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

This article 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 [9] algorithm is the base algorithm for parallel mining and CanTree [8] as the base structure for incremental updates.

As far as we know, this is the first algorithm proposing to use the advantages of tree structures for parallel and incremental mining. In this article, we will present the algorithm, review results comparing to most common parallel and incremental tree mining algorithms, 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 [2] and tree based solutions [8, 12, 13] Each type has benefits and limitations such as simplicity, performance, memory consumptions and scaling. We can also divide the requirements for frequent items mining, into use cases like:

1. Build and mine once.
2. Build once mine many different scenarios.
3. Build and mine with support for incremental updates.

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 [1].
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 [12].

2 Background

Your text here

3 Previous Work

3.1 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.

3.1.1 Related Work

One of the 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. 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 [2]. This algorithm removes the need for candidate generation and yields better performance [5]. A small example is provided in Figure 3.2

3.1.2 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 [7], EFPIM [10] and FUFP-tree [4] are keeping an updated frequency based trees, by reordering branches where frequency has changed.

The work of [8] presented a Canonical Tree (CanTree) which preserves the

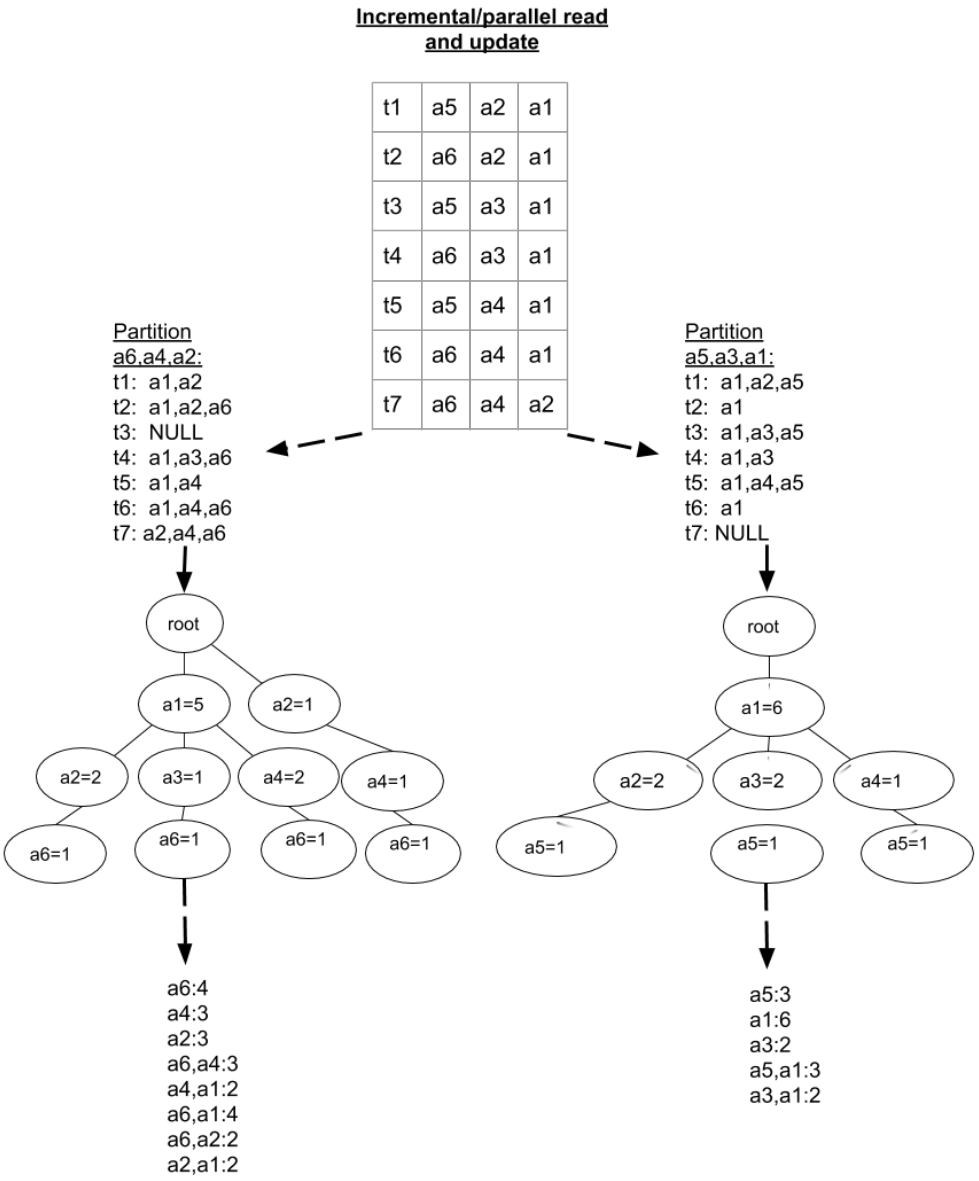

Figure 3.1: An example of trees and mining for minSup=2 and 2 partitions

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 [12] 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 results of XXX, using semi-frequency based order, improves the mining results by 10X. The reasoning is discussed in section XXX.

3.1.3 Parallel Frequent Itemsets mining

The difficulty in parallelizing FP-growth is to distribute iterations to parallel trees while still allowing correct mining. PFP [9] 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 [9] 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.

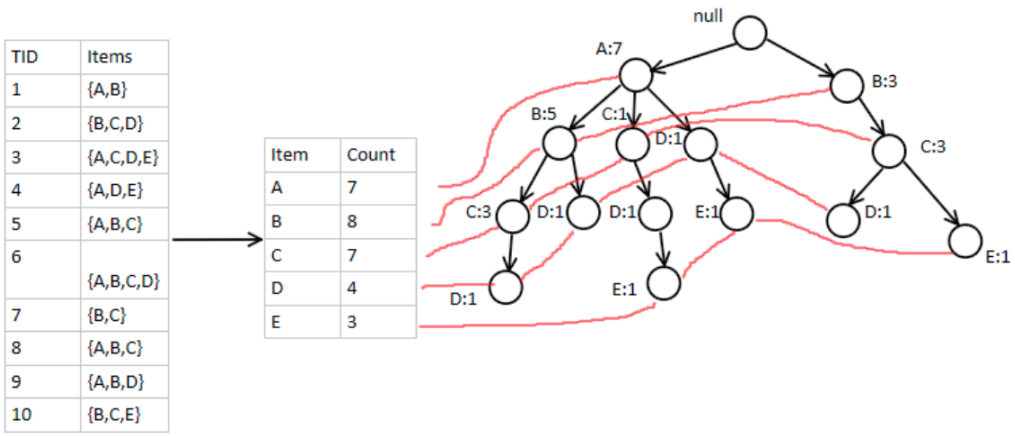

Figure 3.2: FPGrowth example

3.1.4 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 results, using an approach similar to [6] and maintaining a pre-min support, together with using a semi-freq-order as in [12], will significantly improve memory and mining runtime results.

4 IPFIM Algorithm

4.1 IPFIM - Incremental Parallel Frequent Itemsets Mining

The implementation of this algorithm strongly depends on PFP [9]. To support incremental tree updates, we are using a predefined comparison function to arrange the items insertion order, as used in CanTree [8].

4.1.1 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).

4.1.2 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.*

[**?**]

4.2 section1

4.3 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 [3].

The used hardware is 4 clusters each with 20G memory and 40 cores. Also used SparkRDMA Plugin [11].

For PFP [9] performance comparison, Spark [3] mllib package implements just that and was used in iteration while consuming all data.

For CanTree [8] performance, we used IPFIM with only one group, meaning a single tree.

4.3.1 PFP VS IPFIM

For correct PFP [9] mining, a full read of the dataset needs to be performed every iteration.

For the synthetic datasets, a comparison for minSup of 0.001 can be seen at Figure 4.1. As seen, PFP [9] starts better, but after a few iterations, IPFIM is more flat for every iteration, while PFP [9] increases linearly.

For the Kosarak dataset, results are seen in Figure 4.2. Since there are much more frequent items for minSup of 0.001, the results show a poor

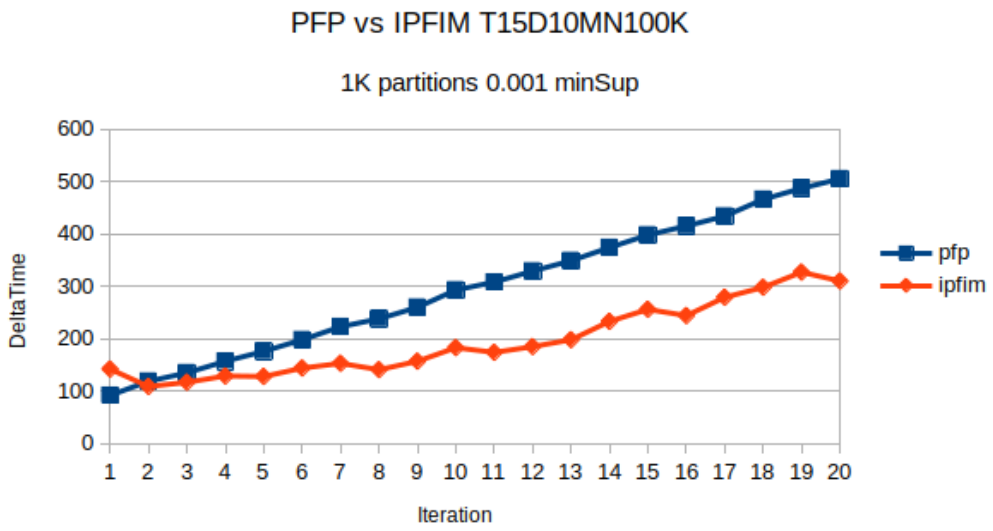

Figure 4.1: PFP vs IPFIM 10M

performance for mining with only 10 partitions, since the trees are much larger. For 100 partitions, the results are still better for PFP [9], where the exponential mining process is the bottle neck for IPFIM. An improvment is discussed in section XXX

4.3.2 IPFIM vs Interactive CanTree

Using 10M iterations and 100K itemsets with only 1 partition (regular CanTree), will result a memory issue 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. For the Kosarak dataset, .

4.4 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.

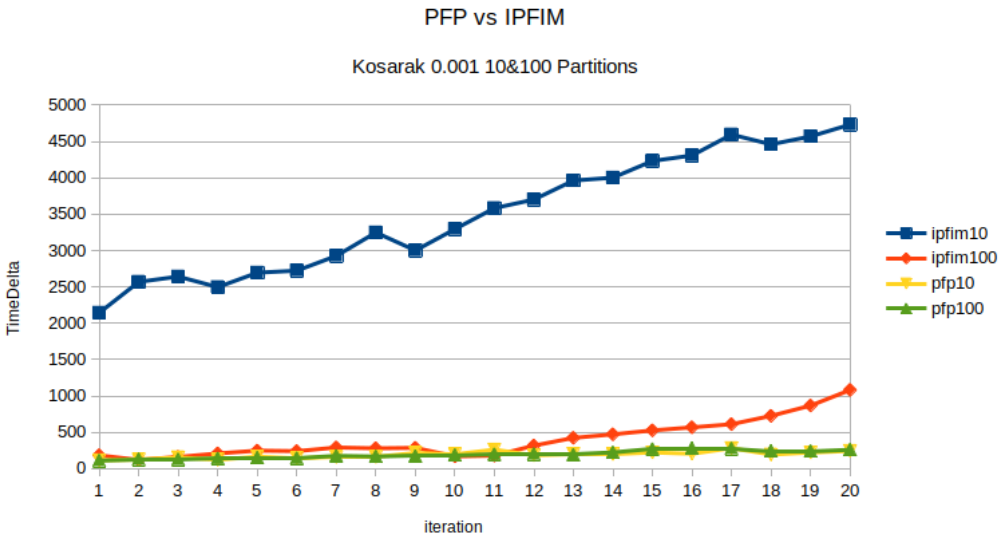

Figure 4.2: PFP vs IPFIM Kosarak

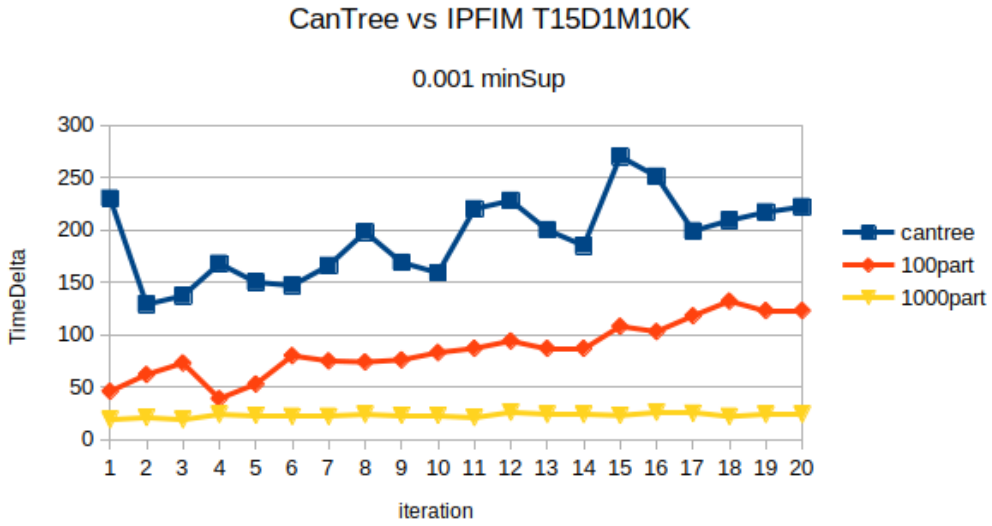

Figure 4.3: Inc. vs IPFIM

To handle these obstacles, 2 techniques were implemented. To handle the computation time, an approach similar to [12] was tested. Although the size of the tree was not effected by more than 10%, computation time was improved by XXX.

To handle the memory limitation, which is caused by the construction of a large tree, a partial approach of AFPIM [7] was implemented. We added a pre-min support to identify pre-frequent items, and tested performance while re reading the full dataset for missed items when those are found. Using this approach we were able to run synthetic datasets of 100M transactions and 1M unique item sets. The results can be seen in XXX compared to PFP.

Theorem 1 *Your theorem here.*

Proof: Your proof here ■

4.4.1 subsection1

5 Conclusion

5.1 Conclusions

From the results, there is a memory limitation when building trees with full items to support interactive transactions. Splitting to multiple partitions and calculations will improve memory and runtime performance.

The improvements mentioned in section XXX are a good approach for handling

Bibliography

[1] AGRAWAL, R., AND SRIKANT, R. Quest synthetic data generator. *IBM Almaden Research Center* (1994).

[2] AGRAWAL, R., SRIKANT, R., ET AL. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB* (1994), vol. 1215, pp. 487–499.

[3] APACHE. Apache spark™ is a unified analytics engine for large-scale data processing, 2020.

[4] HONG, T.-P., LIN, C.-W., AND WU, Y.-L. Incrementally fast updated frequent pattern trees. *Expert Systems with Applications 34*, 4 (2008), 2424–2435.

[5] HUNYADI, D. Performance comparison of apriori and fp-growth algorithms in generating association rules. In *Proceedings of the European computing conference* (2011), pp. 376–381.

[6] KOH, J.-L., AND SHIEH, S.-F. An efficient approach for maintaining association rules.

[7] KOH, J.-L., AND SHIEH, S.-F. An efficient approach for maintaining association rules based on adjusting fp-tree structures. In *International Conference on Database Systems for Advanced Applications* (2004), Springer, pp. 417–424.

[8] LEUNG, C.-S., KHAN, Q. I., AND HOQUE, T. Cantree: a tree structure for efficient incremental mining of frequent patterns. In *Fifth IEEE International Conference on Data Mining (ICDM'05)* (2005), IEEE, pp. 8–pp.

[9] LI, H., WANG, Y., ZHANG, D., ZHANG, M., AND CHANG, E. Y. Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems* (2008), pp. 107–114.

[10] LI, X., DENG, Z.-H., AND TANG, S. A fast algorithm for maintenance of association rules in incremental databases. In *International Conference on Advanced Data Mining and Applications* (2006), Springer, pp. 56–63.

[11] MELLANOX. Sparkrdma plugin is a high-performance, scalable and efficient shufflemanager open-source plugin for apache spark., 2020.

[12] TANBEER, S. K., AHMED, C. F., JEONG, B.-S., AND LEE, Y.-K. Efficient single-pass frequent pattern mining using a prefix-tree. *Information Sciences 179*, 5 (2009), 559–583.

[13] TSAY, Y.-J., HSU, T.-J., AND YU, J.-R. Fiut: A new method for mining frequent itemsets. *Information Sciences 179*, 11 (2009), 1724–1737.

תוכן עניינים

תקציר

תקציר בעברית כאן.

האוניברסיטה הפתוחה
המחלקה למתמטיקה ולמדעי המחשב

נושא התזה כאן

עבודת תזה זו הוגשה כחלק מהדרישות לקבלת תואר
"מוסמך למדעים" .ד. במדעי המחשב
באוניברסיטה הפתוחה
המחלקה למתמטיקה ומדעי המחשב

על-ידי
שם הכותב כאן

בהנחיית שם המנחה כאן

נובמבר 2019