Volume 2, Issue 2, February 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Comparison of FP-Tree and B-Tree in Data Mining

Pallavi Sharma*, Lokesk Kumar Joshi
Department of Computer Science and Engineering,
Arya College of Engineering & Technology,
Jaipur ,Rajasthan, INDIA

Abstract: Association rule mining, one of the most important and well researched techniques of data mining, was first introduced in. It aims to extract interesting correlations, frequent patterns, associations or casual structures among sets of items in the transaction databases or other data repositories.. However, no method has been shown to be able to handle data streams, as no method is scalable enough to manage the high rate which stream data arrive at. More recently, they have received attention from the data mining community and methods have been defined to automatically extract and maintain gradual rules from numerical databases. In this paper, we thus propose an original approach to mine data streams for Association rule mining. Our method is based on B-Trees and FP growth in order to speed up the process. B-Trees are used to store already-known for order to maintain the knowledge over time and provide a fast way to discard non relevant data while FP growth.

1. Introduction of FP growth

The problem of mining association rules from a data stream has been addressed by many authors but there are several issues (as highlighted in previous sections) that remain to be addressed. In the following section existing literature based on the problems in data stream mining that is addressed.

The work in this domain can be effectively classified into three different domains namely, Exact methods for Frequent Itemset Mining, Approximate Methods and Memory Management techniques adopted for data stream mining[1].

2. Exact approaches to Frequent Itemset Mining

Fequent-pattern mining plays an essential role in mining associations[1] if any length k pattern is not frequent in the database, its length (k + 1) super-pattern can never be frequent. The essential idea is to iteratively generate the set of candidate patterns of length (k+1) from the set of frequent-patterns of length k (for k $\geq$ 1),and check their corresponding occurrence frequencies in the database.

The Apriori heuristic achieves good performance gained by (possibly significantly) reducing the size of candidate sets. However, in situations with a large number of frequent patterns, long patterns, or quite low minimum support thresholds, an Apriori-like algorithm may suffer from the following two nontrivial costs:– It is costly to handle a huge number of candidate sets. For example, if there are 104 frequent 1-itemsets, the Apriori algorithm will need to generate more than 107 length-2 candidates and accumulate and test their occurrence frequencies. Moreover, to discover a frequent pattern of size 100, such as {a1, . . . , a100}, it must generate 2100 − 2 ≈ 1030 candidates in total.

This is the inherent cost of candidate generation, no matter what implementation technique is applied.– It is tedious to repeatedly scan the database and check a large set of candidates by pattern matching, which is especially true for mining long patterns.Can one develop a method that may avoid candidate generation-and-test and utilize some novel data structures to reduce the cost in frequent-pattern mining? This is the motivation of this study[5].

In this work, we develop and integrate the following three techniques in order to solve this problem.First, a novel, compact data structure, called frequent-pattern tree, or FP-tree in short,is constructed, which is an extended prefix-tree structure storing crucial, quantitative information about frequent patterns. To ensure that the tree structure is compact and informative, only frequent length-1 items will have nodes in the tree, and the tree nodes are arranged in such a way that more frequently occurring nodes will have better chances of node sharing than less frequently occurring ones.

Subsequent frequent-pattern mining will only need to work on the FP-tree instead of the whole data set. Second, an FP-tree-based pattern-fragment growth mining method is developed, which starts from a frequent length-1 pattern (as an initial suffix pattern), examines only its conditional-pattern base (a "sub-database" which consists of the set of frequent items co-occurring with the suffix pattern), constructs its (conditional) FP-tree, and performs mining recursively with such a tree[5][6]. The pattern growth is achieved via concatenation of the suffix pattern with the new ones generated from a conditional FP-tree.

3 PROPOSED METHOD

3.1 Introduction

The previous chapters have described the fundamental background behind closed itemset mining, work objectives, overall architecture, and experimental design. This chapter will focus on the experimental findings. Both B treee and FP were tested on synthetic datasets and compared against predefined performance metrics such as Accuracy, computational performance, and Memory consumption. Supposed our Database is ginven in this format.

Transactions	SALARY	CAR
T1	2000	2
T2	3000	3
T3	3500	4
T4	2500	4
T5	1000	1
T6	4000	3

Table 1.(example)

3.2 Findings from Experiment 1

This experiment was mainly designed for comparing Data structure using Btree and FP Tree with respect to performance. We first varied the minimum support threshold while keeping the delta parameter constant. We recorded the accuracy, performance and memory consumption for Data structure and then repeated the procedure for FP tree. For this experiment, we have used dense datasets generated using the IBM data generator (IBM). The Recall and Precision were calculated by comparing Data structure using FP Tree and FP tree results against the Apriority implementation

The Apriority implementation was run against data batched across a fixed number of frames and presented as a single unit fixed size dataset to the Apriori algorithm. It should be stressed that Apriori was only used for benchmarking purposes for the Precision and Recall values. As mentioned earlier in the thesis Apriori cannot be used in an actual data stream environment.

In this section, we firstly give an overview of our approach focusing on the requirements needed in a data stream mining scenario. Following, we explain in detail the gradual rule mining algorithms used in our approach. Finally, provide the complexity of the most costly algorithm.

Transactions	SALARY	CAR	B-TREE
T1	0.4	0.4	0.4
T2	0.6	0.6	0.6
T3	0.7	0.8	0.75
T4	0.5	0.8	0.65
T5	0.2	0.2	0.2
T6	0.8	0.6	0.7

Table 2: Normalized Database (min Salary=0, max Salary=5,000,min Cars=0, max Cars=5), and b tree value.

3.3 Basic Idea

We consider databases such as described in Table 2 where attribute values have been normalized in [0, 1], as shown by Table 2. For this purpose, we consider a minimum and maximum value for every attribute1.

In order to have a global idea of each tuple for ordering them, we compute a summary using an BTree. Note that it is very easy to manage both increasing and decreasing gradualness. Indeed, as weights give importance to small or large values, it is not the same to mine a gradual itemset like {(Salary, +)(Cars, +)} or {(Salary, +)(Cars,−)}. For the former case, we have to compute the Btree value as B tree(asalary, acars), whereas, for the latter one, we have to compute the Btree value as Btree(asalary,(1−acars)).

For instance, the last column of the Table 2 reports the value computed by giving the same weight to every attribute value. Here shows the process for mining the gradual itemset {(Salary, +)(Cars, +)}.

Example 1 Consider each attribute of the normalized database of Table 2 as the data coming from the data stream, and suppose that the gradual itemset to mine is GI: {(Salary, +)(Cars, +)}. At the beginning the corresponding B-Tree for GI is empty. At time ts1, the tuple T1 arrives andB tree(T1) is computed. As the B-Tree is empty, T1 is inserted in the B-Tree root node. At time Ts2, the tuple T2 arrives and Btree(t2) is computed. As Btree(t1) <=Btree(T2), we have to check that T1.Salary ,= T2.Salary and t1.Cars

<=T2.Cars hold. As both conditions are fulfilled t2 is inserted in the B-Tree (as it is shown in Figure 2.(b)).This process is repeated at time Ts3 with tuple T3 checking that T3 attributes

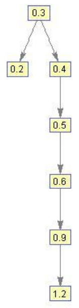


Fig 2(b) BTree insertion Example

are larger thanT2 attributes (Figure 2.). Then, at time Ts4, the tuple T4 arrives and the tree(T4) is computed. As Btree(T2) <= (T4) <= Btree(T3), we have to check that T2.Salary <=T4.Salary <=T3.Salary and T2.Cars <= T4.Cars <= T3.Cars hold. As T2.Salary = 0.6 and T4.Salary = 0.5 the first condition does not hold and tuple T4 cannot be inserted in the B-Tree. After that, tuple T5 is inserted at time Ts5 because Btree(T5) <=Btree(T1) and T5.Salary <=T1.Salary and T5.Cars <= T1.Cars hold, this is depicted in Figure 2(b). Finally, at time Ts6 tuple t6 is discarded in a similar way as T4.This is the key idea.

3.3.1 Algorithm Definition

In this section, we provide the algorithms and all the details of our proposal.We assume that tuples coming from the stream are already normalized.Algorithm 1 is in charge of initializing the gradual itemsets and B-Trees lists (lines 2-3). Basically, such initialization consists in creating all the possible gradual itemsets taking into account the number of attributes of tuples, and creating for each one an empty B-Tree. Following, we have to compute when the first pruning process will be executed . As we are interested in maintaining in memory all tuples belonging to a certain period of time we have to prune the B-Trees for the first time when we have in memory tuples of two window frames. In the main loop of Algorithm 1, we process the data.

```
Algorithm 1: Data Structure using Btree
Data: s: Stream, q: Quantifier, w: Window
Step 1:-   Begin
Step 2 :-  g = Gradual itemset list;
Step3:-    b = B-Tree list;
Step 4 :-  InitializeGradualRule(g);
Step 5:-   InitializeBTree(b);
Step 6 :-  np = now() + 2w;
Step 7 :-  while (!s.empty()) do
Step 8 :-  GradualRuleProcessing(g,b,s,q);
Step 9 :-  if (now() <=np) then
Step 10:-  BTreePruning(b,w);
Step 11:-  np = now() + w;
```

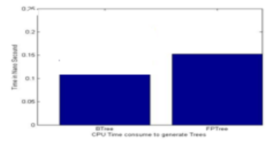
The most costly algorithm is the Algorithm 2. The complexity of this algorithm is related to the number of gradual itemsets (s) and to the number of tuples that we have to process in a window frame (n).B-Tree operations.

```
Algorithm 2:
    Data: itemset list, b: B-Tree list, s: Stream, q:Quantifier
Step 1:- begin
Step 2 :-while (!s.empty()) do
Step 3 :- a=read(s);
Step 4:- for i ( 1 to g.Size do
Step 5:- ReverseValues(g.get(i), a);
Step 6:- o = Compute Btree(a, q);
Step 7:- prev = PreviousNodeSearch(o, b.get(i));
Step 8:- susbseq = SubsequentNodeSearch(o, b.get(i));
Step 9:- if (checkRule(g.get(i), b.get(i), a, prev, subseq)) then
Step 10:- b.get(i).addNode(o, a, timestamp);
Step 11:- UpdateSupport(g.get(i));
Step 12:- end
```

Used in this algorithm (search and insertion) have a complexity equal to O(log (n)). In the worst scenario, for each new tuple, we have to execute two B-Tree operations (one search and one insertion) for each gradual itemset to mine. Therefore, the complexity of this algorithm is equal to O(2s log (n)). As Algorithm 2 has a sub-linear complexity and the tilted-time window technique allows us to keep n as small as possible, we argue that it is affordable for most of real time scenarios.

4. Results Analysis and Discussion:

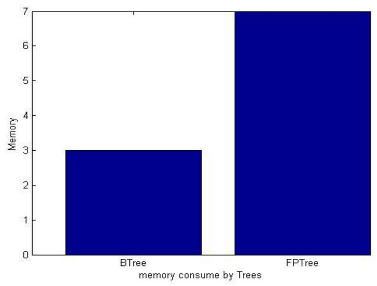
In this section, we present a complete analysis of the experiments carried out in this work, as well as, a short discussion about why the results obtained show that our approach is suitable for the stream mining scenario. The experiments were performed using a Matlab and a windows Operating system.



Graph 4(a)(CPU time to generate a tree)

All the Graphs presented in this section were calculated in the same way. After processing each new tuple the following statistics were computed: the total CPU usage for mining all the graduals itemsets (Figure 4.(a)), the total number of nodes stored in the B-Trees , the average size of Btree .As we have obtained a very large log file, we have computed a smaller one computing the average of these values in groups of 100 elements. The group number is shown in the horizontal axis of all the charts, this gives us a time reference.If we observe in detail the results presented in Figure 4(a), we can see that the CPU time is constant over the time, then it is clear that our approach is able to work in real time scenario.

This graph (fig 4(b))shows the memory utilization of Btree and FP tree. Btree requires less memory as compare to FP tree.



Graph (4b)

However, we should say that the most supported rules are exactly the same in both cases.For this reason, we really believe that such difference is not significative considering the time improvement we obtain.

5. CONCLUSION

Data stream mining is one of the most intensely investigated and challenging work domains in contemporary work in the data mining discipline as a whole. The peculiarities of data streams render conventional mining schemes inappropriate.

In this dissertation we used novel approach for mining the closed item set from a Data stream. We have implemented B-tree to store the closed item set with their support count for this we use Apriori principal to reduce the unnecessary power set creation and prune closed itemset with frequent itemset. Proposed work develop an incremental frequent itemset mining Algorithm based on the Data stream.The Data Stream can find the lot of data in data set.. We compare B-tree with FP tree. Our Experiment show that Btree not only outperformed FP growth but it provide the short time for pruning the frequent itemset.

In this work, we presented an overview of a novel approach for mining the Frequent itemsets from a data stream. We have implemented an efficient closed prefix B-tree to store the intermediate support information of frequent item sets.

REFERENCE

R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. KDD'00, pages 108–118, 2000.

R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In ACM SIGMOD'93, pages 207–216, Washington, D.C.1993.

R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In VLDB'94, pages 487–499, 1994.

R. Agrawal and R. Srikant. Mining sequential patterns. In ICDE'95, pages 3–14, 1995.

B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In Prodeeding of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03), Nov 2003.

G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03), Nov 2003.

J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Mining and Knowledge Discovery, 8:53– 87, 2004.

M. Kamber, J. Han, and J. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In Knowledge Discovery and Data Mining, pages 207–210, 1997.

H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery, 1(3):259–289, 1997.

Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In VLDB'95, pages 432–444, 1995.

H. Toivonen. Sampling large databases for association rules. In VLDB'96, pages 134–145, Sep. 1996.

M. Zaki and K. Gouda. Fast vertical mining using diffsets. In ACM SIGKDD'03, Washington, DC, Aug. 2003.

Claudio Lucchese Mining frequent closed itemsets out of core 2004