A New Compact Structure to Extract Frequent Itemsets

Mohamed El Hadi Benelhadj[1], Khedija Arour[2], Mahmoud Boufaïda[1] and Yahya Slimani[3]

[1] *LIRE Laboratory, Computer Science Dpt, Mentouri University Constantine, Algeria*
[2] *National Institute of Applied Science and Technology Tunis, Tunisia*
[3] *Computer Science Department, Faculty of Sciences Tunis, Tunisia*

me.benelhadj@umc.edu.dz, mboufaida@umc.edu.dz, khedija.arour@issatm.rnu.tn, yahia.slimani@fst.rnu.tn

Abstract

Discovery of association rules is an important problem in KDD process. In this paper we propose a new algorithm for fast frequent itemset mining, which scan the transaction database only once. All the frequent itemsets can be efficiently extracted in a single database pass. To attempt this objective, we define a new compact data structure, called ST-Tree (Signature Transaction Tree), and a new mining algorithm ST-Mine to extract frequent itemsets.

Keywords: *Data mining, Frequent itemset, Association rule, Binary structure, Signature file, Signature tree*

1. Introduction

Originally introduced by Agrawal [1] in the context of transactional databases, the association rule mining approach is now used extensively to find associations in biological databases, web log data, telecommunications data, census data and many other types of databases.

Though several algorithms have been developed for fast mining of frequent itemsets over the years [3, 4, 11, 14, 15]. 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 [1, 3, 4] and the second one is based only on the pattern fragment growth like the *FP-growth* or frequent itemset-growth algorithm [11].

The "generate and test approach" is based on an anti-monotone Apriori property [2]: 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 itemset candidates on length $(k + 1)$ from the set of frequent itemsets of k $(k >= 1)$, and continuously checks their corresponding occurrence frequencies in the database. Though this algorithm works relatively well in smaller database. However, when there exist a large number of frequent patterns and/or long patterns, the "generate and test approach" may still suffer from generating huge numbers of candidates and needs many scans of large databases for frequency checking.

The pattern-growth approach, such as *FP-growth* also uses the *Apriori* property. It works by recursively partition the database into sub-databases 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 database into a specialized structure called *FP-tree* and pursuing partition-based mining recursively. Nevertheless, this

algorithm may still encountered difficulties in large sparse databases when the *FP-tree* will be very large [11].

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 databases with varying degrees of success. But, generally, the problem of repeatedly scanning the databases remains.

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

Signatures are hash coded abstractions of each item. It is a binary pattern of predefined length with fixed number of 1's. The items' signatures are superimposed to form transactions' signatures. Say S_I a signature of candidate itemset, S_I is generated using the same hash function. To find the signature S_I in the *ST-Tree*, we select all transaction signatures S_T, such that $S_T \wedge S_I = S_I$, S_I is called a drop and $\wedge$ is the superimposed operator. Many unqualified transaction signatures are immediately rejected. This method guarantees that all the qualifying transactions' signatures will be selected but some non-qualifying transactions' signatures may also pass the signature test. All signatures are used to compute the itemset support. This support is called the maximum support (denoted *Maxsup*) and is used in the extraction process of frequent itemsets. If the associated value is less than a specified user threshold, the itemset *I* is said to be frequent. The transactions, that actually contain the item *I*, are called *actual drops* and the others called *false drops*. The number of *false drops* can be statistically controlled by careful design of the signature extraction method [9] and by using a reasonable length of signatures [10].

The reminder of this paper is organized as follows: In Section 2, we survey a state of art and gives an overview of the concept of tree signature. Section 3 presents our proposed structure called *ST-Tree*, the tree construction process, the search process of a signature in *ST-Tree*, the process of generating frequent itemsets. In this same section, we analyze the theoretical complexity of our proposition. In Section 4, we present and discuss some experimental results of our proposition. Finally, Section 5 concludes and skittles futures avenues of following works.

2. State of the Art

Indexing plays a fundamental role in the fast recovery of required data from large databases. Index techniques have been extensively investigated in both the information retrieval and database research areas and many methods have been developed within the past three decades [5, 8, 9, 10, 12, 13, 19]. Among the indexing techniques, the signature file approach is extensively used for its efficient evaluation of set-oriented queries and for its easy handling of insert and update operations. Different approaches have been discussed by researchers to represent Signature File in a way conductive for evaluating queries, such as Sequential Signature File [7], Bit-Slice Signature file [7], Compressed Bit Sliced Signature File [19], Multilevel Signature file [17], S-Tree and its variants [12], Signature Graph [6], Signature tree [5, 7, 13] and SD-tree [17].

2.1. Specification of Signature File

A Signature is a bit string formed from a given value. Compared to other index structures, signature file is more efficient in handling new insertions and queries on parts of words. Other advantages include its simple implementation and its ability to support large files. But it introduces information loss which can be minimized by carefully selecting the signature extraction method.

Definition 1: A signature is a binary vector of length m obtained by applying one (or several) hash function(s) [7].

Several techniques for signature extraction such as Word Signature (*WS*) [8, 10], Superimposed Coding (*SC*) [8, 10, 16], Multilevel Superimposed Coding (*MSC*) [18], Run Length Encoding (*RL*) [8, 10], Bit-block Compression (*BC*) [8, 10] and Variable Bit-block Compression (*VBC*) [10] have been developed.

The signature of a text block can be obtained by superimposing all its constituent word signatures using "*logical OR operation*". The set of all signatures forms a signature file. An example of Superimposed Coding is given in Table 1.

Table 1. Superimposed Coding Example

Multilevel Superimposed Coding	0000 0000 0000 0010 0000 0000 0001 0000 0000 0000 0000 1000 0000 0000 0000
Block Signature	0000 1001 0000 0010 0000

An example of sample query evaluation is given below.

Example

Sample queries

Matching query

Word = "Multilevel"

Signature = 0000 0001 0000 0000 0000

Block signature → Actual drop

False Match query

Word = "Information"

Signature = 0000 1000 0000 0000 0000

Block signature → False drop

Non-matching query

Word = "Compression"

Signature = 1000 0000 0000 0000 0000

Block signature → not match

2.2. Signature File Representation

This section presents briefly the various techniques used to represent signature files.

2.2.1. Sequential Signature File (SSF): SSF is the simplest organization, which is easy to implement and requires low storage space and low update cost. The signatures are stored sequentially in the signature file. When a query is given, a full scan of the signature file is required [19]. Therefore, it is generally slow in retrieval.

2.2.2. Bit sliced signature file (BSSF): BSSF stores signatures in a column-wise manner. Thus, F files (called bit-slice files), one for each bit position of the set signatures, are used. In retrieval, only a part of the F bit-slice files have to be scanned, so that the search cost is lower than that of SSF. However, update cost becomes larger, because an insertion of a new signature requires about F disk accesses, one for each bit-slice file [19].

2.2.3. Compressed Bit Sliced Signature File (CBS): By choosing a suitable hashing function for signature extraction, the number of 1's is forced to be one. Here, the signature length should be increased to maintain the false drop probability at minimum. This creates a sparse matrix which is easy to compress [19]. A simple way to compress this matrix is to replace each 1 with its corresponding physical address.

The hash table has a list of pointers pointing to the heads of linked list [19]. For example, assume that the word *Text* has its first bit set to 1 and it appears at the 50th byte of text file then searching the first bucket list, we find the position of the word *Text*. Although this approach gives some space saving, the number of false drops will definitely be increased due to sparse signature files.

2.2.4. S-Tree: S-Tree is a B+ tree like structure [18] with leaf nodes containing a set of signatures with their Object Identifiers (OIDs). The internal nodes are formed by superimposing the lower level nodes.

The advantage is that the simple tree searching way of obtaining signatures rather than searching the whole signature file.

The disadvantage is that due to superimposing, internal nodes in the upper level tend to have more weight which ultimately decreases selectivity. The S-tree has been further improved in [17], where a number of new split methods such as linear split, Quadratic split, Cubic split and hierarchical clustering for S-tree are proposed to improve query response time. A new hybrid scheme combining linear hashing, Stree and parametric weighted filter is used to evaluate subset-superset queries.

2.2.5. Multi-level Signature File: This structure is similar to S-Tree. However a signature at non-leaf node is formed by superimposed coding from all text blocks indexed by the subtree of which the signature is the root. Though this method improves selectivity in an internal node, it requires more space. An improved method for multi-level signature file is discussed in [17].

2.2.6. Signature Graph: The signature file is organized as a trie like structure. However, the path visited in the graph to find a signature that matches a given query signature corresponds to a signature identifier which is not a continuous piece of bits, differentiating the signature graph from trie [6].

Though signatures are represented compactly, thesearch path length is not the same for all queries. In other words the graph is not balanced. In worst case it degrades to a signature file.

2.2.7. Signature Tree: A tree of signatures *Ts* represents a set of signatures $S = \{S_1,...,S_n\}$ where $S_i \neq S_j$ for all $i \neq j$ and $|S_k| = m$, for $1 \leq k \leq n$. *Ts* is a binary tree such that:

- For each internal node of Ts, the left edge leaving it is always labeled with "0" and the right edge is always labeled with "1".
- Ts have n leaves labeled ln_1, $ln_2,...,ln_m$, used as pointers to m different signatures $S_1,...,S_m$ in *S*. Let ln be a leaf node. Denote the pointer *p(nf)* to the corresponding signature
- Each internal node v is associated to a positive number, noted by *Pos(v)*, to tell which bit will be checked.

Each signature is identified from the root by checking the bit positions dictated by the nodes. Nevertheless for a query signature, the tree is searched top to bottom according to the bit positions dictated by the nodes rather than the 1s in query signature. Also, for a match with bit equal to 1, searching follows the right sub-tree and for 0 at a node both left and right sub-trees are followed [5, 6, 7].

2.2.8. Signature Declustering Tree (SD-tree): The *SD-tree* is composed by three types of nodes: Internal nodes, Leaf nodes, Signature nodes.

The internal nodes and leaf nodes are somewhat similar to the internal nodes and leaf nodes of B+ trees respectively. The internal nodes form the upper tree and leaf nodes at last but one level. The signature nodes are at the bottom level of the *SD-tree* [17].

3. ST-Tree Structure

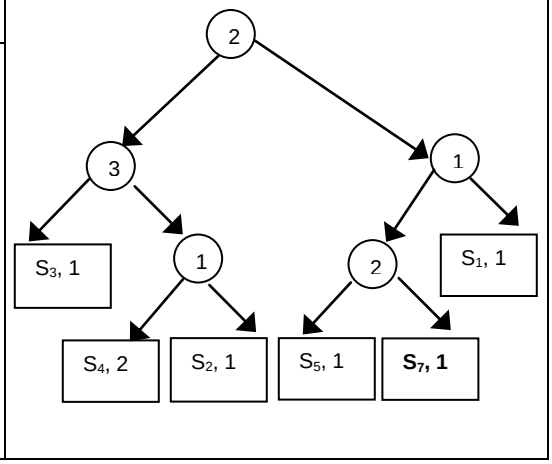
Improving performance of discovering association rules requires an optimization of the extraction phase of frequent itemsets. To reach this objective, we propose to use the *ST-Tree* structure representing the set of transaction signatures. Each transaction is represented by a signature of size *m*. *ST-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 two informations: a signature *S* and the transactions number generating *S*. The number of leaf nodes in a *ST-Tree* is equal to the number of signatures.

The construction of a signature tree requires two phases:

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

Table 2. Example of Tids, Transactions, Transactions signatures, ST-Tree

Tids	Transactions	Signatures	
T_1	1,5,6,8	S_1	11000110
T_2	2,4,8	S_2	10101000
T_3	5,8	S_3	10000100
T_4	2,3	S_4	00110000
T_5	4,5,7,10	S_5	01001101
T_6	3,10	S_6	00110000
T_7	3,6,7,9	S_7	01010011



We note, in example of Table 2, that the transactions T_4 and T_6 generate the same signature (the collision phenomenon). It will be represented only once (S_4 in our example).

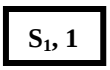
3.1 A Simple Way for Constructing ST-Tree

Below, we give an algorithm to construct *ST-Tree*. At the beginning, the tree contains an initial node containing the first transaction signature S_1 and 1 (the number of transactions generating the first signature). The following is the formal description of the algorithm *ST-Tree-construction*.

3.1.1 Steps to generate ST-Tree: The following steps (a) to (f) present the *ST-Tree* creation process.

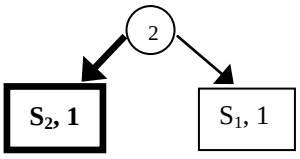
The step (a) build a root node *r* such that *r* is a leaf node and contains the signature S_1, the number "1" and the identifier of the first transaction T_1.

(a) Insert (S_1: 11000110)

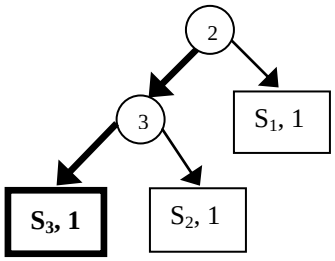


The steps (b) to (g) insert a new signature S_i to the corresponding leaf node in *ST-Tree*, using the value of signature bit position in each internal node.

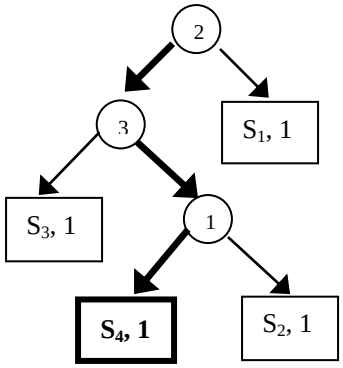
(b) Insert (S_2: 10101000): the first different bit between S_1 and S_2 is the second bit, $S_1[2] = 1 \neq S_2[2] = 0 \Rightarrow$ create internal node v with pos(v) = 2 and leaf node {S_2, 1}.



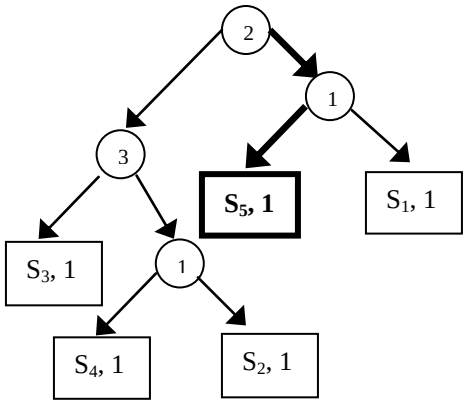
(c) Insert (S_3: 10000100): $S_3[2] = 0$, the 1st different bit between S_3 and S_2 is the 3rd bit, $S_2[3] = 1 \neq S_3[3] = 0 \Rightarrow$ create internal node v with pos(v) = 3 and leaf node {S_3, 1}.



(d) Insert (S_4: 00110000): $S_4[2] = 0 \neq S_4[3] = 1$, the first different bit between S_4 and S_2 is the 1st one, $S_4[1] = 0$ and $S_2[1] = 1 \Rightarrow$ create internal node v with pos(v) = 1 and leaf node {S_4, 1}.

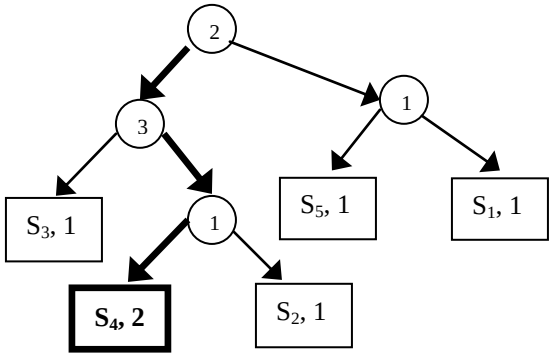


(e) Insert (S_5: 01001101): $S_5[2] = 1$. The first different bit between S_1 and S_5 is the 1st bit, $S_1[1] = 1 \neq S_5[1] = 0 \Rightarrow$ create internal node v with pos(v) = 1 and leaf node {S_5, 1}.

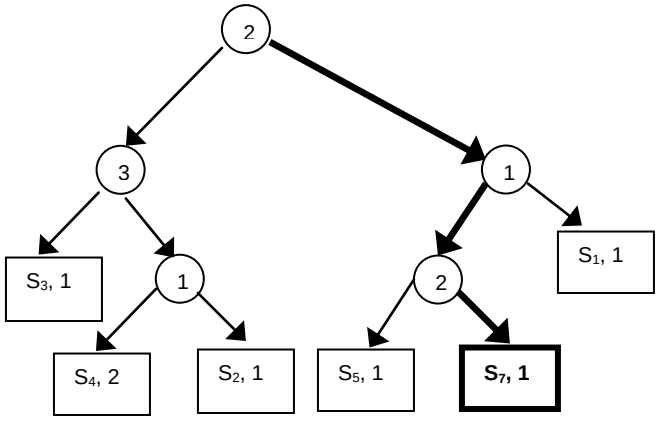


(f) The step (f) inserts an existing signature.

Insert (S_6: 00110000): $S_6[2] = 0$, $S_6[3] = 1$, $S_6[1] = 0$. $S_6 = S_4 \Rightarrow$ Increment the transaction number in the leaf node.



(g) Insert (S_7: 01010011), $S_7[2] = 1$, $S_7[1] = 0$. The 1st different bit between S_5 and S_7 is the 4th bit, $S_5[4] = 0 \neq S_7[4] = 1 \Rightarrow$ create internal node v with position(v) = 4 and leaf node {S_7, 1}.



3.1.2. Algorithm to construct ST-Tree: Below, we present the construction algorithm of *ST-Tree*. The following is the formal description of the algorithm *ST-Tree-construction*.

Algorithm *ST-Tree-construction*
Input: Transactions
Output: ST-Tree
Begin
  $S_1 \leftarrow$ Gen_Signature (T_1)
  Construct a ST-Tree with only the root node r. /* r = {S_1, 1} */
  For i = 2 à n Do
    $S_i \leftarrow$ Gen_Signature (T_i)
    Call Insert (S_i)
  EndDo
End

At the beginning, the tree contains an initial node with the first transaction signature *S_1* and *1* (the number of transactions generating the first signature).

Then, we compute a next transaction signature S_i and insert it into *ST-Tree*. We traverse the tree from the root. Let v an encountered internal node with *position(v) = p*. Then, $S_i[p]$ will be checked. If $S_i[p] = 0$, we go left; otherwise, we go right. If *v* is a leaf node, we compare S_i with the signature *S* in *v*. If we have two identical signatures, we increment the corresponding transaction's number. If not, several bits of S_i agree with *S*. Assume that the first *k* bits of S_i agree with *S*. But, S_i differs from *S* in the $(k+1)^{th}$ position. We construct a new internal node *u* with *position(u) = k+1* and replace *v* with *u*, but *v* will not be removed. By 'replace', we mean that the position of *v* in the tree is occupied by *u* and *v* becomes one of *u's* children. We construct also a new leaf node v_i containing $\{S_i, 1\}$. If $S_i[k+1] = 1$, we make *v* be the left and v_i right child of *u*, respectively. If $S_i[k+1] = 0$, we make *v* the right child of *u* and v_i the left child of *u*. The formal description of the algorithm Insert (S_i) is given bellow.

Algorithm *Insert*
Input: The signature S_i
Output: ST-Tree
Begin
  Stack ← r
  While Stack not empty Do
    v ← Pop (Stack)
    If v is an internal node Then
      j ← position (v)
      If $S_i[j] = 1$ Then
        Push (Stack, right_child)
      Else
        Push (Stack, left_child)
      Endif
    Else
      If $S_i = S$ Then
        nt ← nt + 1
      Else
        /* Assume that the first k bits of S_i agree with S and
          S_i differs from S in the $(k+1)^{th}$ position.
          Generate a new internal node u with position (u) = k+1.
          Generate a new leaf node $v_i = \{S_i, 1\}$ */
        If $S_i [k+1] = 1$ Then
          v_i will be the right child of u and v its left child
        Else
          v_i will be the left child of u and v its right child
        Endif
      Endif
    Endif
  EndDo
End

3.2 Searching in ST-Tree

Now, we discuss how to search a signature S_I of an itemset *I* in the *ST-Tree* structure. During the traversal of *ST-tree*, the inexact matching is done as follows:

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 and its signature $S_I = 10000100$. Then, only part of the *ST-Tree* will be searched (thick edges in Figure 2). On reaching a leaf node *v*, the signature *S* will be checked against S_I. In our example, we visits 3 signatures S_1, S_2 and S_3; but, we select only S_1 and S_3 because S_2 doesn't contains S_I.

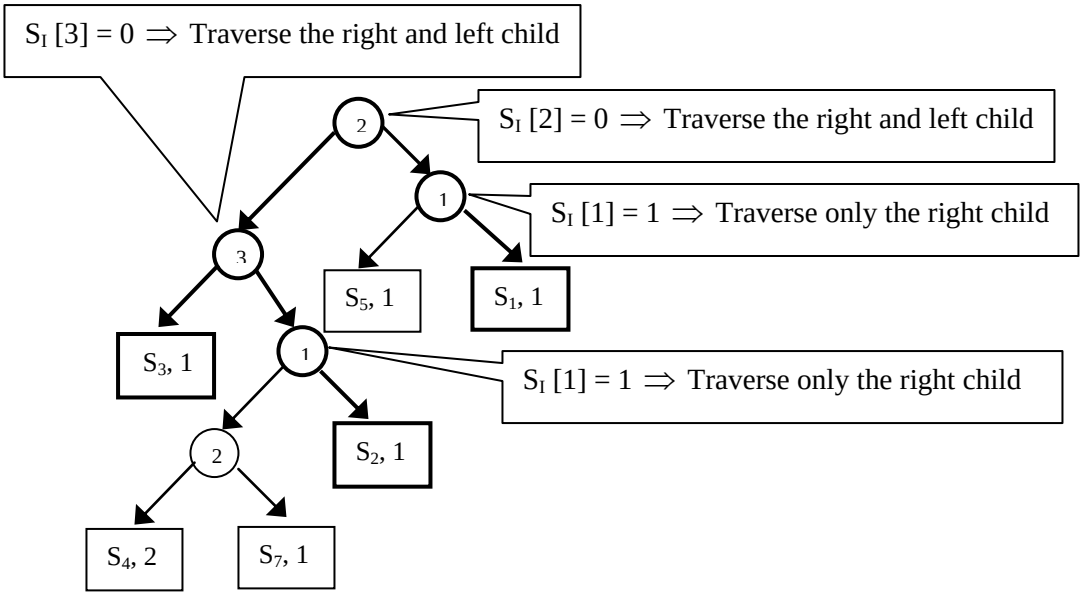


Figure 2. Signature Search Process

Finally, we visit 3 signatures: S_1, S_2 and S_3. S_1 and S_3 contain S_I but not S_2.
Bellow is the formal description of the search algorithm.

Algorithm *ST-Tree-search (I)*
Input: An itemset I
Output: The maximum support (*Maxsup*) of I
Begin
  S_I = Gen_Signature (I)
  Maxsup ← 0
  Push (Stack, root);
  While Stack not empty Do
    v ← Pop (Stack);
    If v is an internal node Then
      i ← position(v)
      If S_I [i] = 1 Then

```
            Push (Stack, right_child (v))
          Else
            Push (Stack, left_child (v))
            Push (Stack, right_child (v))
          Endif
        Else
          If S contains S_I Then
            Maxsup ← Maxsup + nt
          Endif
        Endif
    EndDo
    Return (Maxsup)
End
```

3.3 Discovering Frequent Itemsets

The generation of frequent itemsets computes, for each candidate itemset *I*, the maximum support of *I*, denoted *Maxsup(I)*, and compares it to a minimum support denoted *Minsup*, a threshold fixed by the user. An itemset *I* is said frequent if *Maxsup (I)* ≥ *Minsup*.
The following is the formal description of the extraction algorithm of frequent itemsets.

Algorithm *Extraction_FI*
Input: Frequent 1-Itemsets
Output: The set of frequent k-itemsets
Begin
/* Initially, FI = {Frequent 1-itemsets} */
k ← 2
1. Generate a candidate k-itemset I
ST-Tree-search (I, Maxsup)
If Maxsup(I) ≥ Minsup Then
FI ← FI ∪ {I}
Endif
2. k ← k+1
3. Repeat 1 and 2 until no candidates k-itemset
4. Return (FI)
End

Example 2. Consider the signature $S_I = 10000100$ of example 1 and $Minsup = 2$. The selected signatures is S_1 and S_3, then $Maxsup = nt_1 + nt_3 = 1 + 1 = 2 = Minsup$, where nt_1 and nt_3 are respectively the number of transactions generating S_1 and S_3. These informations are given from associated leaf nodes. We concludes that the itemset *I* is a frequent one.

Algorithm *ST-Mine*
Input: {Transactions}
Output: {Frequent Itemsets}
Begin
ST-Tree-construction

Extraction_FI
End

3.3 Complexity Study

The algorithm *ST-Tree-construction* to build the signatures tree 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:

$1 + 2 + ... + n = n (n +1) / 2 = (n^2 + n) / 2$

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

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

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: $O(p(n/2^l)) \simeq O(pn)$.

Finally, the complexity of *ST-Mine* is polynomial and equal to:

$\boldsymbol{O(nm) + O(n^2) + O(pn)}$.

4. Experimental Study

We have implemented our algorithm in C++ language. The computer was Intel Core 2 Duo CPU T5670 1,80 GHZ. The capacity of the hard disk is 120 GB and the amount of the main memory is 2 GB.

For the first experiment, we use *ST-Mine* to construct *ST-Tree* and we study the impact of the hash function choice on the rate of false drops.

The second experiment compares *ST-Mine* using *ST-Tree* and *FP-Growth*, using *FP-Tree*. The comparison criterion considered is the memory used.

Several transactions databases are considered [20]. The features of the used databases are shown to Table 3.

Table 3. Features of Transaction Databases

Name	Type	#Transactions	# Items	Average transaction size	Database Size
T10I4D100k	Sparse	100 000	870	10.10	3.93 Mb
T40I10D100K	Sparse	200 000	942	39.54	14.80 Mb
Mushroom	Dense	8124	119	23.00	0.565 Mb
Chess	Dense	3196	75	37.00	0.334 Mb
Retail	Sparse	88162	16469	10.30	4.156 Mb
Accident	Sparse	340 184	468	33.80	33.90 Mb

4.1. Impact of the Hash Function

The first experimentation shows the hash function impact of the signature weight and the number of signatures in a leaf node.

The results of this experimentation are summarized in Table 4 with the following notations:

I-Nbre: Number of distinct items in database

I-Avg: Average number of items per transaction

H-Fct: Hash function

Size-S: Signature size

W: Average weight (Number of 1)

Size-B: Transactions number in Database

A-T: Average Number of Transaction per Leaf Node

Table 4a. Different "Modulo" Hash Functions

Database	Type	I-Nbre	I-Avg	Size-S	A-W	Size-B	A-T	H-Fct
Chess	Dense	75	37	128	37	3196	1	Modulo 128
Mushroom	Dense	119	23	128	23	8124	1	
T10I4D100K	Sparse	870	10	128	9	100000	1	
T40I10D100K	Sparse	942	39	128	34	100000	1	
Retail	Sparse	16470	10	128	9	88162	1	
Accidents	Sparse	468	33	128	32	340183	1	
Chess	Dense	75	37	512	37	3196	1	Modulo 512
Mushroom	Dense	119	23	512	23	8124	1	
T10I4D100K	Sparse	870	10	512	10	100000	1	
T40I10D100K	Sparse	942	39	512	38	100000	1	
Retail	Sparse	16470	10	512	10	88162	1	
Accidents	Sparse	468	33	512	33	340183	1	
Chess	Dense	75	37	1024	37	3196	1	Modulo 1024
Mushroom	Dense	119	23	1024	23	8124	1	
T10I4D100K	Sparse	870	10	1024	10	100000	1	
T40I10D100K	Sparse	942	39	1024	39	100000	1	
Retail	Sparse	16470	10	1024	10	88162	1	
Accidents	Sparse	468	33	1024	33	340183	1	

We note that different hash functions are implemented like *Modulo* and *MD5* with *Modulo*.

The *Table 4a* shows that "*Modulo 128*" hash function provides, in the worst case, a signature which 29% of the bits are equal to 1, while "*Modulo 512*" gives 7% and "*Modulo 1024*" 4%. We note that the hash function "*MD5 + Modulo 512*" gives a high average weight. In the best cases, 43% of the bits are 1 (*Table 4b*).

We also note that the average number of signatures per leaf node is equal to 1. The results show, in fact, that we have no transactions that generate the same signature and we have no false drops for dense and sparse databases.

Table 4b. "MD5 + Modulo" Hash Function

Database	Type	I-Nbre	I-Avg	Size-S	A-W	Size-B	A-T
Chess	Dense	75	37	512	467	3196	1
Mushroom	Dense	119	23	512	395	8124	1
T10I4D100K	Sparse	870	10	512	237	100000	1
T40I10D100K	Sparse	942	39	512	465	100000	1
Retail	Sparse	16470	10	512	218	88162	1
Accidents	Sparse	468	33	512	455	340183	1

4.2. Performance Comparaison of ST-Tree and FP-Tree

In this section, we use *ST-Mine*, based on *ST-Tree*, and *FP-Growth*, based on *FP-Tree*, to generate frequent itemsets. We consider several transaction databases.

The parameter which is considered in the analysis is the space complexity. We compare the results of *ST-Tree* with that of *FP-tree*. Our observed results are listed in Table 5.

Table 5. ST-Tree Results

Database	Used Memory (Kbytes)
Chess	75
Mushroom	190
T10I4D100K	7966
T40I10D100K	2089
Retail	2342
Accidents	1931

We note that the memory required by our structure (*ST-Tree*) is independent of any parameter, specially support threshold. Each transaction database 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.

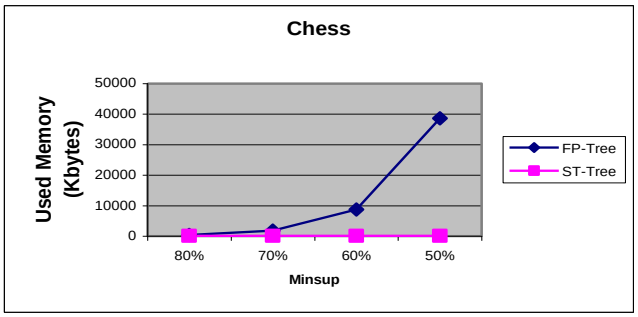


Figure 2. ST-Tree vs FP-Tree for Chess Database

Figure 2 shows the used memory for Chess database for different values of *Minsup*. Our approach gives a better result and that the difference becomes very important for *Minsup* $\leq$ *60%*.

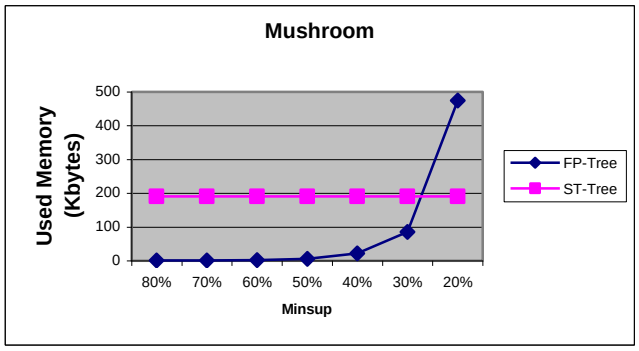


Figure 3. ST-Tree vs FP-Tree for Mushroom Database

Figure 3 shows the used memory for Mushroom. *FP-Tree* result is better for *Minsup* $\geq$ *26%*, but our approach gives a better result for *Minsup* $\leq$ *26%*.

We note, in Figure 4, the same result for *T10I4D100K*. We have a better result for *Minsup* $\leq$ *1,4%*.

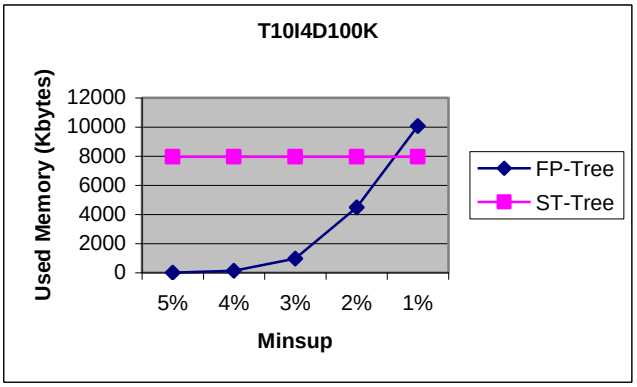


Figure 4. ST-Tree vs FP-Tree for T10I4D100K Database

For *T40I10D100K* database (Figure 5), our results are far better than *FP-Tree*. The difference becomes more evident when the support decreases.

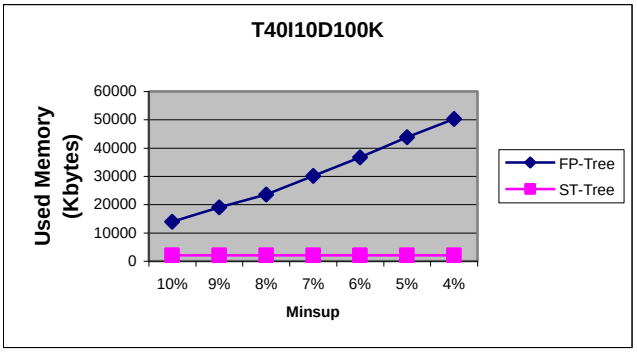


Figure 5. ST-Tree vs FP-Tree for T40I10D100K Database

The figure 6 (Accident database) also shows that our results become better when *Minsup* $\leq$ *55%*.

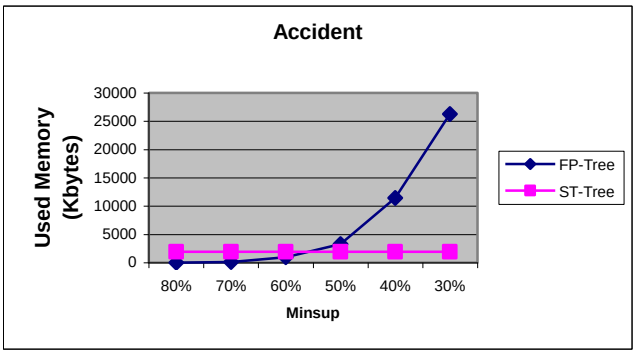


Figure 6. ST-Tree vs FP-Tree for Accident Database

5. Conclusion

In this paper, we have proposed a new compact structure to represent signatures transaction in a tree called *ST-Tree*. Each edge of *ST-Tree* is labeled with "0" or "1" and each internal node contains a number to specify which bit to check in a signature. Thus, the searching of a signature uses only a binary signature tree and need only one access to transactions database.

The complexity of the *ST-Mine* algorithm is polynomial. In order to show the efficiency of our approach, we have conducted a serie of experimentations to compare our proposal with the *FP-Growth* algorithm, using different database transactions and different supports. The results of these experimentations show that *ST-Tree* uses less memory than *FP-Tree* for small *Minsup*.

Reference

[1] R. Agrawal and R. Srikant, "Fast Algorithm for Mining Association Rules", Proceeding of the 20th VLDB Conference Santiago, September 12-15, Chile, 1994, pp. 487-499.

[2] R. Agrawal, T. Imieliński and A. Swami. "Mining association rules between sets of items in large databases", Proceedings of the 1993 ACM SIGMOD, International Conference on Management of Data, New York, NY, USA, 1993, pp. 207-216.

[3] 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.

[4] F. Bodon, "A Fast APRIORI Implementation", Proceeding of 19th Workshop on Frequent Itemset Mining Implementations, in conjunction with the 3rd IEEE International Conference on Data Mining, November 19, Melbourne, Florida, USA, 2003, pp. 16-25.

[5] Y. Chen, "Signature Files and Signature Trees", Information Processing Letters 82, Elsevier, Amsterdam, Pays-Bas, 2002, pp 213-221.

[6] Yangjun Chen and 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.

[7] Yangjun Chen and Yibin Chen,"On the Signature Tree Construction and Analysis", IEEE Transactions on Knowledge and Data Engineering, vol. 18, Issue 9, September 2006, pp. 1207-1224.

[8] C. Faloutsos, "Signature Files: Design and Performance Comparaison of Some Signature Extraction Methods", ACM Sigmod Record, Volume 14, Issue 4, May 1985, pp. 63 – 82.

[9] C. Faloutsos and S. Christodoulakis, "Optimal Signature Extraction and Information Loss", ACM Transac tions On Database Systems, Vol. 12, No. 3, 1987, pp 395-428.

[10] C. Faloutsos, "Access methods for text", ACM Computer Survey, 17(1), 1985, pp. 49-74.

[11] J. Han, J. Pei and Yin Y., "Mining frequent patterns without candidate generation", Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 14-19, Dallas, USA, 2000, pp. 1-12.

[12] D.L. Lee, Y.M. Kim and G. Patel, "Efficient Signature File Methods for Text Retrieval", IEEE TKDE, Vol. 7, No. 3, 1995, pp 423-435.

[13] W. Lee and D.L. Lee, "Signature File Methods for Indexing Object-Oriented Database Systems", Proceeding of the 2nd International Computer Science Conference, Hong Kong, Dec. 13-16, 1992, pp 616-622.

[14] J. Pei, J. Han and R. Mao, "CLOSET: an efficient algorithm for mining frequent closed itemsets", Proceedings of the ACM-SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, May 14, Dallas, USA, 2000, pp. 21-30.

[15] B. Prasetyo, I. Pramudiono M. Kitsuregawa, "Hmine-rev: toward H-mine parallelization on mining frequent patterns in large databases", Technical Report, Institute of Electronics, Information and Communication Engineers, 2005, pp.49-54.

[16] C. S. Roberts, "Partial-Match Retrieval via the Method of Superimposed Codes", Proceeding of the IEEE, Vol. 67, No. 12, 1979, pp 1624-1642.

[17] E. Shanthi, "Applying SD-Tree for Object-Oriented Query Processing", Journal of Informatica 33, June 2009, pp. 177-187.

[18] D. Comer, "The Ubiquitos B-Tree", Computing Surveys, Vol. 11, No. 2, 1979, pp121-137.

[19] C. Faloutsos and R. Chan, "Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and Performance Comparison", Proceeding of VLDB, 1988, pp 280-293.

[20] FIMI repository, http://fimi.cs.helsinki.fi/data. (September 2011)

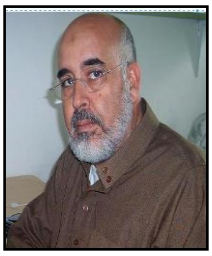
Authors

Mohamed El Hadi Benelhadj got his Engineering and Magister degree in Computer Science from Mentouri University, Constantine, Algeria. Currently, he is an Assistant Professor at the same University. He is a Research Associate in the Research group "Information Systems and Knowledge Bases". His research interests include Datamining and KDD.

Khedija Arour received his Engineering diploma and Ph.D. degrees from the department of Computer Science of the Science Faculty of Tunis, Tunisia in 1992 and 1996, respectively. She is currently an assistant professor in the department of Computer Science and Mathematics at National Institute of Science and Applied Technology of Tunis, Tunisia, Carthage University. Dr. AROUR's research interests are mainly on haute performance data mining and large scale information retrieval systems.

Mahmoud Boufaïda is a full professor in the Computer Science department of the University of Constantine, Algeria. He heads the research group 'Information Systems and Knowledge Bases'. He has published several papers in international conferences and journals. He has managed and initiated multiple national and international level projects including interoperability of information systems and integration of applications in organizations. He has been program committee

member of several conferences. His research interests include cooperative information systems, web databases and software engineering.



Yahya Slimani studied at the Computer Science Institute of Alger's (Algeria) from 1968 to 1973. He received the B.Sc. (Eng.), Dr Eng and PhD degrees from the Computer Science Institute of Alger's (Algeria), University of Lille (French) and University of Oran (Algeria), in 1973, 1986 and 1993, respectively. His currently Full Professor at the Department of Computer Science of Faculty of Sciences of Tunis. These research activities concern Datamining, Parallelism, Distributed systems, Grid and Cloud Computing. Prof. Yahya Slimani has published more than 210 papers from 1986 to 2011. He contributed to Parallel and Distributed Computing Handbook, Mc Graw-Hill, 1996. He is member of Editorial Boards of several international journals and chair of international conferences.