Higher Performance IPPC$^+$ Tree for Parallel Incremental Frequent Itemsets Mining

Van Quoc Phuong Huynh(✉) and Josef Küng

Institute for Application Oriented Knowledge Processing (FAW), Faculty of Engineering and Natural Sciences (TNF), Johannes Kepler University (JKU), Linz, Austria
{vqphuynh, jkueng}@faw.jku.at

Abstract. IPPC tree provides incremental properties and high performance for mining frequent itemsets through shared-memory parallel algorithm IFIN$^+$. However, in the case of datasets comprising a large number of distinguishing items but just a small percentage of frequent items, IPPC tree becomes to lose its advantage in running time and memory for the tree construction. With a motivation of reducing the execution time for the tree building, in this paper, we propose an improved version for IPPC tree, called IPPC$^+$, to increase the performance of the tree construction. We conducted extensive experiments on both synthetic and real datasets to evaluate IPPC$^+$ tree against IPPC tree. Besides, the IFIN$^+$ with the new tree is also compared to the well-known algorithm FP-Growth and the other two state-of-the-art ones, FIN and PrePost$^+$. The experimental results show that the construction time of IPPC$^+$ tree is improved remarkably compared to that of IPPC tree; and IFIN$^+$ is the most efficient algorithm, especially in the case of mining at different support thresholds within the same running session.

Keywords: Incremental · Parallel · Frequent itemsets mining · Data mining Big data · IPPC · IPPC$^+$ · IFIN · IFIN$^+$

1 Introduction and Related Works

Frequent itemsets mining can be briefly described as follows. Given a dataset of n transactions $D = \{T_1, T_2, \ldots, T_n\}$, the dataset contains a set of m distinct items $I = \{i_1, i_2, \ldots, i_m\}$, $T_i \subseteq I$. A k-itemset, *IS*, is a set of k items ($1 \leq k \leq m$). Each itemset *IS* possesses an attribute, *support*, which is the number of transactions containing *IS*. The problem is featured by a support threshold ε which is the percent of transactions in the whole dataset *D*. An itemset *IS* is called frequent itemset iff $IS.support \geq \varepsilon * n$. The problem is to discover all frequent itemsets existing in *D*.

This problem was started up by Agrawal & Srikant with algorithm Apriori [1]. This algorithm generates candidate (k + 1)-itemsets from frequent k-itemsets at the (k + 1)th pass and then scans dataset to check whether a candidate (k + 1)-itemsets is a frequent one. Many previous works were inspired by this algorithm. Algorithm Partition [7] aims at reducing I/O cost by dividing a dataset into non-overlapping and memory-fitting partitions which are sequentially scanned in two phases. In the first phase, local candidate itemsets are generated for each partition, and then they are checked in the second one. DCP [8] enhances Apriori by incorporating two dataset pruning techniques introduced in DHP [9] and using direct counting method for storing candidate itemsets and counting their support. In general, Apriori-like methods suffer from two drawbacks: a deluge of generated candidate itemsets and/or I/O overhead caused by repeatedly scanning dataset.

Two other approaches, which are more efficient than Apriori-like methods, are also proposed to solve the problem: (1) frequent pattern growth adopting divide-and-conquer with FP tree structure and FP-Growth [2], and (2) vertical data format strategy in Eclat [10]. FP-Growth and algorithms based on it such as [11, 12] are efficient solutions since unlike Apriori, they avoid many times of scanning dataset and generation-and-test. However, they become less efficient when datasets are sparse. While algorithms based on FP-Growth and Apriori use a horizontal data format; Eclat and some other algorithms [7, 13, 14] apply vertical data format, in which each item is associated with a set of transaction identifiers, Tids, containing the item. This approach avoids scanning dataset repeatedly, but a huge memory overhead is expensed for sets of Tids when dataset becomes large and/or dense. Recently, two remarkably efficient algorithms are introduced: FIN [3] with POC tree and PrePost$^+$ [4] with PPC tree. These two structures are prefix trees and similar to FP tree, but the two mining algorithms use additional data structures, called Nodeset and N-list respectively, to significantly improve mining speed.

Discovering frequent itemsets in a large dataset is an important problem in data mining. In Big Data era, this mining model, as well as other ones, has been being challenged by very large volume and high velocity of datasets which are fast accumulated over time. As a solution in our previous work [15] to deal with this problem, we proposed a tree structure, named IPPC (Incremental Pre-Post-Order Coding), which supports incremental tree construction; and an algorithm for incrementally mining frequent itemsets, IFIN (Incremental Frequent Itemsets Nodesets). For one our next work [16], we introduced a shared-memory parallel version of IFIN, named IFIN$^+$, to enhance performance by exhausting as much as possible the computational power of commodity processors which are equipped with many physical computational units.

Through experiments, algorithm IFIN$^+$ has demonstrated its superior performance compared to the well-known algorithm FP-Growth [2], and other two state-of-the-art ones FIN [3] and PrePost$^+$ [4]. However, in case of datasets comprising a large number of distinct items but just a small percentage of frequent items for a certain support threshold, we investigates that IPPC tree becomes to lose its advantage in running time and memory for its construction compared to other trees such as FP, POC, and PPC of algorithms FP-Growth, FIN, and PrePost$^+$. The reason is that these trees use just frequent items for their tree structures while the IPPC tree uses all items in datasets for its structure to be compensated with the abilities of incremental tree construction and mining. Table 1 reports the detail of construction time of the trees on a synthetic dataset and a real one, named Kosarak, at support threshold $\varepsilon = 0.1\%$ and 0.2% respectively. Note that, the construction of IPPC tree does not depend on the support thresholds.

Table 1. Tree construction time on datasets.

Tree construction time (Synthetic dataset of 1200k transactions, $\varepsilon = 0.1\%$)						
	200k	400k	600k	800k	1000k	1200k
IPPC tree	2.1 s	3 s	3.2 s	4.1 s	4.3 s	5.1 s
FP tree	3.4 s	5.7 s	8.7 s	10 s	12.9 s	16.2 s
POC/PPC trees	2.5 s	4.5 s	6.7 s	9.7 s	11.9 s	14.7 s
Tree construction time (Kosarak dataset of 990002 transactions, $\varepsilon = 0.2\%$)						
	200k	400k	600k	800k	1000k	
IPPC tree	7.2 s	10.4 s	11.4 s	13.9 s	14.7 s	
FP tree	3.3 s	6.2 s	9.2 s	12.6 s	15.3 s	
POC/PPC trees	2.1 s	3.7 s	5.1 s	6.6 s	7.5 s	

The synthetic dataset at $\varepsilon = 0.1\%$ includes 843 frequent items, 90% of all 932 items. In this case, the construction time of IPPC tree is superior to that of the other trees, approximate a ratio 1:3 to the time of POC/PPC trees at full size of 1200k transactions. In contrast, for Kosarak dataset at $\varepsilon = 0.2\%$, there are just 568 frequent items, 1.37% in the total of 41270 items. IPPC tree becomes to lose its advantage since extra computational overhead for a very large proportion of infrequent items, and its construction time is twice the time of POC/PPC trees at the full size of Kosarak dataset. Hence, the aim of this paper is to reduce the running time of the IPPC tree construction. Through experiments on these two datasets, the IPPC$^+$ tree, the new version of IPPC tree, achieves a remarkable improvement of the tree building performance compared to IPPC tree and contributes better running time for the mining algorithm IFIN$^+$ as well. Besides, the IFIN$^+$ with the new tree is also compared with the well-known algorithm FP-Growth and the other two state-of-the-art ones, FIN and PrePost$^+$. The experimental results show that IFIN$^+$ is the most efficient algorithm, especially in the cases of mining at different support thresholds within the same running session.

Table 2. Example transaction dataset

ID	Items in transactions
1	b, e, d, f, c
2	d, c, b, g, f, h
3	f, a, c
4	a, b, d, f, c, h
5	b, d, c

The rest of the paper is organized as follows. Section 2 will mention some essential concepts of IPPC tree structure and then introduce the improved solution. Section 3 presents experiments; and finally, conclusions are given in Sect. 4.

2 Improved Solution

In this section, we propose a solution to reduce the construction time for the IPPC tree. More detail about the IPPC tree can be found in [16]. For convenience in reference and concise content, just the fundamental concepts and pseudo code of IPPC tree are presented; and based on that the IPPC$^+$ tree will be introduced.

2.1 IPPC Tree

IPPC tree is a compact and information-lossless structure of the whole items of all transactions in a given dataset. Its construction needs only one data scanning and does not require a given support threshold. Local order of items in a path of nodes from the root to a leaf is flexible and can be changed to improve compression. Each node in the tree is identified by a pair of codes: *pre-order* and *post*-order. With these characteristics, a built IPPC tree from a dataset D can be mined at different support thresholds and reused to build up a new IPPC tree corresponding to a new dataset $D' = D + \Delta D$.

To demonstrate the concept of IPPC tree building process, the Fig. 1 records transaction by transaction in Table 2 inserted into an empty IPPC tree. Initially, the tree has only the root node, and transaction 1(b, e, d, f, c) is inserted as it is in Fig. 1(a). The Fig. 1(b) is of the tree after transaction 2 (d, c, b, g, f, h) is added. The item b in transaction 2 is merged with node b in the tree. Although transaction 2 does not contain item e, but its common items d, f, and c can be merged with the corresponding nodes. Item d is found common, so it is merged with node d after node d is swapped[1] with node e to guarantee the Property 2. Similarly, items f and c are merged with node f and

[1] Swapping two nodes is simply exchanging one's item name to that of the other.

c respectively; and the remaining items *g* and *h* are inserted as a child branch of node *c*. In Fig. 1(c), transaction 3 (*f*, *a*, *c*) is processed. Common item *f* is found that can be merged with node *f*, so node *f* is swapped with node *b*. Item *c* is also a common one, but it is not able to be merged with node *c* as node *d* does not satisfy the Descendant Swapping condition with node *c*. Then the items *a* and *c* are added as a branch from node *f*. When transaction 4 (*a*, *b*, *d*, *f*, *c*, *h*) is added in Fig. 1(d), common items *f*, *d*, *b*, and *c* are merged straightforwardly with corresponding nodes *f*, *d*, *b*, and *c*. The remaining items *a* and *h* are then inserted into the subtree having root node *c*. The item *h* is found common with node *h* in the second branch. Node *h* and item *h*, therefore, are merged together after node *h* is swapped with node *g*. The last item *a* is then inserted as a new child branch from node *h*. Insertion of transaction 5 (*b*, *d*, *c*) is depicted in Fig. 1 (e). All items in transaction 5 are common, but they cannot be merged with nodes *b*, *d*, and *c* as node *f* does not guarantee the Child Swapping condition. Thus, transaction 5 is added as a new child branch of the root node.

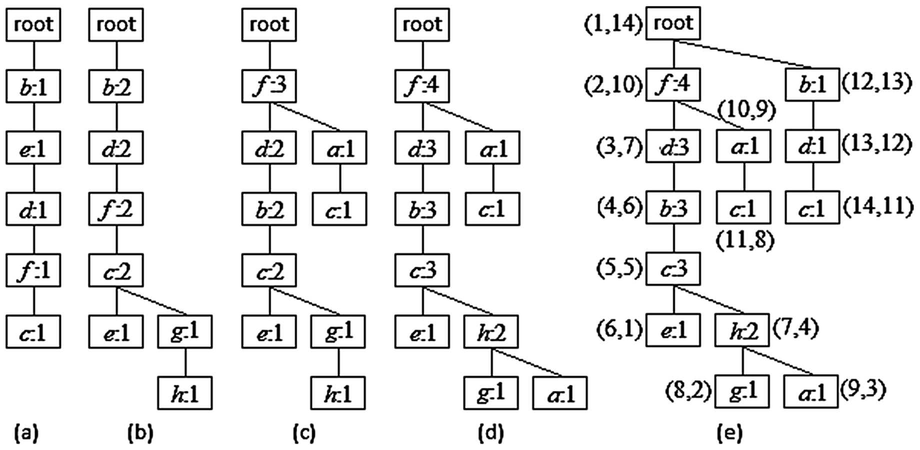


Fig. 1. An illustration for constructing an IPPC tree on example transaction dataset

After the dataset has been processed, each node in the IPPC tree is assigned a pair of sequent numbers (*pre-order*, *post-order*) by scanning the tree with pre-order and post-order traversals through procedure **AssignPrePostOrder**. For an example, node (4, 6) is identified by *pre-order* = 4 and *post-order* = 6, and it registers item *b* with *support* = 3.

```
Algorithm 1: BuildIPPCTree
Input: Dataset D, root node R
Output: An IPPC tree with root R, item list ℒ
1.  For Each transaction $T \in D$
2.    Update items and their supports in ℒ from items in T;
3.    InsertTransaction(T, R);
4.  End For
5.  AssignPrePostOrder(R);

Procedure InsertTransaction(Transaction T, Node R)
1.  subNode ← R; notMerged;
2.  While($T \neq \emptyset$)
3.    notMerged ← true;
4.    For Each child node N of subNode
5.      If(N.item-name ∈ T)
6.        notMerged ← false; N.support++;
7.        subNode ← N; T ← (T \ N.item-name); break;
8.      End If
9.    End For
10.   If(notMerged) break;
11. End While
12. If($T = \emptyset$) Return;
13. For Each child node N of subNode
14.   If(MergeDescendants(T, N)) Return;
15. End For
16. Insert T as a new branch from subNode (added nodes are ini-
    tialized at 1 for their supports);

Function MergeDescendants(Transaction T, Node N)
1.  subNode ← N; mrgNode ← N; merged ← false;
2.  While(subNode satisfies the Child Swapping condition)
3.    descendant ← subNode.child;
4.    If(descendant.item-name ∈ T)
5.      T ← (T \ descendant.item-name); merged ← true;
6.      Exchange item names of mrgNode and descendant;
7.      mrgNode.support++; mrgNode ← mrgNode.child;
8.    End If
9.    subNode ← descendant;
10. End While
11. If(merged) Insert T as a new branch from mrgNode.parent
    (added nodes are initialized at 1 for their supports);
12. Return merged;

Procedure AssignPrePostOrder (Node R)
    // PreOrder and PostOrder are initialized at 1.
1.  R.pre-order ← PreOrder; PreOrder++;
2.  For Each child node N of R Do AssignPrePostCode(N);
3.  R.post-order ← PostOrder; PostOrder++;
```

2.2 IPPC$^+$ Tree

The IPPC tree comprises the whole items of all transactions of a given dataset, regardless of the support threshold. Therefore, in case of a dataset possessing a large number of distinct items, the number of child nodes of every single node in the tree becomes larger. The more distinguishing items, the bigger the number of child nodes of the tree nodes will be. Before an item in a transaction is merged with a tree node or inserted into the tree as a new node, a sequence search for a child node (of the current parent node) having the same item name as the item will be done. Consequently, these factors have caused more computational overhead to insert a new transaction into the current IPPC tree.

To improve the performance of the transaction insertion, we replace the sequence search with the binary search based on the item name. The replacement requires maintaining the item name based order of child nodes of every single node in IPPC tree. The for-loop (**InsertTransaction** procedure, from lines 4 to 9) is changed from sequence traversal on lists of child nodes to sequence traversal on items of transactions. Since the cardinalities of child node lists are very much larger than the numbers of items in transactions, this changing obviously improves the performance. The more difference between the two kinds of cardinality, the more performance is enhanced. However, the changing does not guarantee the items with higher support are accumulated into the IPPC tree before other items with lower support. This drives the reduction of compression for higher-support items; and therefore, the *nodesets* [15, 16] of these items become longer that may decrease the performance in mining process of algorithm IFIN$^+$. To avoid this issue, the major items (refer the definition below) in transactions are extracted and sorted in the descending order of their supports and then combined, as the first part, with the remaining items of the transactions.

Definition (Major Items): Major items are items which their supports are greater than or equal to a given threshold, called major threshold α. As usual, α is greater than the support thresholds ε.

Based on the above concepts, the pseudocode in Subsect. 2.2 is redesigned for the IPPC$^+$ tree construction as follows.

```
Algorithm 2: BuildIPPC+Tree
Input: Dataset D, root node R, major item list MI, threshold $\alpha$
Output: An IPPC+ tree with root R, item list $\mathcal{L}$, MI
1.  If(MI = ∅) Sample on dataset D to achieve major item list
    MI based on the major threshold $\alpha$;
2.  For Each transaction T ∈ D
3.    Update items and their supports in $\mathcal{L}$ from items in T;
4.    Based on MI, major items in T are extracted and sorted in
      descending order of their supports, then combined as the
      first part with the rest of T;
5.    InsertTransaction(T, R);
6.  End For
7.  Update MI based on the item list $\mathcal{L}$;
8.  AssignPrePostOrder(R);

Procedure InsertTransaction(Transaction T, Node R)
1.  parentNode ← R;
2.  While((mergedNode = Merge(T, parentNode)) ≠ null)
3.    parentNode = mergedNode;
4.  End While
5.  If(T = ∅) Return;
6.  For Each child node N of parentNode
7.    If(MergeDescendants(T, N)) Return;
8.  End For
9.  Insert T as a new branch from parentNode (added nodes are
    initialized at 1 for their supports);

Function Merge(Transaction T, Node N)
1.  mergedNode ← null;
2.  For Each item name I of T
3.    mergedNode ← binary search for the child node in child
      list of N which its item name equals I;
4.    If(mergedNode ≠ null)
5.      mergedNode.support++;
6.      T ← (T \ I);
7.      Return mergedNode;
8.    End If
9.  End For
10. Return null;
```

```
Function MergeDescendants(Transaction T, Node N)
1.  subNode ← N; mrgNode ← N; merged ← false;
2.  While(subNode satisfies the Child Swapping condition)
3.    descendant ← subNode.child;
4.    If(descendant.item-name ∈ T)
5.      T ← (T \ descendant.item-name); merged ← true;
6.      Exchange item names of mrgNode and descendant;
7.      mrgNode.support++; mrgNode ← mrgNode.child;
8.    End If
9.    subNode ← descendant;
10. End While
11. If(merged)
12.   Correct the position of node N in the child node list of
      its parent node;
13.   Insert T as a new branch from mrgNode.parent (added nodes
      are initialized at 1 for their supports);
14. End If
15. Return merged;
```

The IPPC$^+$ tree construction needs a major item list to preprocess transactions (line 4 of **BuildIPPC$^+$Tree**) before the transactions are inserted into the tree. The IPPC$^+$ tree construction is not aware of the major item list of the initial dataset. Therefore, it can be achieved by sampling (or even a full scan) on the dataset. The major item list *MI* of a dataset *D* is used as acknowledge to incrementally build up the tree (corresponding to dataset *D*) with a new additional dataset Δ *D*, and then *MI* is updated based on the item list $\mathcal{L}$ of the dataset $D + \Delta D$. The procedure **AssignPrePostOrder** is the same as the one in Subsect. 2.1.

To insert a transaction into the tree, the function **Merge** is executed first to find and merged the first items of the transaction with the tree nodes, the binary search is employed in this process. If there are still items in the transaction, the function **MergeDescendants** is done to merge these items with descendant nodes. When at least a merger between an item and a descendant node happens in the function **MergeDescendants**, this means the item name of node *N* is changed that may cause to lose the right order of child node list of *N*'s parent node. Therefore, an order correction is done at line 12. Finally, the remaining items in the transaction (if possible) is inserted as a new branch into the tree at lines 13 and 9 in function **MergeDescendants** and procedure **InsertTransaction** respectively.

After the tree construction has completed, a mining process with the algorithm IFIN + [16] on the built tree will be executed.

3 Experiments

All experiments were conducted on a 1.86 GHz Intel Core (MT) i3-4030U processor, and 4 GB memory computer with Window 8.1 operating system. To evaluate algorithms, we used the Market-Basket Synthetic Data Generator [5] based on the IBM

Quest to generate a synthetic dataset, and a real dataset named Kosarak [6], online news portal click-stream data. The properties of the datasets are shown in Table 3.

Table 3. The datasets' properties

	No. of transactions	Max length	Average length	No. of total distinct items	No. of frequent items at thresholds		
					0.001	0.002	0.006
Synthetic dataset	1200000	32	10	932	843	774	525
Kosarak	990002	2498	8.1	41270	1260	568	116

All the algorithms were implemented in Java. Experimental values of running time and used memory are the average values from three corresponding individual ones. In our previous works [15, 16], to guarantee available memory of 2 GB used for Java Heap, we set value "-Xmx2G" for _JAVA_OPTIONS, a Windows environment variable. However, we realize that this causes the Java garbage collector to run many times of full memory collection; and consequently, the total running time includes a significant percentage, approximate 45%, for the garbage collection. To avoid this in the current work, we set the value "-Xms2G -Xmx2G" for _JAVA_OPTIONS instead, and result in the running time for garbage collection is reduced to 6% which reflects more exactly the algorithms' performance.

For emulating scenarios of incremental mining, the synthetic dataset was divided into six equal parts, 200 thousand transactions for each one, and so on for Kosarak dataset with five parts in which the last one contains just 190002 transactions. The experiments start mining on the first part and then part by part from the second one is accumulated and mined. IFIN$^+$ can perform following three scenarios:

- **S1** (Incremental in Different Sessions): An IPPC/IPPC$^+$ tree corresponding with a dataset had been constructed, mined and stored in a running session. In the following sessions, the old tree is loaded and then built up with a new additional dataset.
- **S2** (Incremental in the Same Session): An IPPC/IPPC$^+$ tree corresponding with a dataset has been constructed and mined, and then it is built up with a new additional dataset in the same session.
- **S3** (Just Loading Tree): A stored IPPC/IPPC$^+$ tree in a previous session is loaded and mined in the following sessions.

Each execution scenario can be performed with different support thresholds in the same running session. The processor in our computer possesses two physical computational units, and we found that the performance achieved its best with two threads in parallel version IFIN$^+$. We set the major threshold $\alpha = 0.02$ for the construction of IPPC$^+$ tree. The experiments will be presented in three parts: comparisons between the two trees IPPC and IPPC$^+$ with the inclusion of IFIN$^+$, and algorithm IFIN$^+$ with the IPPC$^+$ tree against algorithms FP-Growth, FIN, and PrePost$^+$ on each of the two datasets.

3.1 Comparisons Between IPPC and IPPC$^+$ Trees

In this subsection, we present comparisons between two versions of the algorithm IFIN$^+$ mining on IPPC and IPPC$^+$ trees with the synthetic and Kosarak datasets, based on the running time of the four partial processing phases.

Table 4 reports the running time in seconds of the execution phases of the two version of IFIN$^+$ on the synthetic dataset in data accumulation steps from 200k to 1200k transactions. As shown in Table 3, the number of distinct items in this dataset is small that the efficiency improvement of the tree construction with binary search is not enough to compensate the computational overhead of extracting and sorting the major items in each transaction. This causes the tree construction time to increase, but the amounts are not considerable, 0.3 s in average. Almost there are no performance differences in the second phase, the Frequent 2-itemset Generation. The extracting and sorting the major items in each transaction causes appearances of these items to tend to be lesser and nearer to the root node, that makes the lengths of *nodesets* [15, 16] of major items reduce. This explains why the running time is decreased in the third and the fourth phases, 0.7 s and 0.2 s in average respectively. Consequently, the total efficiency is improved, around 1 s for steps of data accumulation from 600k to 1200k transactions.

Table 4. Running time of IFIN$^+$ mining on the IPPC/IPPC$^+$ trees with the synthetic dataset

Running time in tree construction phase (in S1 Scenario)						
	200k	400k	600k	800k	1000k	1200k
IPPC tree	2.1 s	3 s	3.2 s	4.1 s	4.3 s	5.1 s
IPPC$^+$ tree	2.8 s	3.2 s	3.5 s	4.2 s	4.5 s	5.5 s
Running time in frequent 2-itemset generation phase (ε = 0.001)						
	200k	400k	600k	800k	1000k	1200k
IFIN$^+$ (IPPC tree)	0.4 s	0.5 s	0.8 s	0.9 s	1 s	1.4 s
IFIN$^+$ (IPPC$^+$ tree)	0.3 s	0.5 s	0.8 s	0.9 s	1 s	1.4 s
Running time in nodeset generation phase (ε = 0.001)						
	200k	400k	600k	800k	1000k	1200k
IFIN$^+$ (IPPC tree)	1.4 s	2.4 s	3.7 s	4.3 s	6 s	6.3 s
IFIN$^+$ (IPPC$^+$ tree)	1.1 s	2 s	2.8 s	3.8 s	4.8 s	5.4 s
Running time in discover frequent k-itemsets phase (ε = 0.001)						
	200k	400k	600k	800k	1000k	1200k
IFIN$^+$ (IPPC tree)	1.6 s	2.3 s	3.1 s	4.6 s	5.0 s	6.1 s
IFIN$^+$ (IPPC$^+$ tree)	1.5 s	2.3 s	2.9 s	4 s	4.7 s	5.8 s
Total running time (in S3 Scenario, ε = 0.001)						
	200k	400k	600k	800k	1000k	1200k
IFIN$^+$ (IPPC tree)	5.7 s	8.5 s	11.6 s	14.8 s	17 s	20 s
IFIN$^+$ (IPPC$^+$ tree)	6 s	8.4 s	10.7 s	13.8 s	15.8 s	19 s

Table 5 presents the running time of the four processing phases for the two version of IFIN$^+$ on Kosarak dataset in data accumulation steps from 200k to 990002 transactions. The Kosarak dataset includes a large number of distinct items as reported in Table 3. Therefore, the IPPC$^+$ construction performance based on the binary search is improved significantly, much far the computational overhead for extracting and sorting the major items in each transaction. As a result, the tree construction time is reduced considerably from 29% to 41% in the sequence of data accumulation steps. While the running time in the second phase increases, the running time in the third and the fourth ones reduce. In general, the performance of the last three phases between the two versions is not much difference, except the accumulation step of 1000k transactions. For the total effect, the running time is reduced approximately 25% for all steps of data accumulation of Kosarak dataset.

Table 5. Running time of IFIN$^+$ mining on the IPPC/IPPC$^+$ trees with Kosarak dataset

Running time in tree construction phase (in S1 Scenario)					
	200k	400k	600k	800k	1000k
IPPC tree	7.2 s	10.4 s	11.4 s	13.9 s	14.7 s
IPPC$^+$ tree	5.1 s	6.8 s	7.2 s	8.3 s	8.7 s
Running time in frequent 2-itemset generation phase (ε = 0.002)					
	200k	400k	600k	800k	1000k
IFIN$^+$ (IPPC tree)	0.6 s	1.1 s	1.7 s	2.3 s	2.5 s
IFIN$^+$ (IPPC$^+$ tree)	0.6 s	1.4 s	2 s	2.6 s	3.2 s
Running time in nodeset generation phase (ε = 0.002)					
	200k	400k	600k	800k	1000k
IFIN$^+$ (IPPC tree)	0.2 s	0.3 s	0.5 s	0.6 s	0.8 s
IFIN$^+$ (IPPC$^+$ tree)	0.2 s	0.3 s	0.4 s	0.5 s	0.7 s
Running time in discover frequent k-itemsets phase (ε = 0.002)					
	200k	400k	600k	800k	1000k
IFIN$^+$ (IPPC tree)	1 s	2.4 s	3.4 s	4.8 s	6.7 s
IFIN$^+$ (IPPC$^+$ tree)	1 s	2.2 s	3.3 s	4.5 s	5.4 s
Total running time (in S3 Scenario, ε = 0.002)					
	200k	400k	600k	800k	1000k
IFIN$^+$ (IPPC tree)	9.2 s	14.7 s	17.4 s	22.1 s	25.3 s
IFIN$^+$ (IPPC$^+$ tree)	7.1 s	11 s	13.2 s	16.5 s	18.6 s

In an overview of experiments on both datasets, the performance of IFIN$^+$ using IPPC$^+$ tree is improved compared to that of the version using IPPC tree. The larger the number of distinguishing items and transactions in a dataset; the more the running time is saved for the tree construction beside the minor efficient improvement in the mining process.

3.2 Comparisons with Other Algorithms on the Synthetic Dataset

In this subsection, we benchmark the running time and the peak consumed memory of IFIN$^+$ using IPPC$^+$ tree against that of the three algorithms FP-Growth, FIN, and PrePost$^+$ on the synthetic dataset. In that, the algorithm IFIN$^+$ experiments with all its possible execution scenarios S1, S2, and S3 as referred.

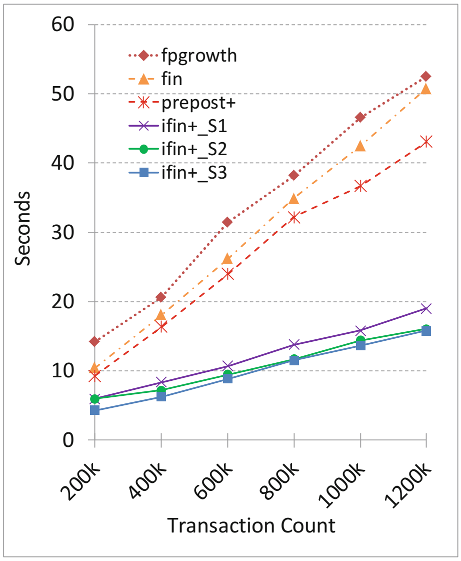


Fig. 2. Running time on the incremental synthetic datasets

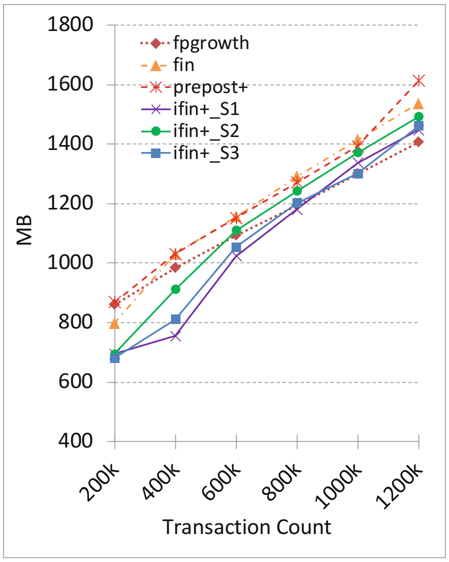


Fig. 3. Peak memory on the incremental synthetic datasets

Figures 2 and 3 sequentially demonstrate the running time and the peak memory of the algorithms in steps of data accumulation at the support threshold $\varepsilon = 0.1\%$. For all algorithms, both running time and peak memory increase linearly when the dataset is accumulated. While FP-Growth is the slowest algorithm, it uses memory more efficient than FIN and PrePost$^+$. Algorithm PrePost$^+$ consumes the most memory, but it runs faster than FP-Growth and FIN. Algorithm IFIN$^+$ is the most efficient for the running time. For data accumulation steps up to 600k transactions, IFIN$^+$ uses memory more efficient than FP-Growth; and for remaining steps, FP-Growth takes this advantage, but not considerable. The slopes of the running time lines of IFIN$^+$ are the same and lower than that of the three remaining algorithms. Hence, follow the data accumulation steps, the execution time of IFIN$^+$ becomes more dominant, lesser than a haft, compared to the remaining algorithms'. Among the three execution scenarios of IFIN$^+$, the running time of S2 and S3 is almost the same and better than S1's but not much difference.

Beside the high performance of mining phases in algorithm IFIN$^+$, one more reason can be found out in Table 6 which reports the construction time of the four trees. Note that the IPPC$^+$ tree construction does not depend on support threshold, but the other three trees. The POC, PPC trees of algorithms FIN and PrePost$^+$ are almost the same, so their running time of the tree building is nearly equal. The IPPC$^+$ tree construction of IFIN$^+$ achieves the best performance, much better than the three algorithms'. Especially in scenario S3, the time ratios are approximately 1:7 and 1:6 compared to FP tree and PPC tree respectively. At the same dataset size, building tree in S3 is faster than that in S1, approximate 2.6 s in average; since the execution scenario S1 must build up the loaded tree with a new additional dataset of 200k transactions. This also reveals that constructing an IPPC$^+$ tree by loading its stored data is much efficient than building the same tree from the same dataset.

Table 6. The tree construction time of the algorithms for the synthetic dataset

	200k	400k	600k	800k	1000k	1200k
IPPC$^+$ tree (Scenario S1)	2.8	3.2	3.5	4.2	4.5	5.5
IPPC$^+$ tree (Scenario S3)	0.5	0.7	1.2	1.6	2.1	2.2
FP tree (ε = 0.001)	3.4	5.7	8.7	10	12.9	16.2
POC/PPC (ε = 0.001)	2.5	4.5	6.7	9.7	11.9	14.7

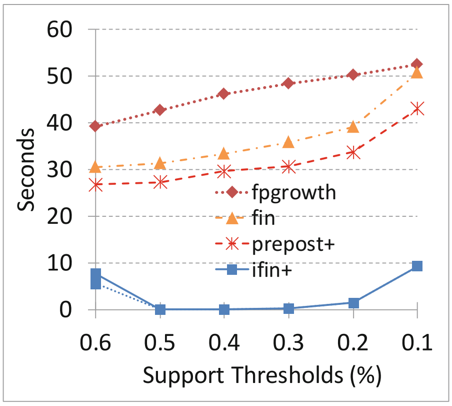


Fig. 4. Running time on the synthetic dataset at different support thresholds

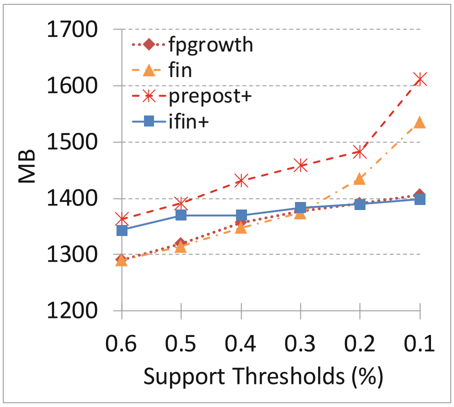


Fig. 5. Peak memory on the synthetic dataset at different support thresholds

In Figs. 4 and 5, the running time and peak used memory are visualized for the algorithms mining on the synthetic dataset of 1.2 million transactions with different support thresholds ε. Start at ε = 0.6%, IFIN$^+$ can perform one of two scenarios S1 or S3 that their two running time values are shown in Fig. 4. For other ε values, IFIN$^+$ just run its mining tasks since the built tree is completely reused. Furthermore, only a portion of its mining is performed. Consequently, with following values of $\varepsilon <$ 0.6%, the running time of IFIN$^+$ takes an overwhelming dominance against that of the three algorithms. The memory used by IFIN$^+$ increases slowly follow the steps of support thresholds, and approximates the memory used by FP-Growth for threshold values from 0.4 to 0.1. The algorithm FP-Growth uses memory more efficient than the two algorithms FIN and PrePost$^+$. However, its running time is considerably longer than that of FIN and PrePost$^+$. Algorithm PrePost$^+$ run faster than FIN and FP-Growth, but it uses the most memory.

3.3 Comparisons with Other Algorithms on Kosarak Dataset

Similar to the previous subsection, this one presents the running time and the used memory of the four algorithms for Kosarak dataset.

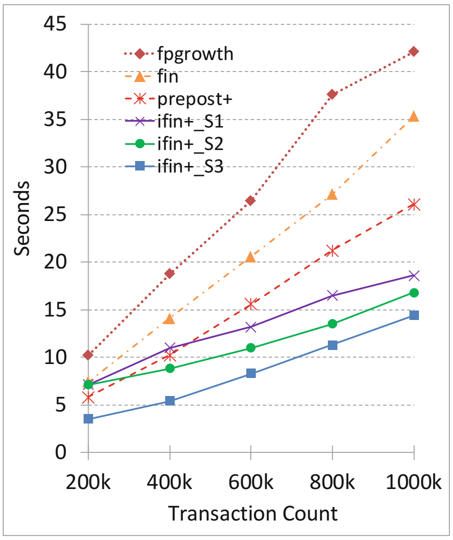


Fig. 6. Running time on the incremental Kosarak datasets

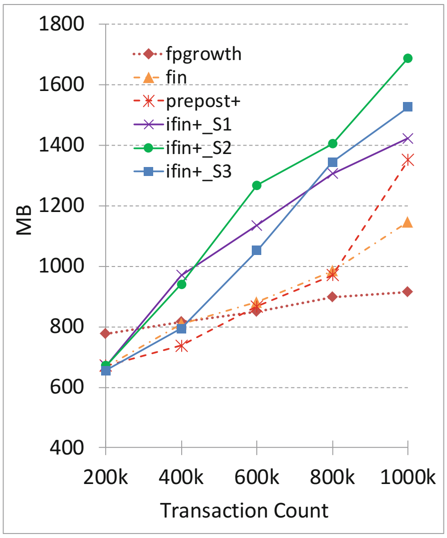


Fig. 7. Peak memory on the incremental Kosarak datasets

Figures 6 and 7 respectively visualize the algorithms' running time and peak used memory in data accumulation steps at the support threshold $\varepsilon = 0.2\%$. Like experiments on the synthetic dataset, for all algorithms, the running time and used memory increase linearly when the dataset is accumulated. Among the three algorithms FP-Growth, FIN and PrePost$^+$, the orders in the memory efficiency and the performance are similar to that in case of the synthetic dataset. FP-Growth still is the slowest algorithm, but it uses memory more efficient than PrePost$^+$ and FIN. While PrePost$^+$ runs remarkably faster than FP-Growth and FIN, it becomes to consume more memory than FP-Growth and FIN follow the data accumulation steps.

Shown in Fig. 6, the execution time of IFIN$^+$ is approximate that of PrePost$^+$ and FIN algorithms in the first two data accumulation steps; but for the following ones, IFIN$^+$ becomes to run faster than the others. In Fig. 7, IFIN$^+$ uses memory as good as the others at the dataset of 200k transactions. However, its consumed memory increases faster than the other algorithms' and is the most for larger sizes, approximate the PrePost$^+$'s memory at full size of Kosarak dataset in execution scenarios S1.

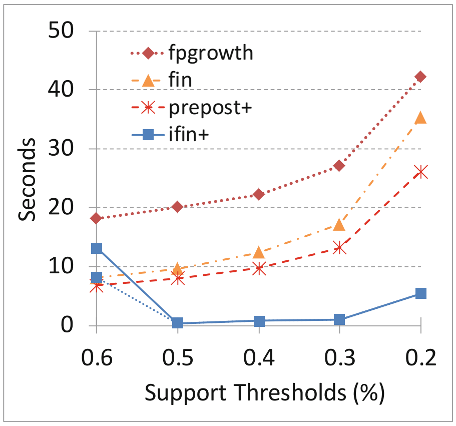
Table 7. Tree construction time of the algorithms for Kosarak dataset

	200k	400k	600k	800k	1000k
IPPC$^+$ tree (Scenario S1)	5.1 s	6.8 s	7.2 s	8.3 s	8.7 s
IPPC$^+$ tree (Scenario S3)	0.5 s	0.7 s	1.3 s	1.5 s	2.2 s
FP tree (ε = 0.002)	3.3 s	6.2 s	9.2 s	12.6 s	15.3 s
POC/PPC (ε = 0.002)	2.1 s	3.7 s	5.1 s	6.6 s	7.5 s

As we knew that the IPPC$^+$ tree of IFIN$^+$ is a compact structure of all items in a dataset, but the trees of the other three algorithms depend on the support threshold and contain only frequent items in a dataset. Looking into Table 3 for the reason of the memory used by IFIN$^+$, the synthetic dataset comprises a considerable percentage of frequent items, [90%–56%] for the support threshold $\varepsilon \in$ [0.001–0.006]; but just a very small quantity, [1.38%–0.28%] for $\varepsilon \in$ [0.002–0.006], is for frequent items in Kosarak dataset. Therefore, in the case of Kosarak dataset, the used memory to maintain the IPPC$^+$ tree of IFIN$^+$ is much larger than that of the trees of FP-Growth, FIN and PrePost$^+$; while the affection of this disadvantage to IFIN$^+$ is not considerable in the synthetic dataset case.

Beside the memory, the computational overhead to construct the IPPC$^+$ tree is also affected. Table 7 reports the running time for building the trees of algorithms on Kosarak dataset. The tree constructions of FIN and PrePost$^+$ on this dataset are very efficient and take only 7.5 s for the full size of Kosarak dataset. The gap in tree construction time between IPPC$^+$ tree (S1 scenario) and the tree of FIN/PrePost$^+$ is gradually reduced follow data accumulation steps. The tree construction in S3, just by loading the built tree, takes the least time and once again asserts its very high performance.

Figures 8 and 9 depict the running time and the peak memory of the algorithms mining on the full Kosarak dataset with different support thresholds ε. Start at ε = 0.6%, IFIN$^+$ can perform one of two scenarios S1 or S3 that their two running time values are shown in Fig. 8. For other ε values, IFIN$^+$ just runs some portions of its mining tasks and reuses completely the built tree. Therefore, the same results as the corresponding experiments on the synthetic dataset in Fig. 4, the running time of IFIN$^+$ takes an overwhelming advantage against that of the three remaining algorithms. IFIN$^+$ consumes the most memory since it needs more memory to maintain IPPC$^+$ tree. FP-Growth uses memory less efficient than the two algorithms FIN and PrePost$^+$ for $\varepsilon > 0.3\%$. However, its consumed memory becomes lesser than other algorithms' for $\varepsilon \leq$ 0.3%. FP-Growth's running time is considerably longer than that of FIN and PrePost$^+$. Algorithm PrePost$^+$ runs faster than FIN, but this dominance of PrePost$^+$ is not significant.



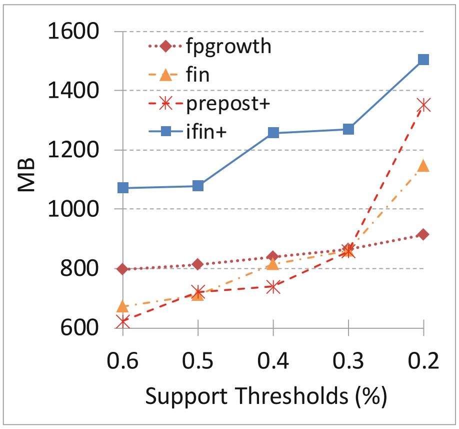


Fig. 8. Running time on Kosarak dataset at different support thresholds

Fig. 9. Peak memory on Kosarak dataset at different support thresholds

4 Conclusions

In this paper, we proposed an improved version of the IPPC tree, called IPPC$^+$, to enhance the performance of the tree construction. Beside the minor improvement in mining performance, the experiments show that with a datasets comprising a small number of distinct items, the tree construction performance of the two versions are approximate to each other; but in case of a dataset with a large number of distinguishing items, the tree construction performance of IPPC$^+$ tree is improved remarkably compared to that of IPPC tree. This contributes to significantly reducing the disadvantage in the tree construction phase of algorithm IFIN$^+$ compared to other algorithms such as FIN, PrePost$^+$ in cases of datasets with a huge number of distinct items but just a small percentage of frequent items.

Experiments also demonstrated that IFIN$^+$ is superior in performance compared to the three remaining algorithms, especially in mining circumstances when its incremental characters take effect. This provides IFIN$^+$ an efficient way to deal with the high-velocity property of Big Data and the data mining practices which often try with different threshold values.

In case of a dataset including a huge number of distinct items but just a small percentage of frequent items, IFIN$^+$ algorithm needs more memory than the other algorithms to retain its tree structure of all items in the dataset. However, when mining with small enough support thresholds, the gap in memory overhead between IFIN$^+$ and the other algorithms will be reduced; and when a dataset is more and more accumulated, all algorithms must face with the problem of memory scalability besides the running time. Therefore, as a potential approach, a distributed parallelization solution for IFIN$^+$ will be proposed to better confront with these problems of Big Data.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of 20th International Conference on VLDB, pp. 487–499 (1994)
2. Han, J., Pei, J., Yin, Y.: Mining frequent itemsets without candidate generation. ACM SIGMOD Rec. **29**(2), 1–12 (2000)
3. Deng, Z.-H., Lv, S.-L.: Fast mining frequent itemsets using nodesets. Expert Syst. Appl. **41** (10), 4505–4512 (2014)
4. Deng, Z.-H., Lv, S.-L.: PrePost$^+$: an efficient N-lists-based algorithm for mining frequent itemsets via children-parent equivalence pruning. Expert Syst. Appl. **42**(13), 5424–5432 (2015)
5. Market-Basket Synthetic Data Generator. https://synthdatagen.codeplex.com/
6. Frequent Itemset Mining Dataset Repository: Kosarak, online news portal click-stream data. http://fimi.ua.ac.be/data/kosarak.dat.gz
7. Savasere, A., Omiecinski, E., Navathe, S.: An efficient algorithm for mining association rules in large databases. In: VLDB, pp. 432–443 (1995)
8. Perego, R., Orlando, S., Palmerini, P.: Enhancing the *Apriori* algorithm for frequent set counting. In: Kambayashi, Y., Winiwarter, W., Arikawa, M. (eds.) DaWaK 2001. LNCS, vol. 2114, pp. 71–82. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44801-2_8
9. Park, J.S., Chen, M.S., Yu, P.S.: Using a hash-based method with transaction trimming and database scan reduction for mining association rules. IEEE Trans. Knowl. Data Eng. **9**(5), 813–825 (1997)
10. Zaki, M.J.: Scalable algorithms for association mining. IEEE Trans. Knowl. Data Eng. **12**(3), 372–390 (2000)
11. Grahne, G., Zhu, J.: Fast algorithms for frequent itemset mining using FP-trees. Trans. Knowl. Data Eng. **17**(10), 1347–1362 (2005)
12. Liu, G., Lu, H., Lou, W., Xu, Y., Yu, J.X.: Efficient mining of frequent itemsets using ascending frequency ordered prefix-tree. DMKD J. **9**(3), 249–274 (2004)
13. Shenoy, P., Haritsa, J.R., Sudarshan, S.: Turbo-charging vertical mining of large databases. In: 2000 SIGMOD, pp. 22–33 (2000)
14. Zaki, M.J., Gouda, K.: Fast vertical mining using diffsets. In: 9th SIGKDD, pp. 326–335 (2003)
15. Huynh, V.Q.P., Küng, J., Dang, T.K.: Incremental frequent itemsets mining with IPPC tree. In: Benslimane, D., Damiani, E., Grosky, W.I., Hameurlain, A., Sheth, A., Wagner, R.R. (eds.) DEXA 2017. LNCS, vol. 10438, pp. 463–477. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64468-4_35
16. Huynh, V.Q.P., Küng, J., Jäger, M., Dang, T.K.: IFIN$^+$: a parallel incremental frequent itemsets mining in shared-memory environment. In: Dang, T.K., Wagner, R., Küng, J., Thoai, N., Takizawa, M., Neuhold, E.J. (eds.) FDSE 2017. LNCS, vol. 10646, pp. 121–138. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70004-5_9