

Different Data Mining Algorithms: A Performance Analysis

Vivekananth.P

Lecturer, Botho College, Gaborone, Botswana

Abstract: *Data mining (also called knowledge discovery in databases) represents the process of extracting interesting and previously unknown knowledge (patterns) from data. By applying artificial intelligence together with analytical methods data can be extracted. An association rule expresses the dependence of a set of attribute-value pairs, called items, upon another set of items (item set). . The association rule mining algorithms can be classified into two main groups: the level-wise algorithms and the tree-based algorithms. The level-wise algorithms scan the entire database multiple time but they have moderate memory requirement. The two phase algorithms scan the database only twice but they can have extremely large memory requirement. In this paper a comparative study of the algorithms used in association rules mining apriori and FP growth is done . A performance study has been done which shows the advantages and disadvantage of algorithms.*

1. INTRODUCTION

The association rule mining is a fundamentally important task in the field of data mining. It is a process of discovering not trivial relationships between data in large databases. The problem of association rule mining was first introduced by Agrawal et al in 1993 [1]. Since then it is one of the most popular research area on the field of knowledge discovery.

The association rule mining problem is commonly known as the market basket analysis, but there are several applications that use association rules as well i.e. biological research areas, telecommunication and network analysis etc. Regarding the diversity of the applications that use association rule mining, several algorithms have been developed. All of these algorithms have their own advantages and disadvantages, so it is useful to compare them. Most of the algorithms find all frequent itemsets.

A very influential association rule mining algorithm, Apriori [1], has been developed for rule mining in large transaction databases. Many other algorithms developed are derivative and/or extensions of this algorithm. A major step forward in improving the performances of these algorithms was made by the introduction of a novel, compact data structure, referred to as *frequent pattern tree*, or FP-tree [2], and the associated mining algorithm, FP-growth. The main difference between the two approaches is that the Apriori-like techniques are based on bottom-up generation of frequent itemset combinations

and the FPtree based ones are partition-based, divide-and-conquer methods.

After this introduction, the paper is organized as follows. Section 2 presents the problem definition. The main aspects of Apriori, FP-growth algorithms are presented in Section 3. Section 4 shows a comparative study of the algorithms and the paper is concluded in Section 5.

2. PROBLEM DEFINITION

Association rule mining finds interesting association or correlation relationships among a large set of data items [8]. The association rules are considered interesting if they satisfy both a *minimum support* threshold and a *minimum confidence* threshold [3]. A more formal definition is the following [4]. Let $I = \{i_1, i_2, \dots, i_m\}$ be a set of items. Let D , the task-relevant data, be a set of database transactions where each transaction T is a set of items such that $T \subseteq I$. Each transaction is associated with an identifier, called TID. Let A be a set of items. A transaction T is said to contain A if and only if $A \subset T$. An association rule is implication of the form $A \Rightarrow B$, where $A \subset B$, $B \subset I$, and $A \cap B = \emptyset$. The rule $A \Rightarrow B$ holds in the transaction set D with *support* s , where s is the percentage of transactions in D that contain $A \cup B$ (i.e., both A and B). This is taken to be the probability, $P(A \cup B)$. The rule $A \Rightarrow B$ has *confidence* c in the transaction set D if c is the percentage of transactions in D containing A that also contain B . This is taken to be the conditional probability, $P(B|A)$. That is, *support* $(A \Rightarrow B) = P(A \cup B)$ *confidence* $(A \Rightarrow B) = P(B|A)$

The definition of a frequent pattern relies on the following considerations [5]. A set of items is referred to as an itemset (pattern). An itemset that contains k items is a k -itemset. For example the set $\{name, semester\}$ is a 2-itemset. The occurrence frequency of an itemset is the number of transactions that contain the itemset. This is also known, simply, as the frequency, support count, or count of itemset. An itemset satisfies *minimum support* if the occurrence frequency of the itemset is greater than or equal to the product of *minimum support* and the total number of transactions in D . The number of transactions required for the itemset to satisfy *minimum support* is therefore referred to as the *minimum support count*. If an itemset satisfies *minimum support*, then it is a *frequent itemset* (*frequent pattern*).

The most common approach to finding association rules is to break up the problem into two parts [6]:

1. Find all frequent itemsets: By definition, each of these itemsets will occur at least as frequently as a pre-determined minimum support count [8].
2. Generate strong association rules from the frequent itemsets: By definition, these rules must satisfy minimum support and minimum confidence [8].

Additional interestingness measures can be applied, if desired. The second step is the easier of the two. The overall performance of mining association rules is determined by the first step. As shown in [2], the performance, for large databases, is most influenced by the combinatorial explosion of the number of possible frequent itemsets that must be considered and also by the number of database scans that has to be performed.

3. THE ALGORITHMS IN ASSOCIATION RULES MINING

3.1 The Apriori Algorithm

The first pass of the algorithm simply counts item occurrences to determine the large 1-itemsets. A subsequent pass, say pass k , consists of two phases. First, the large itemsets L_{k-1} found in the $(k-1)$ th pass are used to generate the candidate itemsets C_k , using the Apriori candidate generation function (apriori-gen) described below. Next, the database is scanned and the support of candidates in C_k is counted. The Apriori algorithm is:

```

L1 = {large 1-itemsets};
for ( k = 2; Lk-1 ≠ ∅; k++ ) do begin
  Ck = apriori-gen(Lk-1); //New candidates
  forall transactions t ∈ D do begin
    Ct = subset(Ck, t);
    //Candidates contained in t
    forall candidates c ∈ Ct do
      c.count++;
    end
    Lk = { c ∈ Ck | c.count ≥ minsup }
  end
  Answer = ∪ k Lk;

```

The *apriori-gen* function takes as argument L_{k-1} , the set of all large $(k-1)$ -itemsets.

It returns a superset of the set of all large k -itemsets and is described in [1].

3.2 The FP-growth Algorithm

As shown in [2], the main bottleneck of the Apriori-like methods is at the *candidate set generation and test*. This problem was dealt with by introducing a novel, compact data structure, called *frequent pattern tree*, or **FP-tree** then based on this structure an FP-tree-based pattern fragment growth method was developed, **FP-growth**.

This method doesn't require candidate generation, but stores in an efficient novel structure, an **FP-tree** (a Frequent Pattern tree, a version of a **prefix tree**), the transaction database. It scans the database once to find frequent items. Frequent items F are then sorted in descending support count and kept in a list L . Another

scan of the databases is then performed, and for each transaction: infrequent items are suppressed and the remaining items are sorted in L -order and inserted in the **FP-tree**.

Definition 1 (FP-tree) A frequent pattern tree is a tree structure defined below.

1. It consists of one root labeled as "root", a set of *item prefix sub-trees* as the children of the root, and a *frequent-item header table*.
2. Each node in the *item prefix sub-tree* consists of three fields: *item-name*, *count*, and *node-link*, where *item-name* registers which item this node represents, *count* registers the number of transactions represented by the portion of the path reaching this node, and *node-link* links to the next node in the **FP-tree** carrying the same *item-name*, or null if there is none.
3. Each entry in the *frequent-item header table* consists of two fields, (1) *item-name* and (2) *head of node-link*, which points to the first node in the **FP-tree** carrying the *item-name*.

The actual algorithm, according also to [2] is:

Algorithm 1 (FP-tree construction)

Input: A transactional database DB and a minimum support threshold ξ .

Output: Its frequent pattern tree, **FP-tree**

Method: The **FP-tree** is constructed in the following steps:

1. Scan the transaction database DB once. Collect the set of frequent items F and their supports. Sort F in support descending order as L , the list of frequent items.
2. Create the root of an **FP-tree**, T , and label it as "root". For each transaction $Trans$ in DB do the following.
 - a. Select and sort the frequent items in $Trans$ according to the order of L . Let the sorted frequent item list in $Trans$ be $[p | P]$, where p is the first element and P is the remaining list. Call *insert_tree* $([p | P], T)$.
 - b. The function *insert_tree* $([p | P], T)$ is performed as follows. If T has a child N such that $N.item-name = p.item-name$, then increment N 's count by 1; else create a new node N , and let its count be 1, its parent link be linked to T , and its node-link be linked to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call *insert_tree* (P, N) recursively. The **FP-growth** [2] algorithm for mining frequent patterns with **FP-tree** by pattern fragment growth is:

Input: a *FP-tree* constructed with the above mentioned algorithm;

D – transaction database;

s – minimum support threshold.

Output: The complete set of frequent patterns.

Method:

call *FP-growth* $(FP-tree, null)$.

Procedure *FP-growth* $(Tree, A)$

```

{
  if Tree contains a single path P
  then for each combination (denoted as B) of the
  nodes in the path P do
    generate pattern  $B \cup A$  with support=minimum
    support of nodes in B
  else for each ai in the header of the Tree do
    {
      generate pattern  $B = ai \cup A$  with support =
      ai.support;
      construct B's conditional pattern base and B's
      conditional FP-tree
      TreeB;
      if TreeB  $\neq \emptyset$ 
      then call FP-growth(TreeB, B)
    }
}

```

4. COMPARATIVE STUDY

The following data has been processed both by Apriori and FP growth algorithms and they were compared. Min-support 50% and Min-confidence 80%.

Table 1:

TID	DATE	ITEMS BOUGHT
T100	10/15/99	K,A,D,B
T200	10/15/99	D,A,C,E,B
T300	10/19/99	C,A,B,E
T400	10/22/99	B,A,D

Apriori algorithm

There are two processes to find out all the large itemsets from the database in Apriori algorithm. First the candidate itemsets are generated, then the database is scanned to check the actual support count of the corresponding itemsets. During the first scanning of the database the support count of each item is calculated and the large 1-itemsets are generated by pruning those itemsets whose supports are below the pre-defined threshold. In each pass only those candidate itemsets that include the same specified number of items are generated and checked. The candidate k -itemsets are generated after the $(k-1)$ th passes over the database by joining the frequent $k-1$ -itemsets. All the candidate k -itemsets are pruned by check their sub $(k-1)$ -itemsets, if any of its sub $(k-1)$ -itemsets is not in the list of frequent $(k-1)$ -itemsets, this k -itemsets candidate is pruned out because it has no hope to be frequent according the Apriori property. The Apriori property says that every sub $(k-1)$ -itemsets of the frequent k -itemsets must be frequent.

Table 2 :PASS 1

Candidate C1

Itemset	Count
A	4
B	4

C	2
D	3
E	2
K	1

Table 3: Frequent item set L1

Itemset	Count
A	4
B	4
C	2
D	3
E	2

Table 5: PASS II C2

ITEM	COUNT
{A,B}	4
{A,C}	2
{A,D}	3
{A,E}	2
{B,C}	2
{B,D}	3
{B,E}	2
{C,D}	1
{C,E}	2
{D,E}	1

Table 7 :L2

ITEM	COUNT
{A,B}	4
{A,C}	2
{A,D}	3
{A,E}	2
{B,C}	2
{B,D}	3
{B,E}	2
{C,E}	2

Table 4: PASS 3 C3

ITEM	COUNT
{A,B,C}	2
{A,B,D}	3
{A,B,E}	2
{A,C,D}	1
{A,C,E}	2
{A,D,E}	1
{B,C,D}	1
{B,C,E}	2
{B,D,E}	1

Table 6: L3

ITEM	COUNT
{A,B,D}	3
{A,B,E}	2
{A,C,E}	2
{B,C,E}	2

The last table L3 gives the frequent item set with minimum support 50%.

Confidence :

Suppose we take any one of the frequent item say {A,B,D}

The possible association rules are given by

$A \wedge B \Rightarrow D$ IS $\frac{3}{4}$ WHICH IS 75%

$A \wedge D \Rightarrow B$ IS $\frac{3}{3}$ WHICH IS 100%

$B \wedge D \Rightarrow A$ IS $\frac{3}{3}$ WHICH IS 100%

$D \Rightarrow A \wedge B$ IS $\frac{3}{3}$ WHICH IS 100%

$B \Rightarrow A \wedge D$ IS $\frac{3}{4}$ WHICH IS 75%

$A \Rightarrow B \wedge D$ IS $\frac{3}{4}$ WHICH IS 75%

Since the given level of confidence is 80% the bolded associations rules are stronger.

FP growth tree algorithm :

CONSTRUCTION OF FP TREE

The process of constructing the FP-Tree is as follows.

(1) The database is scanned for the first time, during this scanning the support count of each items are collected. As a result the frequent 1 -itemsets are generated as shown , this process is the same as in Apriori algorithm. Those frequent itemsets are sorted in a descending order of their supports. Also the head table of ordered frequent 1 -itemsets is created .

TABLE 8: ORDERED LIST L1

Itemset	Count
A	4
B	4
D	3
C	2
E	2

(2) Create the root node of the FP-Tree T with a label of Root. The database is scanned again to construct the FP-Tree with the head table, for each transaction the order of frequent items is resorted according to the head table. For example, the first transaction (K;A;D;B) is transformed to (A;B;D;K), since A

occurs more frequently than other items in the database. Let the items in the transaction be [p j P], where p is the first frequent item and P is the remaining items list, and call the function *Insertf*[p j P];

(3) The function *Insertf*[p j P]; works as follows. If T has a child N such that N.item-name=p.item-name then the count of N is increased by 1, else a new node N is created and N.item-name=p.item-name with a support count of 1.Its parent link be linked to T and its node link is linked to the node with the same item-name via a sub-link. This function *Insertf*P;Tg is called recursively until P becomes empty.

Let's take the insertion of first transaction to the FPTree as an example to illustrate the *insert* function and construction of FPTree we mentioned above. After reorder this transaction is (A; B;D;K), so p is A in this case, while P is (B;D;K). Then we call the function of

insert, first we search and determine the node A exists in the tree or not, it turns out A is a new node. According to the rules, a new node named A is created with a support count of 1. Since here T is Root, node A is linked to Root and call the insert function again. At this time p is B, P is (D;K), T is A.The result of the FPTree of the database is shown.

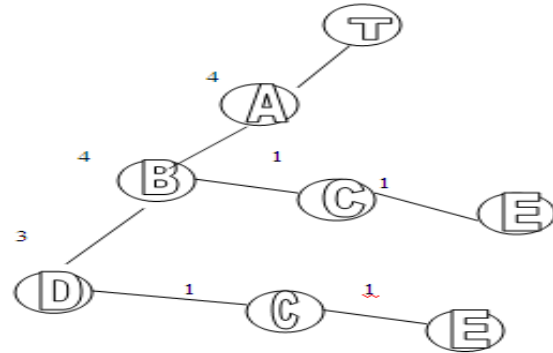


Figure 1: Mining FP tree

Table 9: mining of FP tree

Item	Conditional pattern	Condition FP tree	Frequency pattern generated
E	(ABDC:1),(ABC:1)	ABC:2	ABCE :2
D	(BA :3)	BA :3	BAD:3
C	ABD:1,AB:1	AB:2	ABD:2
B	A:4	A:4	AB:4

Mining of the FP tree is summarized in the above table.Let us first consider the last item in the ordered list that is K.K occurs in one branch of the tree that is ABDK: 1 .Considering K as suffix its corresponding path is ABD : 1. Since it 's support count does not satisfy the condition of minimum support it is not included in the conditional pattern.For the next data E it's two paths are ABDC: 1,ABC : 1 which generates conditional FP tree ABC :2 and derives frequent item pattern as ABCE:2.

In Apriori algorithm to find a frequent k-itemset it requires k passes over the database. Frequent itemsets of over 50-60 items are not feasible.Apriori needs to run over all $2^{60}-1$ frequent subsets. But Apriori avoids the effort wastage of counting the candidate itemsets that are known to be infrequent. The candidates are generated by joining among the frequent itemsets level-wisely, also candidate are pruned according the Apriori property. As a result the number of remaining candidate itemsets ready for further support checking becomes much smaller, which dramatically reduces the computation, I/O cost and memory requirement.

There are two bottlenecks of the Apriori algorithm. One is the complex candidate generation process that uses most of the time, space and memory. Another bottleneck is the multiple scan of the database.

Frequent pattern mining, is another milestone in the development of association rule mining, which breaks the two bottlenecks of the Apriori. The frequent itemsets are

generated with only two passes over the database and without any candidate generation process. FP-Tree was introduced by Han et al in [Han et al. 2000]. By avoiding the candidate generation process and less passes over the database, FP-Tree is an order of magnitude faster than the Apriori algorithm. The frequent patterns generation process includes two sub processes: constructing the FT-Tree, and generating frequent pat-terns from the FP-Tree.

The efficiency of FP-Tree algorithm account for three reasons. First the FP-Tree is a compressed representation of the original database because only those frequent items are used to construct the tree, other irrelevant information are pruned. Also by ordering the items according to their supports the overlapping parts appear only once with different support count. Secondly this algorithm only scans the database twice. The frequent patterns are generated by the FP-growth procedure, constructing the conditional FP-Tree which contain patterns with specified suffix patterns, frequent patterns can be easily generated as shown in above the example. Also the computation cost decreased dramatically. Thirdly, FP-Tree uses a divide and conquer method that considerably reduced the size of the subsequent conditional

FP-Tree, longer frequent patterns are generated by adding a suffix to the shorter frequent patterns. In [Han et al. 2000] [Han and Pei 2000] there are examples to illustrate all the detail of this mining process.

The number of disk access is critical in data mining task, because the I/O operation is more time consuming than a memory operation. It can be of advantage when we minimize the disk access. The level wise algorithms (Apriori) read the whole database k times for finding the k-frequent itemset. The algorithm will read the database as many times, as long the longest itemset is. If there is only one long frequent itemset, the mining time will increase rapidly.

The two-phase mining algorithms (FP tree) read the database only twice. In one case by the first database reading the support count of the 1-frequent itemsets are determined, and using this information reading the second time the database we can build the whole database in the memory with a clever compression of the database . The mining operation has to be accomplished only in the memory.

Every algorithm has his limitations, for FP-Tree it is difficult to be used in an interactive mining system. During the interactive mining process, users may change the threshold of support according to the rules. However for FP-Tree the changing of support may lead to repetition of the whole mining process. Another limitation is that FP-Tree is that it is not suitable for incremental mining. Since as time goes on databases keep changing, new datasets may be inserted into the database, those insertions may also lead to a repetition of the whole process if we employ FP-Tree algorithm. Regarding the I/O cost, it is a very essential parameter, how many times the algorithm accesses the database.

5. CONCLUSION

From the experimental data presented it can be concluded that the FP-growth algorithm behaves better than the Apriori algorithm. First of all, the FP-growth algorithm needs at most two scans of the database, while the number of database scans for the candidate generation algorithm (Apriori) increases with the dimension of the candidate itemsets. Also, the performance of the FP-growth algorithm is not influenced by the support factor, while the performance of the Apriori algorithm decreases with the support factor.

Thus, the candidate generating algorithms (derived from Apriori) behave well only for small databases (max. 50,000 transactions) with a large support factor (at least 30%). In other cases the algorithms without candidate generation FP-growth behave much better.

ACKNOWLEDGEMENT

The Author would like to thank the management of Botho college for their incomparable motivation and encouragement, especially Mr. Ramaraju-Dean (Research) and Mr. Sivakumar (Manager-NIIT)

REFERENCES

- [1] R. Agrawal, R. Srikant. "Fast algorithms for mining association rules in large databases". *Proc. of 20th Int'l conf. on VLDB*: 487-499, 1994.
- [2] J. Han, J. Pei, Y. Yin. "Mining Frequent Patterns without Candidate Generation". *Proc. of ACM-SIGMOD*, 2000.
- [3] C. Györfi, R. Györfi. "Mining Association Rules in Large Databases". *Proc. of Oradea EMES'02*: 45-50, Oradea, Romania, 2002.
- [4] R. Györfi, C. Györfi. "Architectures of Data Mining Systems". *Proc. Of Oradea EMES'02*: 141-146, Oradea, Romania, 2002.
- [5] C. Györfi, R. Györfi, S. Holban, M. Pater. "Mining Knowledge in Relational Databases". *Proc. of CONTI 2002, 5th International Conference on Technical Informatics*: 1-6, Timisoara, Romania, 2002.
- [6] M. H. Dunham. "Data Mining. Introductory and Advanced Topics". Prentice Hall, 2003, ISBN 0-13-088892-3.
- [7] U.M. Fayyad, et al.: "From Data Mining to Knowledge Discovery: An Overview", *Advances in Knowledge Discovery and Data Mining*: 1-34, AAAI Press/ MIT Press, 1996, ISBN 0-262-56097-6.
- [8] J. Han, M. Kamber, "Data Mining Concepts and Techniques", Morgan Kaufmann Publishers, San Francisco, USA, 2001, ISBN 1558604898.
- [9] L. Cristofor, "Mining Rules in Single-Table and Multiple-Table Databases", PhD thesis, University of Massachusetts, 2002.

- [10] Ashoka Savasere, Edward Omiencinski, and Shamkant B. Navathe. "An Efficient Algorithm for Mining Association Rules in Large Databases." In Proceedings of the 21st International Conference on Very Large Databases, pag. 432 - 444, 1995.
- [11] Cornelia Györödi, Robert Györödi, T. Cofeey & S. Holban – "Mining association rules using Dynamic FP-trees" – in Proceedings of The Irish Signal and Systems Conference, University of Limerick, Limerick, Ireland, 30th June- 2nd July 2003, ISBN 0-9542973-1-8, pag. 76-82



Vivekananth is working as lecturer in Botho College, Botswana. He is a research scholar and currently perusing his research in cloud Storage security based on trusted third party auditing. Previously he was working as lecturer/HOD in St Joseph college of Engineering and Technology, Dar-Es-

Salaam Tanzania Having worked as Infrastructure Management-Associate, His Technical exposure includes System and Storage Administration. He has Technical certifications in IBM TSM, IBM AIX, and Oracle 11g Administration and Microsoft .NET framework 4 using C#. His area of Research is Cloud Computing, Grid Computing and Peer to Peer Networks. His Research publications includes 8 International journals, 6 International conferences (IEEE, ACM, Springer, Elsevier, IET), 6 National Level Conferences/Symposiums in India. He is Member of BCS(British Computer Society),UK.