Signature-based Tree for Finding Frequent Itemsets

Mohamed El Hadi Benelhadj, Mohamed Mahmoud Deye, and Yahya Slimani

Original scientific article

Abstract—The efficiency of a data mining process depends on the data structure used to find frequent itemsets. Two approaches are possible: use the original transaction dataset or transform it into another more compact structure. Many algorithms use trees as compact structure, like FP-Tree and the associated algorithm FP-Growth. Although this structure reduces the number of scans (only 2), its efficiency depends on two criteria: (i) the size of the support (small or large); (ii) the type of transaction dataset (sparse or dense). But these two criteria can generate very large trees. In this paper, we propose a new tree-based structure that emphasizes on transactions and not on itemsets. Hence, we avoid the problem of support values that have a negative impact on the generated tree.

***Index terms*—Data mining, Data compression, Data storage, Tree structure, Signature.**

I. Introduction

The efficient search of information in large datasets to extract knowledge is vital for any expert. Several methods and techniques are used in KDD (Knowledge Discovery in Databases) process to extract knowledge from large datasets. Mining association rules which trends to find interesting association or correlation relationships among large amounts of data is one of these techniques. Originally introduced by Agrawal [1] in the context of transactional datasets, the association rule mining approach is now used extensively to find associations in biological datasets, web log data, telecommunications data, census data, social data and other types of datasets [21].

Though several algorithms have been developed for fast mining of frequent itemsets over the years [20], [7], [27], [8]. Association rule mining algorithms can be classified into two categories: the first one is based on the candidate generate and test approach, such as Apriori [2], [6] while the second one is based only on the pattern fragment growth like the FP-Growth or frequent itemset-growth algorithm [20].

The "*generate and test approach*" is based on an anti-monotone property [1]: if an itemset with k items is not frequent, any of its super-itemset with (k+1) or more items can never be frequent. So, this approach iteratively generates a set of candidate itemsets of length (k + 1) from the set of frequent itemsets of length k ($k \geq 1$) and their corresponding occurrence frequencies are checked in dataset.

Manuscript received July 27, 2022; revised December 12, 2022. Date of publication March 14, 2023. Date of current version March 14, 2023. The associate editor prof. Toni Mastelić has been coordinating the review of this manuscript and approved it for publication.

M. El Hadi Benelhadj is with the Faculty of Science and Technologies, Tamanrasset University, Algeria (Mohamed.benelhadj@univ-tam.dz).

M. M. Deye is with the Cheikh Anta Diop University of Dakar, Senegal (mohamed.oulddeye@ucad.edu.sn).

Y. Slimani is with the Institute of Multimedia Art of Manouba (ISAMM), University of Manouba, Tunisia (yahya.slimani.guest@isamm.uma.tn).

Digital Object Identifier (DOI): 10.24138/jcomss-2022-0065

Though this algorithm works relatively well with smaller dataset. However, when we have a large number of frequent patterns and/or long patterns, the "*generate and test approach*" may still suffer from huge number of candidates and needs many scans of large datasets for frequency checking.

The pattern-growth approach, such as FP-Growth (Frequent Pattern-Growth) also uses the anti-monotone property. In this approach, the dataset is recursively split into sub-datasets according to the frequent itemsets found and search for local frequent itemsets to assemble longer and larger ones.

FP-Growth avoids candidate generation by compressing the transaction dataset into a structure called FP-Tree. Nevertheless, this algorithm may still encounter difficulties for large sparse datasets when the FP-Tree will be very large [20].

Finally, to improve the efficiency of the association rule mining algorithm, the Apriori-like algorithms and FP-Tree-based algorithms have been used on various types of datasets with varying degrees of success. But, generally, the problem of repeatedly scanning the datasets remains.

In this paper, we propose a new data structure, called FI-Tree (Frequent Itemset Tree), to represent a transaction dataset and a new mining algorithm, FI-Mine, to extract the frequent itemsets. With FI-Mine algorithm, in the first time, we scan the dataset only once to generate a binary signature for each transaction, to construct the FI-Tree and to extract the frequent 1-itemsets. In second time, the step of extraction frequent itemsets is done. It assigns a signature for each k-itemset candidate ($k \geq 2$), searches it in the FI-Tree, computes its support and keeps only the frequent k-itemsets.

The reminder of this paper is organized as follows: in Section II we present and discuss a state of the art about the concept of signature and its different representations. Section III reviews the main used of signature file for tree structure. In Section IV, we present our proposed structure FI-Tree and we compare it with FP-Tree. Section V compares and discusses our algorithm FI-Mine with Apriori algorithm using the response time as a comparative criterion. We also compare it with FP-Growth using the memory space as a comparative criterion. Finally, Section 6 concludes and skittles future research works.

II. Related Works

The role of indexing is evident for accelerating data recovery from large datasets. Index techniques have been extensively investigated in both the information retrieval and dataset research areas, and many methods have been developed within the past three decades. Being efficient in evaluating set-oriented query and allowing the easy handling of updates and insert operations makes the signature file techniques are the best indexing approaches. Signature file representation has been tackled using several techniques. We cite, for instance, Bit-Slice Signature file [12], Sequential Signature File [12],

Multilevel Signature file [25], [12], Signature Graph [10], and Compressed Bit Sliced Signature File [12].

We cite, for instance, Bit-Slice Signature file [12], Sequential Signature File [12], Multilevel Signature file [25], [12], Signature Graph [10], and Compressed Bit Sliced Signature File [12].

A. Signature File

A signature is a string of bits constructed from a defined value. It is better than other indexing approaches, since it allows the efficient treatment of new insertions and queries on word parts. It is also simple to implement. Besides, it works well on large files. Comparatively, to other index structures, the signature file is more efficient at processing new insertions and queries on words. Other advantages include the simplicity of its implementation and its ability to support large files. Though, it suffers from the problem of information loss. The careful selection of the signature extraction technique allows to minimizing this loss. A signature is a binary vector of length m obtained by applying one (or several) hash function(s) [12].

Several methods to extract signatures such as Superimposed Coding (SC), Variable Bit-block Compression (VBC), Word Signature (WS), Bit-block Compression (BC), Run Length Encoding (RL) [15], [16] and Multilevel Superimposed Coding (MSC) [24], [17] have been developed.

The text block signature is the result of combining all the signatures of words composing it by the "OR" logical operation. A signature file is formed by the set of all signatures. Table I shows an example of signature extraction of a block ("frequent itemset extraction").

TABLE I
SIGNATURE EXTRACTION EXAMPLE

Frequent	0000	0000	0000	0010	0000
Itemset	0000	0001	0000	0000	0000
Extraction	0000	1000	0000	0000	0000
Bloc Signature	0000	1001	0000	0010	0000

B. Signature Representation

Signature files methods are described in this sub-section.

B.1 Sequentiel Signature File (SSF)

SSF is the simplest organization which requires low algorithm and low update cost. It is also easy to implement. The signatures are stored sequentially in the signature file. When a query is given, a full scan of the signature file is required [4]. Therefore, it is generally slow in retrieval.

B.2 Bit Sliced Signature File (BSSF)

A column-wise way is used here to store signatures. If the length of the signatures is m, then all the signatures will be stored in m files. So, for each bit position of all signatures, one bit-slice file F is created. For extraction, only part of the m bit-slice files should be scanned. Hence, the search cost is reduced and is lower than that of SSF. However, update cost becomes larger. For example, an insertion of a new signature requires about m disk accesses, one for each bit-slice [12].

B.3 Compressed Bit Sliced Signature File (CBSSF)

The number of 1's is ensured by the adoption of a suitable hash function for signature extraction. To limit the false drop, the length of signature should be increased. By doing so, a sparse matrix, which is easily compressible, is created [4]. This matrix can be compressed by replacing each 1 with its associated physical address. The hash table uses a set of pointers to the head of the linked list. For instance, if the word "Text" has "1" as the first bit, and it is positioned at the 50th byte of the text file, then by looking at the 1st bucket, the position of the word "Text" is found. Although this method saves space, the number of false drops increases because the signature files are sparse.

B.4 Multi-level Signature File

It is a structure similar to S-Tree, but different in that a signature at a higher level is a superimposed code generated directly from a group of text blocks, instead of superimposing signatures at a lower level. However, this method needs more subspace. An improved method for multi-level signature file is discussed in [14].

B.5 Signature Graph

A tree like structure is used to organize the signature file. Through, to match a given query, the path explored in the graph is not a continuous chain of bits. It corresponds to a signature identifier, which makes the difference between the signature graph and the tree [10]. Unlike signatures, no compact representation is used for the search path; the length is not the same for all queries. This means that the graph is not balanced, and in worst cases, it is reduced to a signature file.

III. BACKGROUND

We are mainly interested in representation of the signature file in tree structure. In the following, we present the main used representations.

A. Signature Representation as a Tree

A.1 S-Tree

An S-tree is a height balanced multi-way tree. Each internal node corresponds to a page, which contains a set of signatures and each leaf node contains a set of entries of the form, where the object is accessed by an oid and s is its signature. Combining lower-level nodes permits the construction of internal nodes. The advantage is that it is not needed to search the whole signature file; instead, a simple tree is searched. However, the combination of nodes results in the situation in which the internal node at higher level is likely to have more weight. This is decreasing selectively. The authors in [14] enhanced the S-Tree technique by proposing some new split methods, such as Linear split, Quadratic split, Cubic split and hierarchical clustering. By doing so, the query response time was improved.

A.2 Signature Tree

A tree of signatures T_s represents a set of signatures $S = \{S_1,...,S_n\}$, where $S_i \neq S_j$ for all $i \neq j$ and $\left| S_k \right| = m$, for $1 \leq k \leq n$. T_s is a binary tree in which:

- The left edge issued from it is tagged with a "0" while the right one is tagged with "1".
- T_s has n leaves tagged $L_1,\ldots, L_m$. There are used as pointers to m different signatures $S_1,...,S_m$ in S.
- A positive number, annotated Pos(v), is associated with every node v. Its role is telling which bit to be checked.

The bit positions given by the nodes are used to identify signatures. However, in the case of query signatures, the tree is explored from top to bottom with respect to the bit positions defined by the nodes, instead of the 1's given by the query signature. Besides, for matching 1's, the right sub-tree is explored, while both left and right sub-trees are searched to match 0's [9],[10],[12].

A.3 Signature Declustering Tree (SD-Tree)

Three types of nodes are used in the SD-Tree: Internal nodes, Leaf nodes and Signature nodes. The two first types are similar to their counter parts in B+ trees. The internal nodes compose the upper tree, while the leaf nodes are the components of the pre-ultimate level. The signature nodes are situated at the bottom level of the SD-Tree [25].

B.Representation of Transaction Dataset

To compute the supports of itemsets, we need to access the transaction dataset. As transaction datasets are usually very large, one solution, which avoids repetitive and costly accesses of these datasets, may be to represent them by compact structures in order to optimize memory usage. Many structures have been proposed with the aim of optimizing memory use, reducing I/O costs and making processing faster.

In this subsection, we will present and discuss the characteristics of the main tree structures used for frequent itemset generation.

B.1 FP-Tree Structure

The FP-Tree structure is an extension of the Trie data structure [6] which belongs to the family of prefixed trees. The FP-Tree combines vertical and horizontal data representation schemes. The proposal of the FP-Tree has been a starting point for the development of algorithms for the extraction of frequent itemsets without candidate generation [3],[20].

The FP-Tree is a compact structure consisting of:

- A tree whose root has the value "null" and, where each node, other than the root, contains three pieces of information: the item representing the node, its frequency and the next node in the tree.
- An index contains the list of frequent itemset. Each item is associated with a pointer that indicates the first node of the tree where this item appears. The construction of the FP-Tree requires two accesses of the transaction dataset and is done as follows: the first access is used to determine the number of occurrences of each item, to eliminate the infrequent items and to order the frequent items by decreasing order of support. During the second access, the items of each transaction are sorted according to the order obtained during the first access (see Table II).

TABLE II
ORDERED TRANSACTIONS DATASET

Tid	Transactions	Ordered Transactions
T1	1, 2, 5	2, 1, 5
T2	2, 4	2, 4
T3	2, 3	2, 3
T4	1, 2 ,4	2, 1, 4
T5	1, 3	1, 3
T6	1, 2, 3, 5	2, 1, 3, 5
T7	1, 2, 3	2, 1, 3

The tree building process starts with the creation of a root node. Then, a branch is added to the tree for a processed transaction, exploiting the fact that transactions with the same prefix will share the same start node of the tree branch (Figure 1).

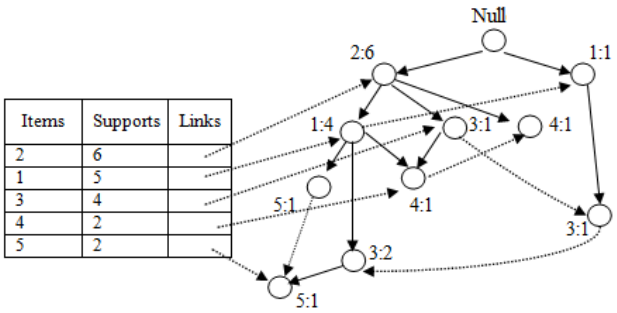

Fig. 1. FP-Tree example

Next, the frequent pattern mining process is converted into an FP-Tree mining process. For each frequent item, construct its corresponding conditional pattern base and FP-Tree. Repeat this process for each new constructed FP-Tree, until it is empty or contains only one path. When the constructed FP-Tree is empty, the prefix is the frequent pattern. When it contains only one path, all frequent patterns can be acquired by connecting their prefixes with all possible combinations enumerated from the path (Table III).

TABLE III
CONDITIONAL FP-TREE

Item	Conditional Pattern Base	Conditional FP-Tree
5	{{2, 1: 1}, {2, 1, 3: 1}}	{2: 2, 1: 2}
4	{{2, 1: 1}, {2: 1}}	{2: 2}
3	{{2, 1: 2}, {2: 2}, {1: 2}}	{2: 4, 1: 2}, {1:2}
1	{{2: 4}}	{2: 4}

B.2 H-struct Structure

The main disadvantage of the FP-Tree structure is the explosion of the tree size for sparse transaction datasets, for which the representation by an FP-Tree generates a tree of the same size as the sizes of these datasets. To overcome this drawback, a structure called H-struct has been proposed [22]. In this structure, transactions are ordered in an arbitrary way. Only frequent items are projected into the H-structure. An H-structure consists of projected transactions, and each node in these transactions contains the label of the item and a hyperlink to the next occurrence of this item. A header table is created for each H-struct. This table contains the frequencies of all items

and a given hyperlink to the first transaction containing the item.

B.3 CATS Tree

A CATS tree is an extension of the FP-Tree, and it is a pre-expressed tree containing all elements of the FP-Tree including the header table, pointers to item occurrences, etc. Each item in the transaction dataset has a node in the header table, containing the total frequency of that item in all transactions. In addition, each node in the header table contains a pointer to the first node within the tree, bearing the same label as the node in the header table. Each node in the tree contains the label of the item, its frequency, a pointer to its parent, pointers to its l's and a pointer to the next occurrence of that item to form a doubly-linked list connecting all nodes with the same label. The l's of a node, in a CATS tree, are arranged in descending order of their frequency. The paths from the root to the leaves represent all the transactions in the dataset. In fact, a CATS tree is an FP-Tree rearranged to improve compression. Unlike the FP-Tree structure, where the l's of each node are ordered in descending order of their global supports, the l's of each node in a CATS tree are ordered relatively to their local supports [13].

B.4. Patricia Tree

A Patricia Tree is a compression of the FP-Tree [26]. In this tree, each maximal chain of nodes v1→ v2→…→ vk, whose nodes vi have the same support c except vk, is merged into a single node of support c; this node has vk with its associated support as l's. The size of a Patricia Tree, representing a base of M transactions of average length N, is at most equal to N+O(M). We can illustrate this compression gain by Figure 4, which represents a FP-Tree and its version as a Patricia Tree. From Figure 4, we notice that the Patricia Tree is more compact in number of nodes. All adjacent nodes that share the same support are merged into a single node.

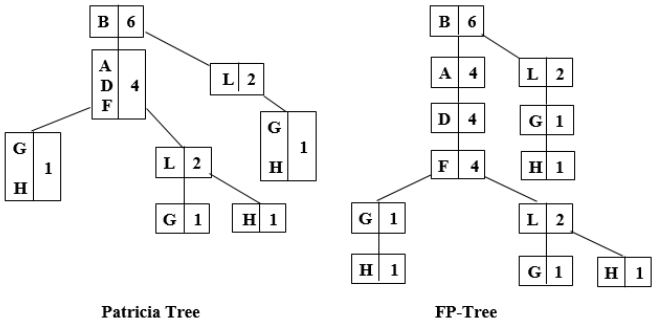

Fig. 2. FP-Tree and its compression into Patricia-Tree

IV. Our Main Contribution

A. Our Proposed Structure: FI-Tree

Improving performance of discovering association rules requires an optimization of the extraction phase of frequent itemsets. To reach this objective, we propose to use the Frequent-Itemset-Tree (FI-Tree) structure representing the set of transaction signatures. Each transaction is represented by a signature of size m. Signatures are abstractions of items, which are coded. They are represented by a binary correspondence with a specified number of 1's. The transaction signatures are formed by combining the signatures of items. FI-Tree has the advantage of being both a compact (binary representation) and dynamic (care of updates) structure. A signature tree contains two types of nodes: internal nodes and leaf nodes. For each internal node, the left child corresponds to the value "0" and the right one to the value "1". Each leaf node contains a signature S_i. The number of leaf nodes in a FI-Tree is equal to the number of signature transactions. The construction of the FI-Tree requires two phases:

1. The application of the hash function H(item) (for example, we use modulo 5 function) to obtain the signature for each item into a transaction. The superimposed coding of these signatures will give the transaction signatures. An example of signatures generation is given below (see Table IV).
2. Each transaction signature S_i is inserted in the FI-Tree and a leaf of this tree is a signature.

TABLE IV
TRANSACTIONS AND SIGNATURES

Tid	Transactions	Signatures
T1	1, 2, 5	S_1:11100
T2	2, 4	S_2:00101
T3	2, 3	S_3:00110
T4	1, 2 ,4	S_4:01101
T5	1, 3	S_5:01010
T6	2, 3	S_6:11110
T7	1, 2, 3, 5	S_7:01110

A.1. A Simple Way for Constructing FI-Tree

At the beginning, the tree has an initial node containing the first signature transaction. Then, we take a new signature transaction, a composition of signatures items, and we insert it into the FI-Tree. Let S be the signature we wish to enter. We cross the tree from the root. Let v be the node encountered and assume that v is an internal node with position (v) = k. Then, S[k] will be checked. If S[k] = 0, we go left, otherwise, we go right. Let v be a leaf node and S' its corresponding signature; we compare S with S'. We assume that the first k bits of s agree with S'; but S differs from S' in the $(k+1)^{th}$ position. We construct a new node u with position (u) = k+1, replace v with u and v becomes one of u's children. If position (u) = 1, we make v be the left and s be the right child of u, respectively. If position(u) = 0, we make v the right child of u and s the left child of u. In the following, we describe formally the FI-Tree construction process.

- Steps to generate FI-Tree

Steps (a) build a root node r such that r is a leaf node that contains the signature S_1.

(a) Insert (S_1 = 11100)

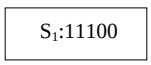

Fig. 3 (a)

(b) Insert (S_2 = 00101): the 1st different bit between S_1 and S_2 is the 1st bit such that $S_1[1] = 1 \neq S_2[1] = 0$; hence, we create internal node v with pos(v) = 1, right leaf node S_1 and left leaf node S_2.

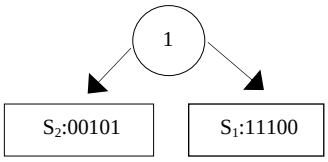

Fig. 3 (b)

(c) Insert (S_3 = 00110): $S_3[1] = 0$, we go to the left and we compare S_3 with S_2. The 1st different bit between S_3 and S_2 is the 4th bit, $S_2[4] = 0 \neq S_3[4] = 1$; hence, we create internal node v with pos(v) = 4, left leaf node S_2 and right leaf node S_3.

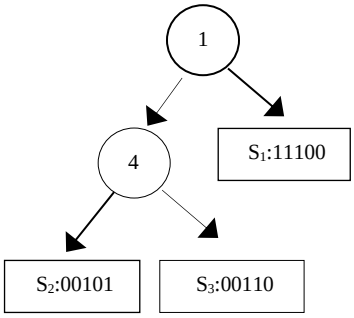

Fig. 3 (c)

The insertion of the signature S_7 = 01110 completes the process of building the FI-Tree (Figure 3(d)).

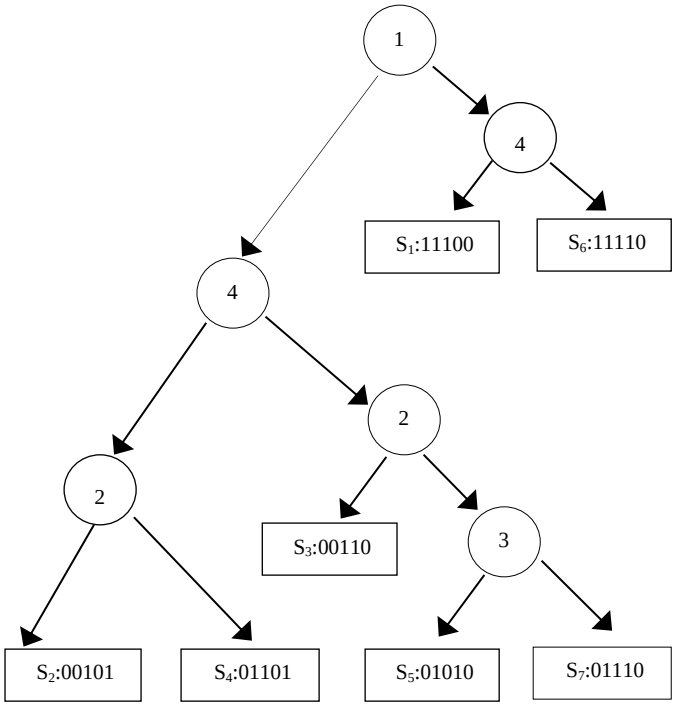

Fig. 3 (d). Final FI-Tree

The formal description of the algorithm FI-Tree-Construction is given in Table V.

TABLE V
FI-TREE-CONSTRUCTION ALGORITHM

```
Algorithm FI-Tree-Construction
Input: Set of transactions
Output: FI-Tree

Begin
  Generate all signatures (S1, ..., Sn)
  Insert signatures in FI-Tree
End
```

The formal description of the algorithm Insert (S_i) is given in Table VI.

TABLE VI
INSERT ALGORITHM

```
Algorithm Insert
Input: The signature Si, FI-Tree
Output: FI-Tree

Begin
  Traverse the tree from the root
  While Stack not empty Do
      v ← Pop (Stack)
      If v is an internal node Then
        Check Si (p)
        If Si[j] = 1 Then
              Push (Stack, right_child)
        Else
              Push (Stack, left_child)
        Endif
      Else
        Generate a new internal node u
        Generate a new leaf node vi = {Si}
      Endif
  Enddo
End
```

A.2. Searching in FI-Tree

Now, we discuss how to search a signature S_I of an itemset I in the FI-Tree structure:

1. Let v be the node encountered and position (v) be the position to be checked.
2. If position (v) = 1, we move to the right child of v.
3. If position (v) = 0, both the right and left child of v will be explored.

In fact, this process corresponds to the signature matching criterion. For a bit position p in S_I, if it is to "1", the corresponding bit position in S (S is a signature transaction) must be set to "1"; if it is set to "0", the corresponding bit position in S can be equal to "1" or "0". The following example helps for illustrating the main idea of the algorithm.

Example 1. Consider an itemset I{2,3,5} and its signature S_I = 10110. Then, only part of the FI-Tree will be searched (thick edges in Figure 4). On reaching a leaf node v, the signature S will be checked against S_I. In our example, we visit 2 signatures S_1 and S_6, but we select only S_6 because S_1 doesn't contain S_I (Figure 4).

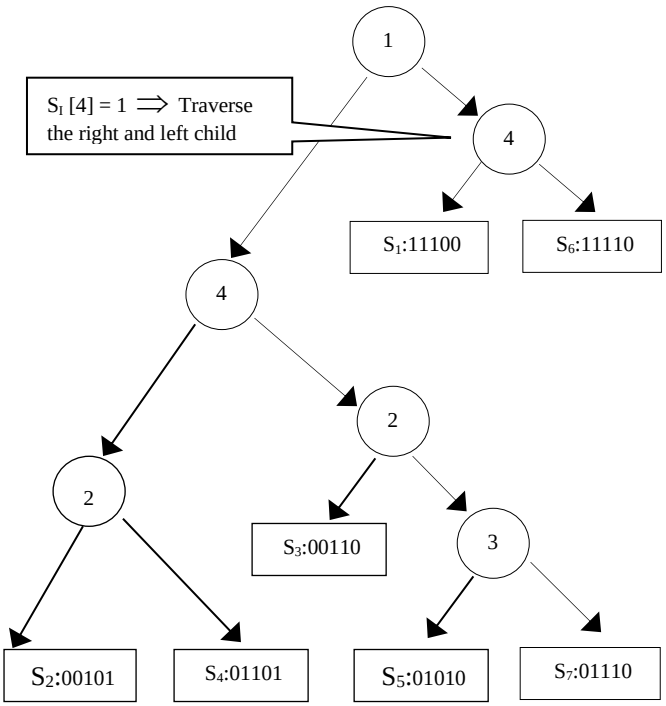

Fig. 4. Signature Search Process

In Table VII, we present the formal description of the search algorithm.

TABLE VII
FI-TREE SEARCH ALGORITHM

```
Algorithm FI-Tree-Search
Input: An itemset I
Output: Support(I)

Begin
    S_I = Gen_Signature (I)
    Calcul Support (S_I)
End
```

B. Discovering Frequent Itemsets

The generation of frequent itemsets computes, for each candidate itemset I, its support, denoted Support(I), and compares it to a minimum support denoted Minsup, a threshold fixed by the user. An itemset I is said to be frequent if Support(I) ≥ Minsup.

A signature of candidate itemset S_I is created using the same method as a transaction signature. To find the signature S_I in the FI-Tree, we select all transaction signatures S_T, such that ($S_T \wedge S_I$) = S_I, S_I is called a drop and ∧ is the superimposed operator. Many unqualified transaction signatures are immediately rejected. This method guarantees that all the qualifying transaction signatures will be selected. All signatures are used to compute the itemset supports. This support is used in the extraction process of frequent itemsets. If the associated value of an itemset is less than a specified user threshold, this itemset is said to be not frequent.

B.1. Algorithms

Tables VIII and IX illustrate the formal description of the extraction algorithm of frequent itemsets and the FI-Mine algorithm.

TABLE VIII
EXTRACTION-FI

```
Algorithm Extraction-FI
Input: Frequent 1-Itemsets, FI-Tree
Output: The set of frequent k-itemsets

Begin
    /* Initially, FI = {Frequent 1-itemsets} */
    k ← 2
    1. Generate a candidate k-itemset I
        Call FI-Tree-search (I, Support(I))
        If Support(I) ≥ Minsup Then
            FI ← FI ∪ {I}
        Endif
    2. k ← k+1
    3. Repeat steps 1 and 2 until no candidate k-itemset
    4. Return (FI)
End
```

Example 2. Consider the itemset I = {2,3,5} and his signature S_I = 10110 of example 1 and Minsup = 2. The selected signature is S_6; then Support (I) = 1 < Minsup. We conclude that the itemset I is not frequent.

TABLE IX
FI-MINE ALGORITHM

```
Algorithm FI-Mine
Input: {Transactions}
Output: {Frequent Itemsets}

Begin
    FI-Tree-Construction
    Extraction-FI
End
```

B.2. Complexity Study

The algorithm FI-Tree-construction has a complexity of O(n*m), where n is the number of transaction signatures and m the size of a signature.

For against, the algorithm Insert(S_i) requires one tree parsing for the first signature, 2 for the second, and so on. The number of path traversed is:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2} + (n^2 + n) \qquad (1)$$

Hence, the associated complexity is about $O(n^2)$.

The complexity of the search procedure is of order $(n/2^k)$, where n is the number of transaction signatures and k the number of bits set to "1" in the signature. In the worst case, this complexity is about:

$$O(n/2^m) \approx O(n) \qquad (2)$$

The Extraction_FI algorithm contains a loop that is run p times (p being the number of candidate itemsets). Complexity to handle the candidate itemsets is equal to p times the search procedure, so it is in the order of:

$$O(p(n/2^k)) \approx O(pn) \qquad (3)$$

Finally, the complexity of FI-Mine is polynomial and equal to

$$O(nm) + O(n^2) + O(pn) \qquad (4)$$

C. FI-Tree vs FP-Tree

In this sub-section, we compare our proposed tree structure (FI-Tree) with the will know FP-Tree. This comparison is made on criteria contained in Table X.

TABLE X
COMPARISON BETWEEN FP-TREE AND FI-TREE

Criteria	FP-Tree	FI-Tree
#Scans	2	1
#Trees	1+k	1
Nodes	Items	Decimal values
Leaves	Items	Binary signatures
Tree structure	Support-dependent	Support-independent
Additional data structures	Tables and Conditional FP-Trees	No
Dataset type	Type-dependent (sparse or dense)	Independent
Redundancy	Yes (nodes)	No
Updating dataset	Rebuild or not rebuild tree	Not rebuild tree

C.1. Number of scans

To build FP-Tree structure, two scans of the dataset are needed: the first one allows to calculate the occurrence frequency of 1-itemsets and to build the head table containing only the frequent 1-itemsets. Then, another scan is required to build the FP-Tree and to complete the associated head table with additional information's. In our proposal, only one scan is required to build the FI-Tree and extract the frequent 1-itemsets. This is the first fundamental difference between our proposal and the FP-Tree approach in terms of scans. This difference is not negligible, especially for large transaction datasets.

C.2. Trees

In terms of number of generated trees, we remark a significant difference between our approach and that of FP-Tree. For FP-Tree, (1+k) trees are generated. The first tree (the main tree) contains only the frequent 1-itemsets. For others frequent k-itemsets (k varies between 2 and the maximal frequent itemset, namely k), k FP-Trees, called the conditional FP-Trees, are generated from the main one. In addition to these k FP-Trees, k conditional tables are also generated. In our proposal, only one tree is generated. This difference can be explained by the fact that FP-Tree depends on the support value, while our proposal is independent of any support value.

If we analyze the type of tree used, we remark that FP-Tree and FI-Tree use two types of trees that have different properties. For FP-Tree, the tree structure is free in the sense where a node can have one or several children. In addition, there is no semantic difference between a node and a leaf. Both correspond to items.

In our proposal tree, we use an interesting kind of trees, namely binary tree. These last have interesting properties [25]:

- FI-Tree is a full binary tree.
- Searching in binary tree become faster.
- Make insertion and deletion operations faster than linked lists and arrays.
- A flexible way of holding and moving data.
- Binary trees are used to store as many nodes as possible.

C.3. Nodes

The main concept in a FP-Tree is an item. For this purpose, all nodes of a FP-Tree are associated to the items of the transaction dataset. In contrast, in our proposal the main concept is a transaction. All the transactions are represented in the leaves of our tree. Because FP-Tree is item-based, the nodes of a FP-Tree are linked with paths. These paths connect items that belong to the same transaction. Since the intersection of transactions is not empty, one or more items are shared between different paths, which creates redundancy

C.4. Leaves

In a FP-Tree, there are no difference between nodes and leaves. Both represent an item of the transaction dataset. In our proposal, nodes are not related to items while leaves represent signature transactions of the dataset. The nodes have no semantic relation with the items or the transactions of a dataset. They only represent path links to access to the leaves containing signatures of transactions.

C.5. Tree structure

For FP-Tree, the worst case occurs when every transaction has a unique itemset and so the space needed to store the tree is greater than the space used to store the original dataset. This case is justified by the fact that FP-Tree requires additional space to store pointers between nodes and the counters for each item.

In our proposal, the FI-Tree is totally independent of itemsets in the dataset. With this important property, our proposal does not require additional data structures or space to represent a dataset.

C.6. Additional data structures

In addition to the main FP-Tree, different data structures are required to find k-frequent itemsets (k>1). Firstly, we must construct a main table corresponding to the main FP-Tree. This table contains frequent 1-itemsets with their associated supports and must be sorted on decreasing support values. Then, the FP-Growth algorithm generates a huge number of conditional FP-Trees. Hence, FP-Tree is expensive to build. In our proposal, no additional data structure is required. Only the signature tree is needed.

C.7. Dataset type

Generally, there are two types of datasets: sparse and dense. In the FP-Tree structure, this type has a negative impact on the generated FP-Tree. For sparse datasets, this structure generates a tree with a size equal to that of the dataset. In our proposal, the dataset type has any incidence on the generated FI-Tree. This main difference between these two approaches can be explained by the fact that our proposal is support independent.

C.8. Redundancy

The FP-Tree depends on the itemsets contained in a dataset. Hence, one frequent itemset is represented by a node in the FP-Tree structure. The nodes of the FP-Tree are connected between them by paths, and each path corresponds to a transaction in the dataset. Because the transactions contain redundant items, the same items can be founded in different transactions. Hence, a same item will be represented by the same node in different paths of the FP-Tree, thus generating redundancy. The immediate consequence of this redundancy is that the size of the FP-Tree grows. On the other hand, our tree structure does not contain redundancy, because it emphasis on transactions and not on items.

C.9. Updating dataset

One important problem in a data mining process concerns the content of a dataset (static or dynamic content). In a static scenario, the tree representing a dataset is also static. But transaction datasets are updated regularly in real world (updating items and/or transactions). Here, the question is: *Does this change require rebuilding the tree or not?* In most cases, FP-Growth algorithm must rescan the updated dataset and rebuild FP-Tree, due to the change in the support count of frequent 1-itemsets. This process generated a lot of overhead that is not negligible. To avoid this important overhead, some research works have proposed to modify the generating process of the FP-Tree [23] [13] [3]. In our proposal, the FI-Tree is not rebuilding.

C.10. Practical example

To highlight the criteria discussed above between FP-Tree and FI-Tree, we illustrate the main differences between these two trees taking as an example the transaction dataset of Table II and its FP-Tree representation (see Figure 1), conditional FP-Tree (see Table III) and FI-Tree (see Figure 3).

TABLE XI
QUANTITATIVE COMPARISON BETWEEN FP-TREE AND FI-TREE

Criteria	FP-Tree	FI-Tree
#Scans	02	1
#Trees	05	1
#Nodes	18	6
#Leaves	?	7

The obtained results (see Table XI) show clearly that FI-Tree has many advantages relatively to FP-Tree.

V. EXPERIMENTAL RESULTS

In this section, we study the practicability of our proposal for finding frequent itemsets. We performed different experiments to find frequent itemsets using our proposed approach (with signatures), comparatively to Apriori algorithm that uses candidate itemsets.

All experiments are conducted on an Intel(R) Core(TM) i5-3470T CPU @ 2.90GHz x 4 with 8 GB of RAM under the JDK8 on a Fedora 27 system.

A. Datasets

For the experiments, we used real and synthetic datasets. The datasets were produced by the library “arules” [18] of R-cran Software. The used datasets have different characteristics (see Table XII).

TABLE XII
DATASET CHARACTERISTICS

Dataset name	Type	#Transactions	#Items	Average transaction size	Database size (MB)
Mushroom	Dense	8124	119	23	0,565
Retail	Sparse	88162	16469	11	4,156
Accidents	Sparse	340184	468	34	33,900
T10I4D100K	Sparse	100000	870	10	3,930
T40I10D100K	Sparse	200000	942	40	14,800
T30D50K1K	Sparse	50000	1000	14	2,730
T30D100K1K	Sparse	100000	1000	14	5,460
T30D150K1K	Sparse	150000	1000	14	8,180

B. Dataset Results and Discussions

B.1. FI-Mine vs Apriori

The different results (execution time) obtained with our proposed algorithm (FI-Mine) are compared to those of Apriori. These results are synthesized in Figures 5 to 10.

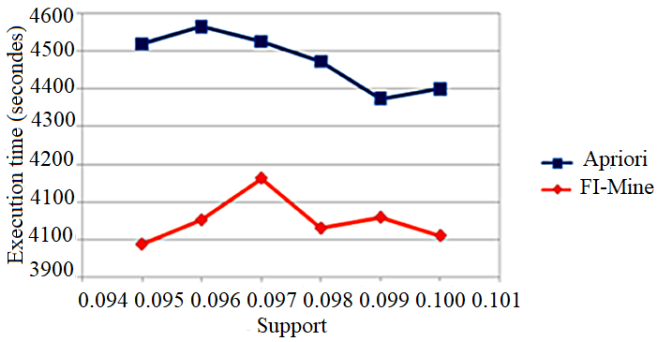

Fig. 5. Results for Mushroom Dataset

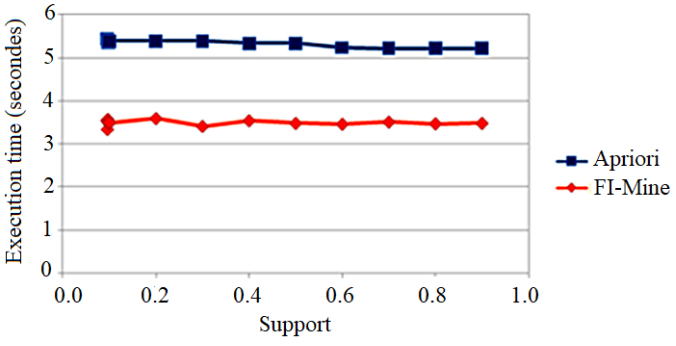

Fig. 6. Results for Retail Dataset

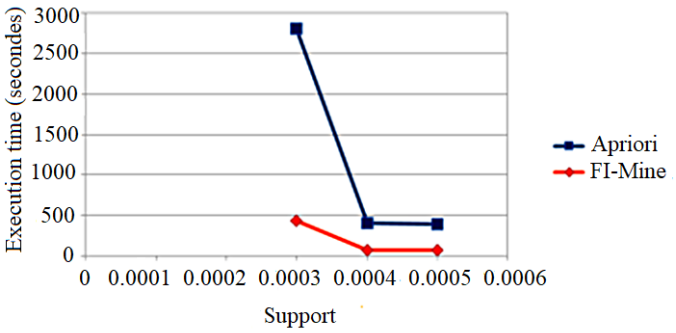

Fig. 7. Results for 50K Transactions Dataset

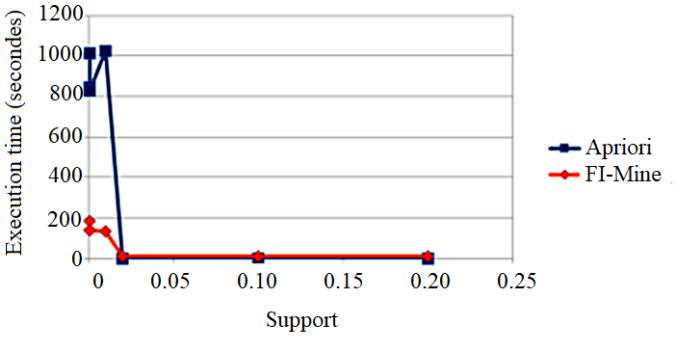

Fig. 8. Results for 100K Transactions Dataset

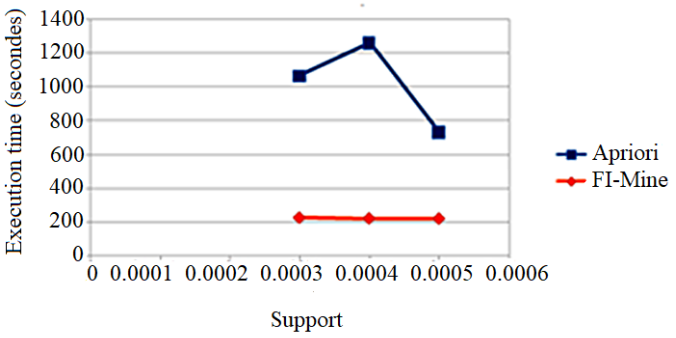

Fig. 9. Results for 150K Transactions Dataset

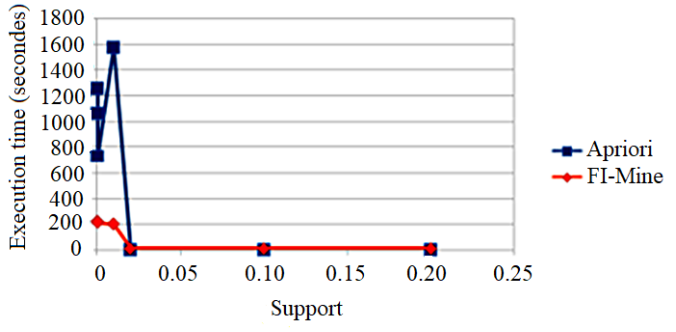

Fig. 10. Results for 150K Transactions Dataset

From the above figures, we remark that results of our proposed algorithm (FI-Mine) outperform those of Apriori one.

For example, for the case of 50K transactions dataset and Minsup equals to 0.0003, we have a gain of 2380.57 sec. comparatively to Apriori algorithm (2812.01 sec. for Apriori algorithm and 431.44 sec. for FI-Mine). The same remark is valid for the other figures.

B.2. FI-Mine vs FP-Growth

In this section, we use the algorithms FI-Mine, based on FI-Tree, and FP-Growth, based on FP-Tree, to generate frequent itemsets. We consider several transaction datasets.

The parameter which is considered in the analysis is the space complexity. We compare the results of FI-Tree with those of FP-tree. Our observed results are listed below.

TABLE XIII
FI-TREE RESULTS

Datasets	#Nodes	Space memory for FI-Tree (MB)
Chess	6391	0.08
Mushroom	16247	0.19
Accidents	164771	1.98
T40I10D100K	178269	2.14
Retail	199861	2.40

We note that the memory required by our structure FI-Tree is independent of any parameter, specially support threshold. Each dataset needs a fixed size memory.

The size of the FP-Tree depends on the minimum support specified by the user. The size of the FP-Tree is inversely proportional to the support value.

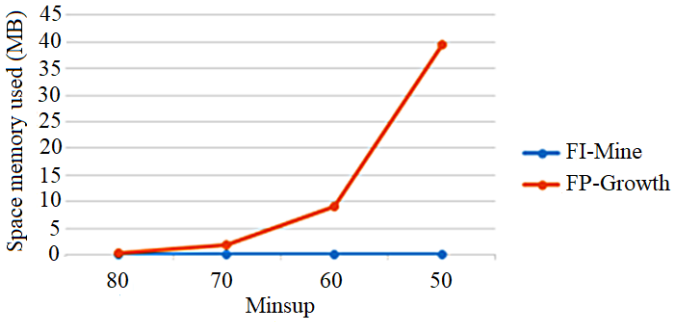

Fig. 11. FI-Mine vs FP-Growth for Chess Dataset

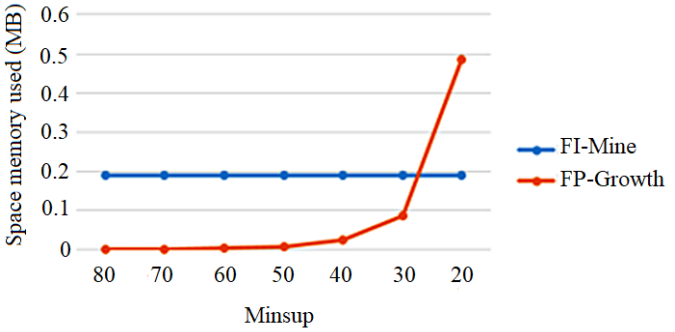

Fig. 12. FI-Mine vs FP-Growth for Mushroom Dataset

B.3. Discussion

Figures 11 to 16 highlight the space memory required by each one of the two trees (FI-Tree and FP-Tree). To analyze more precisely the difference in space memory between these structures, we calculated a gain factor. This gain is the

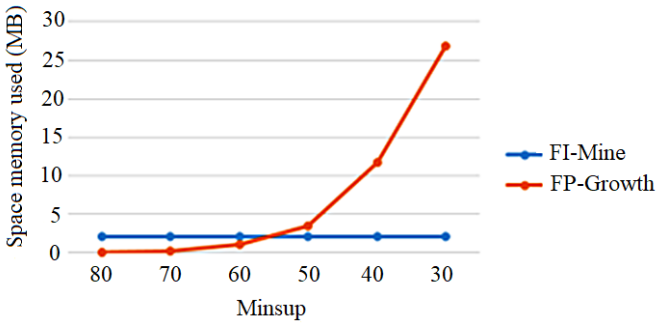

Fig. 13. FI-Mine vs FP-Growth for Accident Dataset

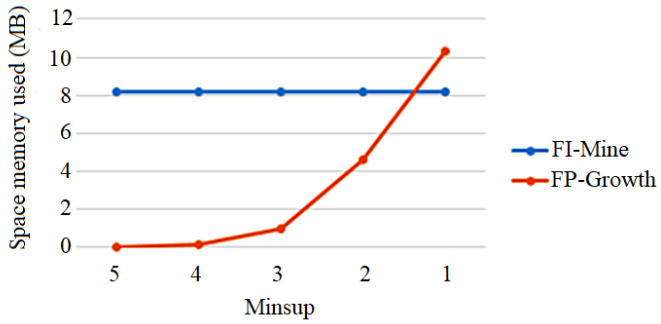

Fig. 14. FI-Mine vs FP-Growth for T10I4D100K Dataset

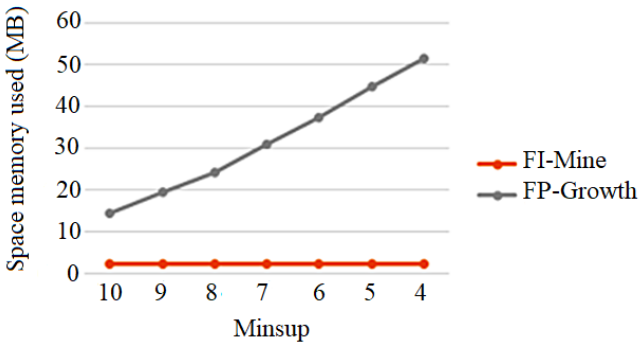

Fig. 15. FI-Mine vs FP-Growth for T40I10D100K Dataset

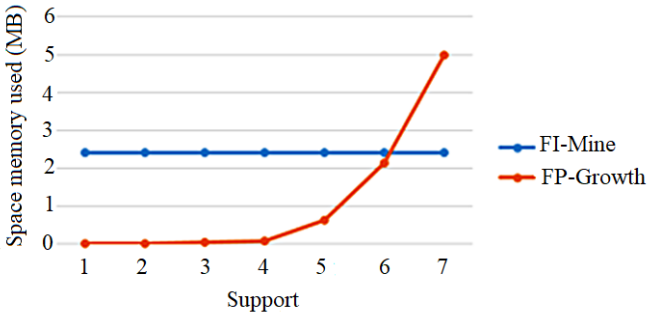

Fig. 16. FI-Mine vs FP-Growth for Retail Dataset

difference between the space memory required for FP-Growth and that required for FI-Mine. We distinguish several cases depending on the used dataset:

- Chess dataset (Figure 11): FI-Mine outperforms the FP-rowth for all minsup values (50 to 80%). We remark also that whatever the values of the minsup, the space memory for FI-Tree remains constant (minsup independent), while for FP-Tree the more the value of the minsup decreases, the more the memory space increases. For example, between 60 and 50% (minus 10%), the used space memory for FP-Growth increases five times.
- Mushroom dataset (Figure 12): For high minsup (30 to 80%), the difference between the two trees is not significant, but for the minsup 20%, the space memory required by FP-Growth is almost 3 times than that of FI-Mine. We notice also that this dataset is dense.
- Accidents dataset (Figure 13): For this dataset that is sparse, we note that the more the value of the support decreases, the more the memory space increases until reaching a multiplying factor of 2 between 40 and 30%. If we analyze the memory space needed for FI-Mine and FP-Growth, we notice that FP-Growth requires a larger memory space than that of FI-Mine up to a factor of 13.
- T10I4D100K (Figure 14): For this sparse synthetic dataset, we use small values for the minsup (from 5 to 1%). First, we notice that the size of the FP-Tree gradually increases until it exceeds that of FI-Tree (minsup = 1). Despite this, FP-Growth remains better than FI-Mine in terms of memory space.
- T40I10D100K dataset (Figure 15): This dataset is also synthetic and sparse, for which FP-Tree greatly exceeds FI-Tree. Each time the minsup decreases (from 10 to 4), the space memory required by FP-Growth is very much greater than that of FI-Mine (ranging from a factor of 7 for a support value equal to 10 up to a factor of 25 for a support value equal to 4).
- Retail dataset (Fig. 16): In this dataset, we use small values for the minsup (from 5 to 0.25%). We note that the used space memory for FP-Growth increases each time the minsup values decrease with different factors. However, the used space memory of FI-Mine remains constant.

As important remark, we notice that the used space memory for FI-Tree always remains constant and does not depend on the variation of the minsup values (small or high values). Like the performance of FP-Growth depends on many factors: dataset type, dataset size, minsup values, etc.

V. Conclusion and Future Works

In this paper, we reviewed the various data structures, specifically trees, using in data mining algorithms. We summarize the main features of some relevant tree structures, their tree representation scheme, and the way they build the tree. Based on this review, we proposed a new tree structure to represent a transaction dataset. Our proposal structure is compared to the well-known FP-Tree. This comparison is made between the space memory used by FP-Tree and FI-Tree. Finally, we have compared experimentally our proposal algorithm with Apriori one, and we have showed that our signature-based structure can enhance the time for finding frequent itemsets. As a future work, we plan to experiment our proposal on different datasets with different parameters (dense and sparse datasets, low and high supports, static and dynamic datasets). Future work also includes a parallel implementation

of our approach to test its performance and scalability with large datasets.

REFERENCES

[1] R. Agrawal, T. Imieliński, A. Swami.: "Mining association rules between sets of items in large databases", Proceedings of the ACM SIGMOD. International Conference on Management of Data, New York, NY, USA, 1993, pp. 207-216. https://dl.acm.org/doi/10.1145/170036.170072

[2] R. Agrawal, R. Srikant: "Fast Algorithm for Mining Association Rules", Proceeding of the 20th VLDB Conference, Santiago. September 12-15. Chile, 1994, pp. 487-499. https://doi.org/10.1007/BF02948845

[3] S. Ahmed, B. Nath: "Modified FP-Growth: An Efficient Frequent Pattern Mining Approach from FP-Tree", In Proceeding of International Conference on Pattern Recognition and Machine Intelligence, December 17-20,Tezpur, India, 2019, pp 47-55. https://doi.org/10.1007/978-3-030-34869-4_6

[4] T. Bao, T. Tuan: "Query Optimization in Object Oriented Databases Based on Signature File Hierarchy and SD-Tree", EAI Endorsed Transactions on Context-aware Systems and Applications, Vol. 3, Issue 8, 2016, pp 1-7. http://dx.doi.org/10.4108/eai.9-3-2016.151114

[5] M.E.H. Benelhadj, K. Arour, M. Boufaïda, Y. Slimani: "Mining Frequent Itemsets with Tree Signatures", In Proceeding of European Conference on Data Mining, Freiburg Germany. July 28-30, 2010, pp 163-166. http://www.iadisportal.org/digital-library/mining-frequent-itemsets-with-tree-signatures

[6] F. Bodon: "A Trie-based APRIORI Implementation for Mining Frequent Item sequences", Proceeding of the 1st International Workshop on Open Source Data mining. August 21, Chicago, Illinois, USA, 2005, pp. 56-65. https://doi.org/10.1145/1133905.1133913

[7] F. Bodon, L. Rónyai: "Trie: An Alternative Data Structure Data Mining Algorithms", Mathematical and Computer Modelling, Vol. 8, Issues 7–9, 2003, pp 739-751. https://doi.org/10.1016/0895-7177(03)90058-6

[8] C.H. Chee, J. Jaafar, I. Abdul Aziz, M.H. Hasan, W. Yeoh: "Algorithms for frequent itemset mining: a literature review", Artificial Intelligence Review, An International Science and Engineering Journal, Vol. 52, Issue 4, 2019, pp 2603–262. http://doi.org/10.1007/s10462-018-9629-z

[9] Y. Chen: "Signature Files and Signature Trees", Information Processing Letters 82, Elsevier, Vol. 82, Issue 4, 2002, pp 213-221. https://doi.org/10.1016/S0020-0190(01)00266-6

[10] Yangjun Chen, Yibin Chen: "Signature File Hierarchies and Signature Graphs: a New Index Method for Object-Oriented Databases", Proceeding Of ACM Symposium on Applied Computing, March 14-17, Nicosia, Cyprus, 2004, pp 724-728. https://doi.org/10.1145/967900.968050

[11] Y. Chen: "On the Signature Trees and Balanced Signature Trees", In proceeding of the 21st International Conference on Data Engineering (ICDE'05), April 5-8. Tokyo, Japan, 2005, pp. 742-753. https://doi.org/10.1109/ICDE.2005.99

[12] Yangjun Chen, Yibin Chen: "On the Signature Tree Construction and Analysis", IEEE Transactions on Knowledge and Data Engineering, Vol. 18, Issue 9, 2006, pp. 1207-1224. https://doi.org/10.1109/TKDE.2006.146

[13] W. Cheung: "Incremental mining of frequent patterns without candidate generation or support constraint", Seventh International Database Engineering and Applications Symposium, 2003. Proceedings, Jul 18-18, Hong Kong, China, 2003, pp. 111-116. https://doi.org/10.1109/IDEAS.2003.1214917

[14] D. Comer: "The Ubiquitos B-Tree", Computing Surveys, Vol. 11, Issue 2, 1979, pp 121-137. https://doi.org/10.1145/356770.356776

[15] C. Faloutsos: "Access methods for text", ACM Computer Survey, Vol. 17, Issue 1, 1985, pp. 49-74. https://doi.org/10.1145/4078.4080

[16] C. Faloutsos: "Signature Files: Design and Performance Comparaison of Some Signature Extraction Methods", ACM Sigmod Record, Vol. 14, Issue 4, 1985, pp. 63-82. https://dl.acm.org/doi/10.1145/971699.318903

[17] C. Faloutsos, R. Chan: "Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and Performance Comparison", Proceeding of VLDB, 1988, pp 280-293. https://doi.org/10.1184/R1/6605615.v1

[18] http://fimi.uantwerpen.be/data/. Repository is the result of the workshops on Frequent Itemset Mining Implementations. Last access July 2021.

[19] M. Hahsler, B. Grun, K. Hornik, C. Buchta: "Introduction to arules: A computational environment for mining association rules and frequent itemsets", Journal of Statistical Software, Vol. 14, Issue 15, 2016, pp 1-25. https://doi.org/10.18637/jss.v014.i15

[20] J. Han, J. Pei, Y. Yin: "Mining frequent patterns without candidate generation", ACM SIGMOD Record, Vol. 29, Issue 2, 2000, pp. 1-12. https://doi.org/10.1145/335191.335372

[21] M. Ikhlef: "A Quantum Swarm Evolutionary Algorithm for Mining Association Rules in Large Databases", Journal of King Saud University - Computer and Information Sciences Vol. 23, Issue 1, 2011, pp 1-6. https://doi.org/10.1016/j.jksuci.2010.03.001

[22] D.L. Lee, Y.M. Kim, G. Patel: "Efficient Signature File Methods for Text Retrieval", IEEE Transactions on Knowledge and Data Engineering, Vol. 7, Issue 3, 1995, pp 423-435. https://doi.org/10.1109/69.390248

[23] B. Rácz: Nonordfp: "An FP-Growth Variation without Rebuilding the FP-Tree", Proceedings of the {IEEE} {ICDM} Workshop on Frequent Itemset Mining Implementations, Brighton, UK, November 1, 2004. https://dblp.org/rec/conf/fimi/Racz04.html

[24] C.S. Roberts: "Partial-Match Retrieval via the Method of Superimposed Codes", Proceeding of the IEEE, Vol. 67, Issue 12, 1979, pp 1624-1642. https://doi.org/10.1109/PROC.1979.11543

[25] E. Shanthi, R. Nadarajan: "Applying SD-Tree for Object-Oriented Query Processing", Journal of Informatica, Vol. 33, Issue 2, 2009, pp 177-187. https://www.informatica.si/index.php/informatica/article/view/235/232

[26] W. Wang, L. Tang, J. Han, J. Liu: "Top down fp-growth for association rule mining", 6th Pacific-Asia Conference Proceeding, Taipei, Taiwan, May 6–8, 2002, pp 334-340. https://link.springer.com/chapter/10.1007/3-540-47887-6_34

[27] M. Yarlagadda K. Gangadhara Rao, A. Srikrishna: "Frequent itemset-based feature selection and Rider Moth Search Algorithm for document clustering", Journal of King Saud University – Computer and Information Sciences, Vol. 34, Issue 4, 2022, pp 1098-1109. https://doi.org/10.1016/j.jksuci.2019.09.002

Mohamed El Hadi Benelhadj got his Engineering and Magister degree in Computer Science from Mentouri University, Constantine, Algeria. He obtained his PhD in computer science from the same University. Currently, he is an Assistant Professor at the Tamanrasset University Algeria. He is a Research Associate in the Research group "Information Systems and Knowledge Bases", Mentouri University and in the Joint group for Artificial Reasoning and Information Retrieval (https://jarir.tn/), University of Manouba Tunis Tunisia. His research interests include Datamining, KDD and Parallel Computing.

Mohamed Mahmoud Ould Deye is an Assistant Professor at Cheikh Anta Diop University of Dakar, Senegal. He is also a member of the Joint group for Artificial Reasoning and Information Retrieval (https://jarir.tn/), University of Manouba Tunis Tunisia. His areas of interest are Cloud Computing, Web Services and performance evaluation of distributed systems.

Yahya Slimani is a Emerite Professor at the Higher Institute of Multimedia Art of Manouba (ISAMM), University of Manouba, Tunisia (https://www.yahya-slimani.net/). He is a member of the LISI research Laboratory, Carthage University, Tunisia. He obtained his Doctor-Engineer in computer science from the University of Lille, France, in 1986. He presented a PhD thesis entitled "A Relational Logic for Parallel Programming" and obtained his doctoral diploma in computer science from the University of Es-Senia, Algeria, in 1994. His main research interests include Cloud Computing, Load Balancing, Grid and Cloud Computing, Information Retrieval, Parallel and Distributed Computing, High Performance Computing, Data Science, Data Mining and Machine Learning. It is the head of the research group JARIR (https://jarir.tn/).