incremental updates for frequent items mining, is a non consistent frequency order. Several algorithms such as AFPIM [**?**], EFPIM [**?**] and FUFP-tree [**?**] are keeping an updated frequency based trees, by reordering branches where frequency has changed.

The work of [**?**] presented a Canonical Tree (CanTree) which preserves

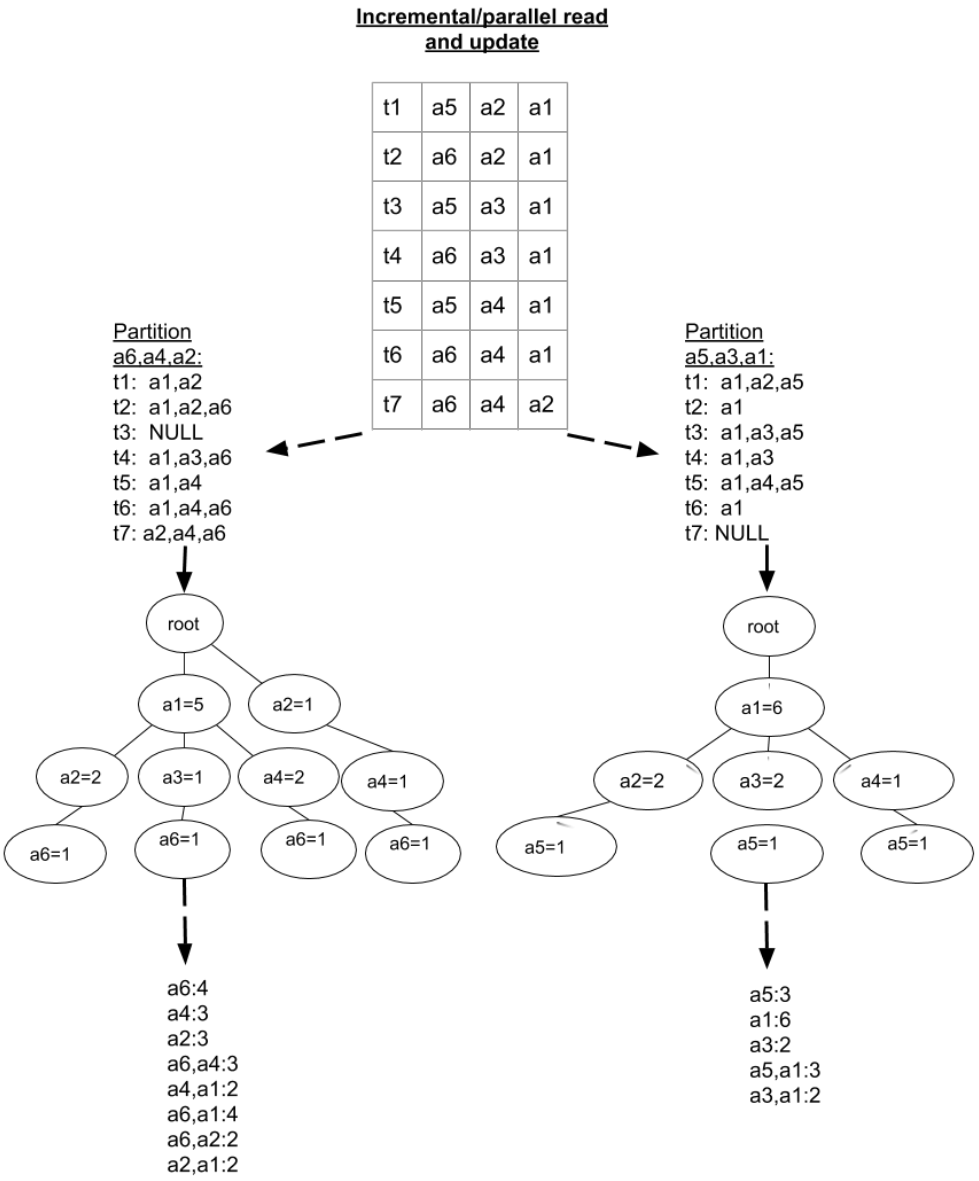

Figure 2.1: An example of trees and mining for minSup=2 and 2 partitions

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 work of [**?**] proposes an improvement to CanTree, 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 mention in the original article and as seen by our experiments, the CanTree and CP-Tree has a similar tree size, and the difference for our test cases was 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.

2.1.3 Parallel Frequent Itemsets mining

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

Step 1: Find global frequency list, F-List

Step 2: Group items G-list

Step 3: PFP:

1. For each Ti, order by F-List frequency
2. For each aj in Ti, replace aj with gi that aj belongs to its group
3. For each gi, if it appears in Ti, find its right-most location in Ti, say L and output: ¡key'=gi; value'=Ti[0]...Ti[L]¿
4. Group by key' = gi
5. For each group gi, build appropriate tree
6. For each group gi, mine the generated tree (filter items not in gi).

It is important to mention that there might be some duplicates at several groups of frequent itemsets in stage 3.6, but this is solved when filtering the 1-length items that are not relevant to the specific group. This duplication also affects the size of the generated trees and overall memory.

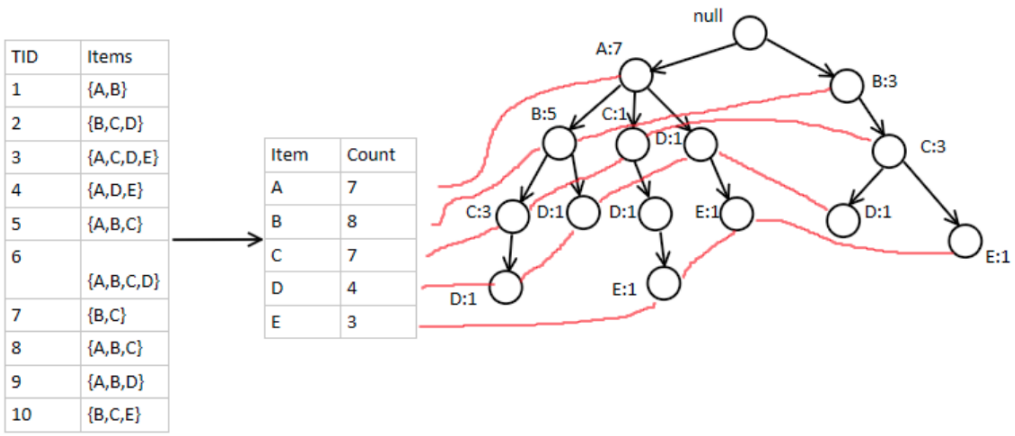

Figure 2.2: FPGrowth example

2.1.4 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 [**?**] and maintaining a pre-min support, together with using a semi-freq-order as in [**?**], will significantly improve memory and mining runtime results.

A paper by [**?**] proposes 2 Incremental and Parallel algorithms, IncMiningPFP and IncBuildingPFP. IncBuildingPFP and IPFIM algorithm are very much alike and were developed independently. IncMiningPFP is an improvement for IncBuildingPFP. IncMiningPFP is elaborated and compared later in this work.

3 IPFIM Algorithm

3.1 IPFIM - Incremental Parallel Frequent Itemsets Mining

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

3.1.1 IPFIM structure

Step 1: Define a comparison function for items:

compare(item1,item2)-¿bool

Step 2: For each set of transactions in iteration:

1. For each Ti in current set of transaction, order by pre defined compare function.
2. For each aj in Ti, replace aj with gi that aj belongs to its group
3. For each gi, if it appears in Ti, find its right-most location in Ti, say L and output: ¡key'=gi; value'=Ti[0]...Ti[L]¿
4. Group by key' = gi
5. For each group gi, merge to existing tree if exists ELSE build new tree
6. For each group gi, mine the generated tree (filter items not in gi of length 1).

3.1.2 Correctness

The correctness of the tree structure is driven from the correctness of mining CanTree, which preserves 2 properties:

Property 1: *The ordering of items is unaffected by the changes in frequency caused by incremental updates.*

Property 2: *The frequency of a node in the CanTree is at least as high as the sum of frequencies of its children.*

[**?**]

3.2 Experiments and Results

All the test cases were divided in to 50% base case and the remaining 50% to iterations of 2.5% (e.g. 20 iterations).

The implementation was done using Spark [**?**].

The used hardware is 4 clusters each with 20G memory and 40 cores. Also used is SparkRDMA Plugin [**?**].

For PFP [**?**] performance comparison, Spark [**?**] mllib package implements just that, and was used per iteration.

3.2.1 IPFIM vs PFP

For correct PFP [**?**] mining, a read of all the dataset till that point needs to be performed.

Synthetic Dataset

A comparison for 10M transactions (T15D10MN100K) with minSupport of 0.001, 1K partitions is seen at Figure 3.1.

Kosarak Dataset

A comparison with minSupport of 0.001, partitions of 10 and 100 is seen at Figure 3.2.

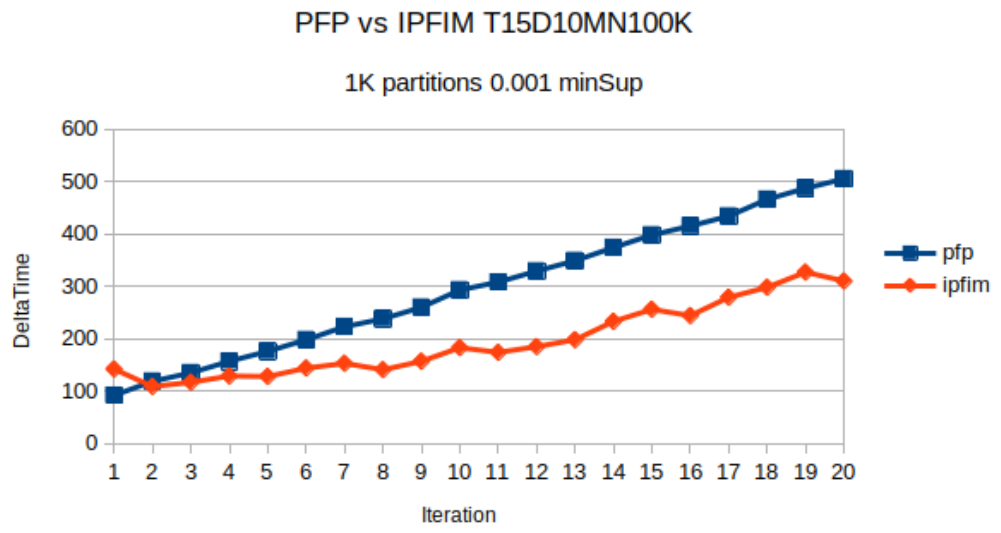

Figure 3.1: PFP vs IPFIM 10M

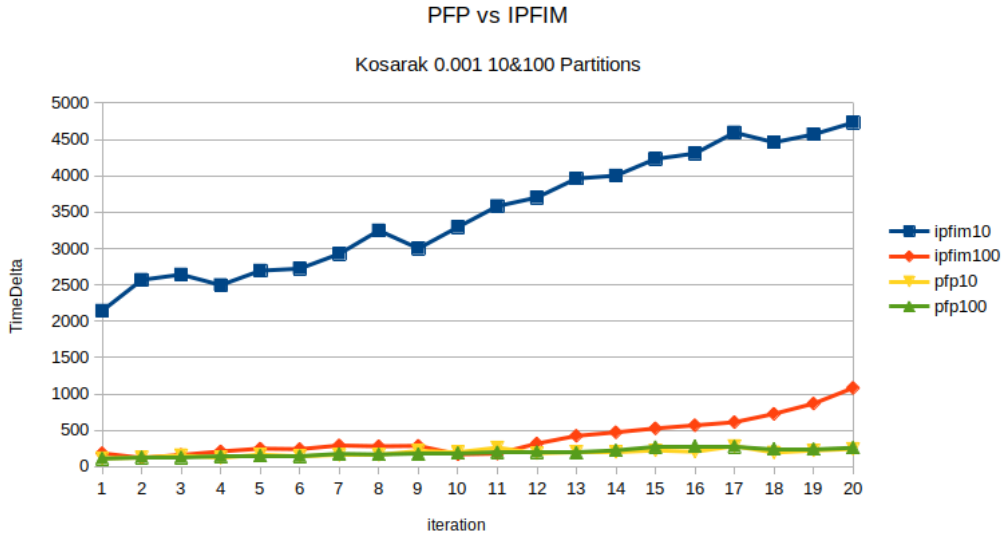

Figure 3.2: PFP vs IPFIM Kosarak

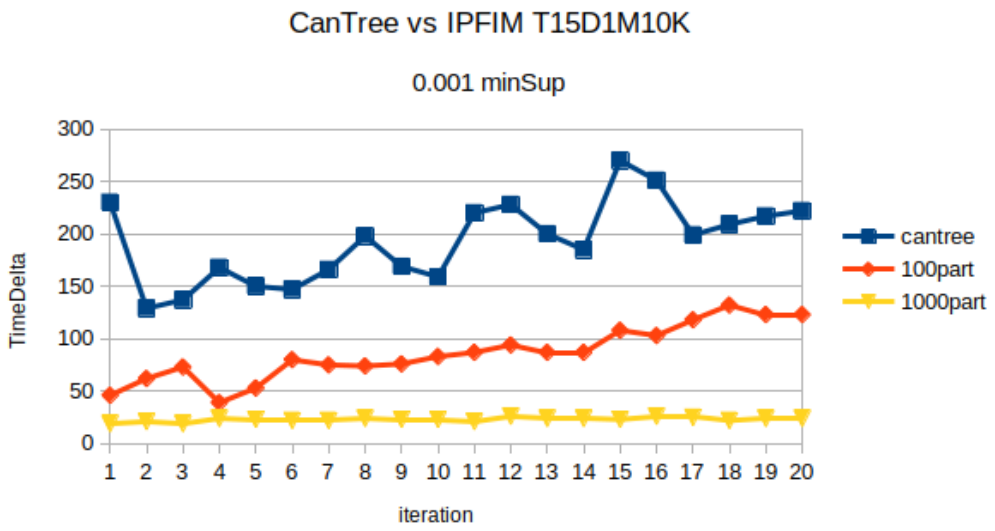

Figure 3.3: Inc. vs IPFIM

3.2.2 IPFIM vs CanTree

For CanTree [**?**] performance, we used IPFIM with only one group, meaning a single tree.

Synthetic Dataset

A comparison for 1M transactions (T15D1MN10K) with minSupport of 0.001, partitions of 1, 10 and 100 is seen at Figure 3.3.

Kosarak Dataset

A comparison with minSupport of 0.001, partitions of 1, 10 and 100 is seen at Figure 3.4.

3.3 Discussion

The total computation time consists from 2 main section:

1. Read, scan and build model.

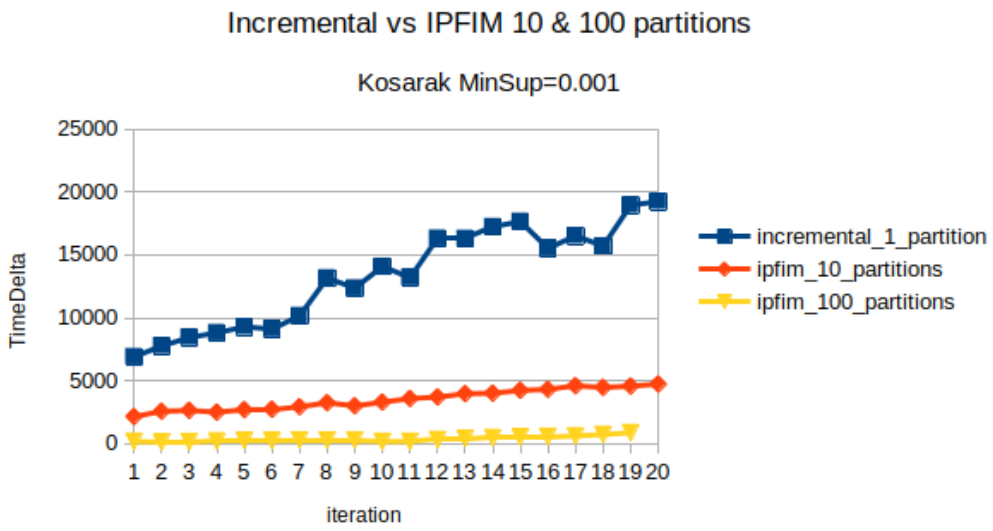

Figure 3.4: Inc. vs IPFIM 10 & 100 partitions

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 section *Improvments*.

For moderate support values and dataset sizes, where ratio of 1 and 2 is balanced, IPFIM outperforms PFP as it will require only one DB scan and iterative construction (Figure 3.1)

3.3.1 PFP VS IPFIM

Synthetic Datasets

As seen at Figure 3.1, PFP [**?**] starts better, but after a few iterations, IPFIM is more flat for every iteration, while PFP [**?**] increases linearly due to full data read.

Kosarak Dataset

For the Kosarak dataset, results are seen in Figure 3.2. Since there are much more frequent items (20X) for minSup of 0.001, the results show a poor performance for mining with only 10 partitions. For 100 partitions, the results are still better for PFP [**?**], where the exponential mining process is the bottle neck for IPFIM.

3.3.2 CanTree VS IPFIM

Synthetic Datasets

Using 10M iterations and 100K itemsets with only 1 partition (regular CanTree), will result a memory overflow with the used architecture, thus the results are not presented here. For a 1M transaction with 10K unique itemsets, the results improve at 5X for every 10X partitions. This is due to memory improvements and parallel computations as seen in Figure 3.3.

Kosarak Dataset

For the Kosarak dataset, the improvement is 5X and 50X for 10 and 100 partitions, as seen in Figure 3.4.

3.4 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.

To handle these obstacles, 2 techniques were implemented. To handle the computation time, an approach similar to CPTree [**?**] 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 AFPIM [**?**] was implemented. We added

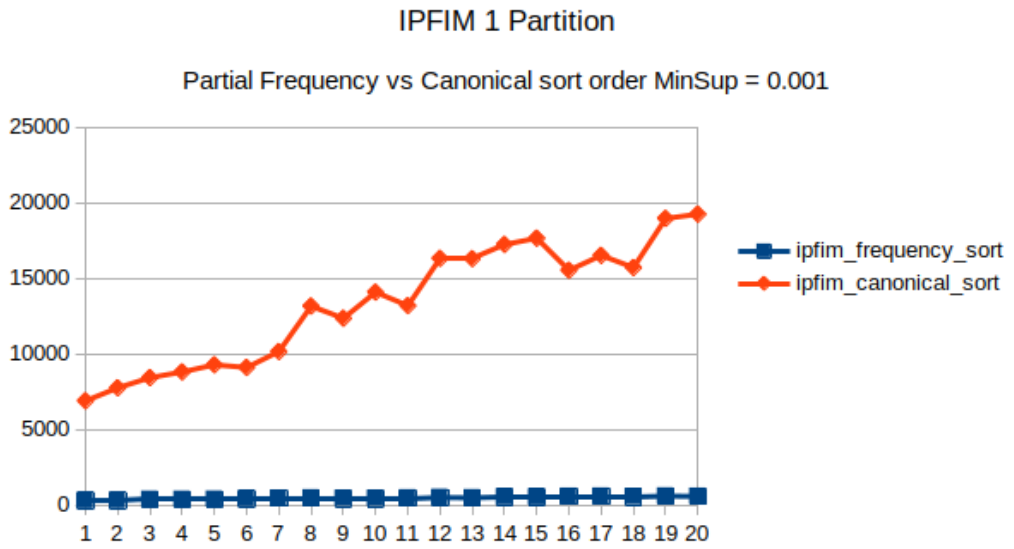

Figure 3.5: IPFIM partial frequency sort vs IPFIM cannonical sort

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.

3.4.1 IPFIM Improved vs IPFIM - Computation

Although the size of the tree was not effected by more than 10%, when using semi-frequency order, computation time was improved by 30X when running single partition, as seen in Figure 3.5.

3.4.2 IPFIM Improved vs IPFIM - Memory

Using a partial approach of AFPIM [**?**], we were able to run synthetic datasets of 100M transactions and 100k unique item sets. The results, compared to PFP, for 100 partitions, min support of 0.01 and 0.003 can be seen in Figure 3.6. As there is only 1 dataset scan for IPFIM, and we pre-defined the semi-frequency order, the results are 10x faster even for 1st iteration, and improve to 25x for last one.

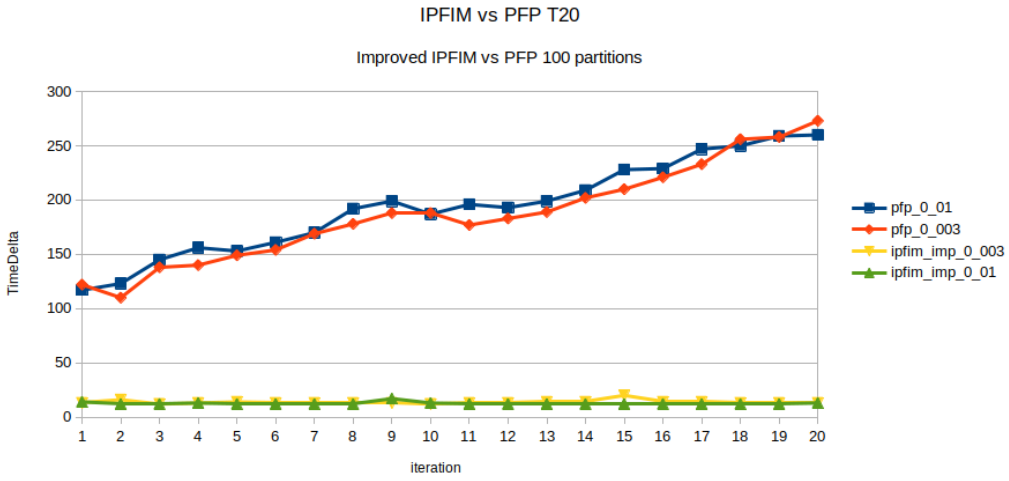

Figure 3.6: IPFIM pre-min, semi-frequency vs PFP

4 Conclusion

4.1 Conclusions

For a single computation of frequent items, the benchmark for performance and memory for IPFIM, is PFP. This is because IPFIM is using similar techniques, and FP tree is the optimal structure for this purpose (except some variations mentioned in previous sections, e.g. optimal sharding). As already mentioned in the *discussion* section, when there is a relatively equal ratio between reading a dataset and computation time of frequent item sets, IPFIM with the suggested improvements out performs PFP. However, for large FIS computation time, this advantage is negligible in total.

Using a canonical order approach, as in Cantree, was almost not practical for large data sets, nor for small min support calculations. The improvement of using a semi-frequency and pre-min support limitation, provides the best balance , and provides best performance.

For future work, it is interesting to enhance PFP to use ”smart” grouping. For example trying to use greedy set cover to find groups for of frequent itemsets.