Prefix Tree with Encryption of Data and Itemsets

Ramkishore Bhattacharyya
Department of Computer Science and Engineering
Jadavpur University
Kolkata
India
rk_ju@yahoo.com

Abstract

The principal criterion of any mining algorithm is to welcome influx of huge data that poses a real challenge to space-time requirement. Unless data are arranged in a compact and efficient way, algorithms, with limited primary storage, fail to produce output within reasonable time.
In this paper, we present an encryption technique for data leading to construction of a highly compact and storage efficient prefix tree with little overhead. In the representation, transactions with a common prefix can be arranged together, thereby increasing processing capability of mining algorithms. We also introduce a new data structure, Binary Search Prefix Tree (BSPT), for systematic management of encrypted itemsets. Experimental results show that incorporation of the new data structures makes an algorithm scalable to a large extent.

Keywords: transaction prefix tree, data partitioning, binary search prefix tree, Apriori algorithm

1. Introduction

Scalability of any frequent itemset mining algorithm with decreasing threshold support as well as increasing number of transactions highly depends on how data has been stored and processed. The classical Apriori algorithm, introduced by Aggarwal et al [1,2] relies on generation-and-test approach of candidates. At k^{th} iteration, candidate k-itemsets are generated using the set of frequent (k-1)-itemsets at the previous iteration. The algorithm, along with several efficient data structures such as hash tree, trie, prefix tree etc, is able to exhibit good performance with sparse datasets that fail to generate patterns of higher length. However with dense datasets, capable of generating several long patterns even at considerable threshold support, performance of this algorithm falls sharply mainly due to generation of huge candidates and their support computation through pattern matching.

International Conference on Management of Data
COMAD 2006, Delhi, India, December 14-16, 2006

A number of vertical mining techniques [8,9] have been proposed that incorporate vertical representation of a dataset and thereby getting rid of pattern matching. In a vertical mining algorithm, each itemset is associated with a set of identifiers of transactions (tid) containing the itemset. Support of a candidate is obtained either directly by intersecting the tidsets of the two generating frequent itemsets or by keeping track of the difference of two tidsets (diffset). But this is at the cost of higher memory consumption required to maintain tidsets or diffsets of frequent itemsets. Furthermore, tidset intersection quickly loses its advantage when their cardinality becomes very large which is usual with dense datasets. Both Apriori and vertical mining techniques require equal proportion of execution time with increase in dataset size.
The FP-growth algorithm [6], with a novel data structure FP-tree, is able to bypass candidate generation and pattern matching, thereby achieving good scalability. The FP-tree allows common prefixes of different transactions to share storage space. All patterns, having a common prefix, can be mined from the tree without further accessing the original dataset.

2. Problem Statement

The availability of huge raw data poses the following challenges to a mining algorithm:

- Efficient management of data explosion within a limited amount of main memory
- Least possible impact on processing time
- A compact storage structure for efficient management of large number of frequent itemsets

In this study we introduce a data encryption technique to meet the first criteria. We further present a new algorithm that organizes the encrypted data into a prefix-based pattern tree, thereby addressing the second criteria. To attend the third criteria, we establish a one-to-one mapping from the set of itemsets to the set of natural numbers. The mapping, along with the concept of equivalence class decomposition of itemsets [8], comes out with a new data structure that stores itemsets again in encrypted form. We call it a Binary Search Prefix Tree (BSPT).

3. Data Encryption and Transaction Prefix tree(TP-tree)

Let $\mathcal{I}$ = {f,a,g,c} be a set of frequent items sorted in descending order of support. A frequent item, getting a position i in $\mathcal{I}$ is represented by a bit-vector of length $|\mathcal{I}|$ with a set bit at i^{th} position from MSB. The equivalent natural number is the identifier of the item.

$f = 1000_2=8_{10}$, $a = 0100_2=4_{10}$, $g = 0010_2=2_{10}$,$c = 0001_2=1_{10}$

Let **t** be a transaction containing n items $i_1, i_2, ...,i_n$.

$$Id(\mathbf{t}) = \{ \sum_{k=1}^{n} Id(i_k) \mid i_k \in \mathcal{I} \} \ \forall \mathbf{t} \in \mathcal{D}$$

For efficient storage and faster processing, the encrypted transactions are arranged into a highly compact storage structure TP-tree as defined below.

Definition 1:

1. A TP-tree is a binary tree with a finite set of nodes which is either empty or consists of a root and at most two disjoint TP-trees, called the left TP-tree and the right TP-tree.
2. The root contains the most frequent item with its frequency of occurrences in the dataset.
3. The left subtree is the TP-tree of transactions that do not contain the item at the root.
4. The right subtree is the TP-tree of transactions that contain the item at the root.

Fig 1 presents the algorithm for TP-tree construction.

Lemma 1 Let i be the most frequent item in a set of transactions $\mathcal{D}$. Then $\forall \mathbf{t} \in \mathcal{D}$, $Id(\mathbf{t}) \geq Id(i) \Leftrightarrow i \in \mathbf{t}$

Proof: Let $|\mathcal{I}| = n$. So $Id(i) = 2^{n-1}$. If possible, let $Id(\mathbf{t}) \geq Id(i)$ but $i \notin \mathbf{t}$. A transaction without item i can have maximum identifier when all the remaining items are associated to it. So,

$$Max(Id(\mathbf{t})) = \sum_{k=0}^{n-2} 2^k = 2^{n-1}-1 < 2^{n-1} = Id(i)$$

So, $Id(\mathbf{t}) < Id(i)$. But this is a contradiction to our assumption that $Id(\mathbf{t}) \geq Id(i)$. Hence, $Id(\mathbf{t}) \geq Id(i) \Rightarrow i \in \mathbf{t}$.

Again, consider $i \in \mathbf{t}$. In that case **t** may contain some other items apart from i which contribute a non-negative identifier value to **t**. So, $Id(\mathbf{t})$ = { Id(i) + a non-negative identifier} $\geq Id(i)$ which implies $Id(\mathbf{t}) \geq Id(i)$. Hence, $i \in \mathbf{t} \Rightarrow Id(\mathbf{t}) \geq Id(i)$. Together, the result follows.

Corollary 1 Let i be the most frequent item in a set of transactions $\mathcal{D}$. Then $\forall \mathbf{t} \in \mathcal{D}$, $Id(\mathbf{t}) < Id(i) \Leftrightarrow i \notin \mathbf{t}$.

Proof: As a consequence of lemma 1, $\neg (Id(\mathbf{t}) \geq Id(i)) \Leftrightarrow \neg(i \in \mathbf{t})$ which implies $Id(\mathbf{t}) < Id(i) \Leftrightarrow i \notin \mathbf{t}$, '$\neg$' denotes complement operation. So all of $\mathcal{D}'_R$ and none of $\mathcal{D}'_L$ contain the itemset represented by pivot. Again $\mathcal{D}'_L \cup \mathcal{D}'_R = \mathcal{D}'$ and obviously $\mathcal{D}'_L \cap \mathcal{D}'_R = \emptyset$. Hence is the statement in line 8.

Number of items in $pivot_L$ remains same by removing the least frequent item from the current pivot and including the next frequent item. This is actually done in line 11. Number of items in $pivot_R$ is increased by one simply by including the next frequent item. This is done in line 12.

Algorithm 1(TP-tree Construction)
Input: encrypted dataset $\mathcal{D}'$, m initialized to $|\mathcal{I}|$-1, pivot initialized to 2^m
Output: The transaction prefix tree, TP-tree
1. begin
2. partition $\mathcal{D}'$ into $\mathcal{D}'_L$ and $\mathcal{D}'_R$
3. $\mathcal{D}'_L = \{Id(\mathbf{t}) \mid Id(\mathbf{t}) < pivot\}, \mathbf{t} \in \mathcal{D}'$
4. $\mathcal{D}'_R = \{Id(\mathbf{t}) \mid Id(\mathbf{t}) \geq pivot\}, \mathbf{t} \in \mathcal{D}'$
5. if $\mathcal{D}'_R \neq \emptyset$ then
6. declare a new node p
7. item[p] ← pivot
8. sup[p] ← $|\mathcal{D}'_R|$
9. $\mathcal{D}''_R \leftarrow \mathcal{D}'_R \setminus \{pivot\}$
10. end if
11. $pivot_L \leftarrow pivot - 2^{m-1}$
12. $pivot_R \leftarrow pivot + 2^{m-1}$
13. if m > 0 then
14. if $\mathcal{D}'_L \neq \emptyset$ then
15. left[p] ← TP-tree($\mathcal{D}'_L$, $pivot_L$, m-1)
16. if $\mathcal{D}''_R \neq \emptyset$ then
17. right[p] ← TP-tree($\mathcal{D}''_R$, $pivot_R$, m-1)
18. end if
19. return p
20. end

Fig 1. Algorithm for TP-tree Construction

Analysis of algorithm The algorithm is recursive with two recursive calls. At every level of recursion tree, partition step of line 2 together takes $O(|\mathcal{D}'|)$ time complexity. The depth of recursion tree can be at most m i.e. $|\mathcal{I}|$-1. So worst case time complexity for the algorithm is $O(|\mathcal{I}|*|\mathcal{D}'|)$. Height of TP-tree is bounded by $|\mathcal{I}|$.

FP-tree vs. TP-tree

The new tree contains all the properties of FP-tree[6] with almost same storage requirement except the lateral node links which can easily be added when mining with FP-growth algorithm. A node in TP-tree contains entire information about the prefix in a compressed form. So a backward journey starting from a node up to the root is not required to collect entire prefix information.

4. Data Structure for Itemset Arrangement

We further introduce a function that essentially associates each itemset to a unique natural number. Let s = {i_1 i_2 i_3 …… i_k} be an itemset with i_1, i_2, i_3,……,i_k having respective identifiers $n_1, n_2, n_3, ..., n_k$

$\mathcal{F}(s) = \bigvee_{j=1}^{k} (n_j)$, '$\vee$' denotes bitwise-OR operation.

$fag = f \cup a \cup g = (8)_{10} \vee (4)_{10} \vee (2)_{10} = (14)_{10}$
$fac = f \cup a \cup c = (8)_{10} \vee (4)_{10} \vee (1)_{10} = (13)_{10}$

A set of n frequent items can produce 2^n itemsets (power set) forming a lattice under subset relation. A prefix-based

equivalence relation (ρ_k) decomposes the power set lattice into a number of disjoint equivalence classes, k being the length of prefix. We arrange the lattice in the form of a binary tree. Each node, along with its entire right subtree represents an equivalence class determined by the prefix at the node. Corresponding left subtree does not contain any itemsets with that prefix. Replacement the itemsets with their corresponding identifiers yields the binary search tree in Fig 2. As it is essentially a prefix tree with an additional property of binary search, it is termed as Binary Search Prefix tree (BSPT).

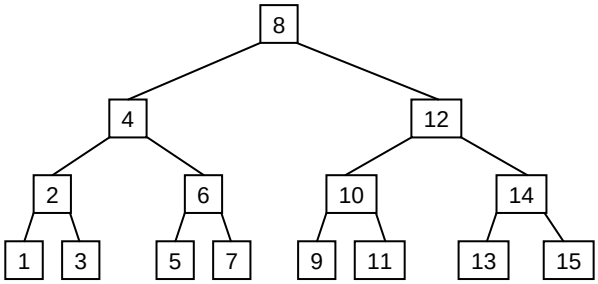

Fig 2. The Binary Search Prefix Tree

5. Algorithm for BSPT Construction

Our method adopts the generate-and-test approach of Apriori. Prefix-based storage structure enables candidates

```
Algorithm 2 (BSPT-Construction)
Input: set of freq. 1-itemsets I, BSPT root T
Output: root of BSPT containing all freq. itemsets
1.  for i = |I| downto 1 do
2.    Xi ← delete ith itemset from I
3.    set Xi as new root of BSPT
4.    make T its left subtree and T the new root
5.    initialize Ri to ∅   /* Ri is a set*/
6.    for all other itemsets Xj in left subtree of Xi
        having same no. of items as that of Xi do
7.      X←generate sound candidate from Xi and Xj
8.      if support of X >= threshold support
9.        Ri← {Ri ∪ X}
10.   end for
11    if Ri != ∅
12.     right[T] ← BSPT-Construction (Ri , right[T])
13. end for
14. return T
```

to be generated through bitwise-OR operation in O(1) complexity. Nodes of TP-tree are decomposed into suffix-based equivalence classes determined by frequent items. For support computation, an itemsets picks up the equivalence class determined by its suffix item. Since both data and itemsets are encrypted to natural numbers, pattern matching is done through bitwise-AND operation in O(1) complexity.

Lemma 3 Properties of BSPT are not violated by inserting a frequent itemset as the root of the right subtree of its prefix.

Proof Let i^{th} element of I be X_i. Since $\forall j | (j > i)$, $X_j < X_i$, R_i is sorted in descending order of identifier value. Again, $\forall X_k' \in R_i$, X_k' contains a prefix X_i and $X_k' > X_i$. So there is no harm in inserting X_k' in the right subtree of X_i. Since $\forall l | (l > k)$, $X_l' < X_k'$, making X_k' the root will not violet the BST property as all $X_l's (< X_k')$ appear in the left subtree of X_k'. This completes the proof.

Lemma 4 Working storage of BSPT-Construction algorithm is bounded by cardinality of I.

Proof: The additional storage, required by the algorithm, is due to maintaining the set of frequent prefixes (viz. R_i). The way prefixes are stored in the set, indices of the last items of the prefixes form a strictly monotonic increasing sequence starting from 1. So in the worst case, the algorithm needs to maintain at most $|I|$ prefixes.

6. Experimental Results and Analysis

Algorithm BSPT-Construction is coded in C and run on a computer system having AMD Athlon XP2400 processor with 512 MB RAM and Red Hat Linux 9.0 operating system. For preciseness of presentation, results have been presented only for the following two datasets available at FIMI repository [http://fimi.cs.helsinki.fi/].

Dataset		# items	Avg. trans length	# trans
Connect		130	43	67557
Pumsb		7117	74	49046

Table 1. Characteristics of datasets for experiments

Fig 3 depicts a comparative study on computation time with several other implementations of Apriori [3,4,7], Eclat[3], dEclat[4], and FP-growth[4] algorithms, results being plotted in logarithmic scale. It shows that BSPT-Construction with TP-tree structure outperforms all Apriori implementations for all datasets by a considerable margin even if it relies on generate-and-test approach of classical Apriori. Unlike to Apriori, BSPT-Construction performs better in dense and voluminous datasets compared to sparser ones. More interestingly, up to a considerable threshold support, its performance is comparable to Eclat, dEclat and FP-growth which are shown to outperform Apriori by a large factor at any support especially in dense datasets. The main reason for performance improvement is bit-vector encryption of both data and itemsets that enable candidate generation and pattern matching to be done in O(1) complexity irrespective of number of items in candidates.

Since experiments are performed with real datasets, there is no way other than replicating the dataset several times to check the response of the algorithm with large scale increase in number of transactions. Fig 4 shows that all other algorithms, except the FP-growth, require equal proportion of time with increase in volume of data. Time requirement also increases in FP-growth algorithm, but with a lower slope. Computation time of BSPT-Construction remains almost constant with increase in bulk of transactions as TP-tree construction is very fast

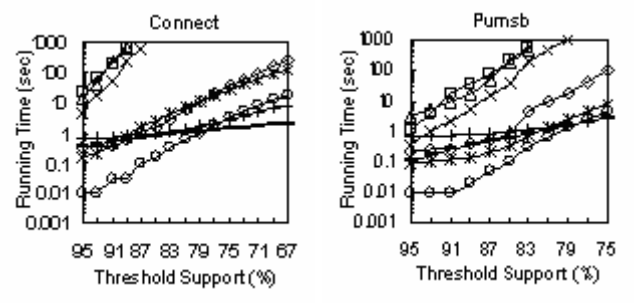

Fig 3. Performance Analysis

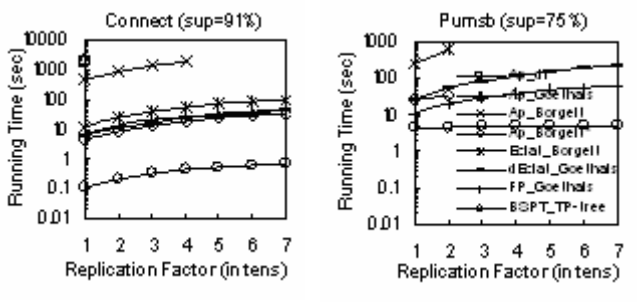

Fig 4. Scalability Analysis

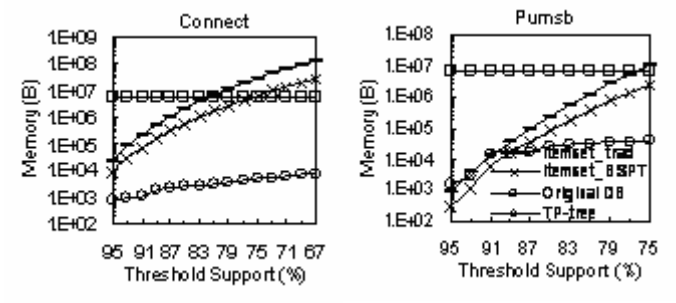

Fig 5. Memory Compaction

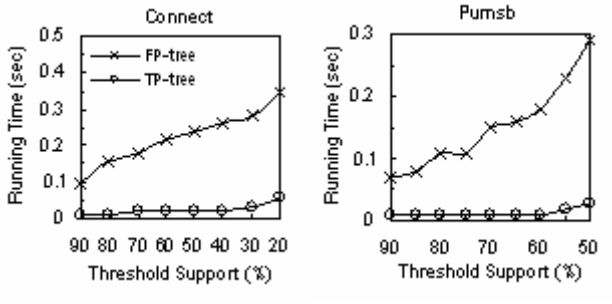

Fig 6. TP-tree Performance Analysis

with a least possible overhead. Thus, BSPT-Construction mines voluminous dataset within reasonable time compared to other algorithms.

Fig 5 depicts a comparative study of input and output storage, required for dataset and frequent itemsets respectively, of our encryption technique with traditional storage structure. The TP-tree of encrypted data reduces compression factor to an impressive value though it gradually increases. Bit-vector encryption of candidates allows frequent itemsets to be stored within 25% or even less storage compared to traditional one. It is worth mentioning that storage minimization is much more impressive in dense datasets compared to sparser ones. The reason is in dense datasets, $|I|$, the number of frequent items, increases less rapidly with decrease in threshold support.

Finally in Fig 6 we compare TP-tree construction time with that of FP-tree in [5] to prove its superiority. Construction of TP-tree does not require items in a transaction to be sorted in descending order of support as FP-tree does. Searching for a node repeatedly to increment its support count does contribute a lot in the construction time of FP-tree. On the contrary, support of a node is finalized right at the time of its creation in TP-tree. Furthermore, experimentally we reveal that row scalability of TP-tree algorithm is remarkably better than FP-tree which actually makes BSPT-Construction highly row scalable.

The present work reveals that appropriate coding for data and itemsets can improve performance of a mining algorithm as in the case of Apriori algorithm.

References

[1] R. Aggarwal, T. Imielinski and A. Swami, “Mining Association Rules between Sets of Items in Large Databases”, ACM SIGMOD Conf. on Management of Data, 1993

[2] R. Aggarwal and R. Srikant, “Fast Algorithm for Mining Association Rules”, Proc. 20th Very Large Database(VLDB) Conf. pp. 487-499, 1994

[3] C. Borgelt, “Efficient Implementations of Apriori and Eclat”, FIMI’03: Workshop on Frequent Itemset Mining Implementations, 2003

[4] B. Goethals, “Survey on Frequent Pattern Mining”, Helsenki 2003, http://www.cs.helsinki.fi/u/-goethals/ publications/survey.ps

[5] G. Grahne and J. Zhu, “Efficiently using prefix trees in mining frequent itemsets”, In B. Goethals and M.J. Zaki, editors, Proc. Of the IEEE ICDM Workshop on frequent itemset mining implementations (FIMI 2003), vol 90 of CEUR Workshop Proceedings, 2003.

[6] J. Han, J. Pei and Y. Sin, “Mining Frequent Pattern without Candidate Generation”, ACM SIGKDD, 2000

[7] W. A. Kosters and W. Pijls, “Apriori, A Depth First Implementation”, 2003, FIMI repository: http://fimi.cs.helsinki.fi/

[8] M. J. Zaki, “Scalable Algorithms for Association Rule Mining”, IEEE Trans. on Knowledge and Data Engg vol 12, no 3, pp 372-390, 2000

[9] M. Zaki and C. Gouda, Fast vertical mining using diffsets, Proc. of ACM SIGKDD’03, Washington DC, Aug.2000