Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 47 (2015) 84 – 91

Procedia Computer Science

An Improved Method for Counting Frequent Itemsets Using Bloom Filter

J. Shana[1], T. Venkatachalam[2]

Department of Computer Applications, Coimbatore Institute of Technology, Coimbatore- 641014, India
Department of Physics, Coimbatore Institute of Technology, Coimbatore- 641014, India

Abstract

Frequent itemsets play an essential role in many data mining tasks that try to find interesting patterns from databases. Frequent itemset mining is one of the time consuming tasks in data mining. It is one of the prime steps in association rule mining. Many versions of frequent itemset mining algorithms have been proposed by many researchers that aim at reducing the time and space complexities. In this work we attempt to use bloom filter, a probabilistic data structure to determine the frequent itemsets. Bloom filter uses hashing to store data. Experiments on real datasets have shown that there is considerable advantage in terms of memory and performance in this technique compared to other hash based techniques.

Peer-review under responsibility of organizing committee of the Graph Algorithms, High Performance Implementations and Applications (ICGHIA2014)

Keywords: Bloom Filter; Frequent Itemset Mining; Assoication Rule Mining; Data Mining

Introduction

The goal of frequent-itemset mining is to discover sets of items that frequently co-occur in the data. The problem is non-trivial because datasets can be very large, consist of many distinct items, and contain interesting itemsets of high cardinality. Frequent-itemset mining is a key component of many data mining tasks and has applications in areas such as bioinformatics [11], market basket analysis [2], and web usage mining [12]. The need for finding frequent itemsets in ever-growing datasets has driven a wealth of research

*J.Shana:Email.:shana.cta@gmail.com;.

Peer-review under responsibility of organizing committee of the Graph Algorithms, High Performance Implementations and Applications (ICGHIA2014)
doi:10.1016/j.procs.2015.03.186

in efficient algorithms and data structures [1, 14]. Existing approaches can be classified into three major categories. First, bottom-up algorithms such as the well-known Apriori algorithm [1, 14]‖repeatedly scan the database to build itemsets of increasing cardinality. They exploit monotonic properties between frequent itemsets of different cardinalities and are simple to implement, but they suffer from a large number of expensive database scans as well as costly generation and storage of candidate itemsets. top-down algorithms proceed the other way around. The largest frequent itemset is built first and itemsets of smaller cardinality are constructed afterwards, again using repeated scans over the database. Finally, prefix-tree algorithms operate in two phases. In the first phase, the database is transformed into a prefix tree designed for efficient mining. The second phase extracts the frequent itemsets from this tree without further database access. Algorithms of this class require only a fixed number of database scans, but may require large amounts of memory. There is no best algorithm for frequent itemset mining in general, but there is lot of variations from the basic algorithm proposed by many that aims to improve its efficiency. I n this paper we have considered only those variations of Apriori that uses hashing, for comparing the results. The paper has the following sections. Section II explains the basics of frequent itemset mining and Section III gives the literature survey. Section IV describes the two existing methods and section V the proposed Bloom filter based algorithm. Section VI and section VII gives the experimental results and conclusion respectively.

Background

Formally let I be the set of items. A transaction over I is a couple T = (tid, I) where tid is the transaction identifier and I is the set of items from I.A database D over I is a set of transactions over I such that each transaction has a unique identifier. A transaction T = (tid, I) is said to support a set X, if $X \subset I$. The cover of a set X in D consists of the set of transaction identifiers of transactions in D that support X. The support of a set X in D is the number of transactions in the cover of X in D. The frequency of a set X in D is the probability that X occurs in a transaction, or in other words, the support of X divided by the total number of transactions in the database. A set is called frequent if its support is no less than a given minimal support threshold min_sup with $0 > min\_sup > |D|$. The goal of frequent itemset mining is to find all itemsets that are frequent. The algorithm given by R. Agrawal and R. Srikant called Apriori is a seminal algorithm, which uses an iterative approach known as a level-wise search, where k-itemsets are used to explore (k+1) itemsets. For example, consider the database shown in Table 1 over the set of items.

Table 1. Sample Dataset

Tid	Set of Items
T1	{i1,i2,i4}
T2	{i1,i2}
T3	{i2,i3,i4}
T4	{i1,i2,i3}

The following table shows all frequent sets in D with respect to a minimal support threshold equal to 2, their cover in D, plus their support and frequency.

Table 2. Support count for the itemsets

Set	Cover	Support	Frequency (%)
{ }	{T1,T2,T3,T4}	4	100
{i1}	{T1,T2}	3	75
{i2}	{T1,T2,T3}	4	100
{i1,i2}	{T1,T2}	3	75

The task of discovering all frequent sets is quite challenging. The search space is exponential in the number of items occurring in the database and the targeted databases tend to be massive, containing millions of transactions. Both these characteristics make it a worthwhile effort to seek the most efficient techniques to

solve this task.

Literature Survey

With the introduction of the frequent set mining problem, the first algorithm denoted as AIS was proposed. This algorithm was improved by R. Agrawal and R. Srikant and named as Apriori. It is a seminal algorithm, which uses an iterative approach known as a level-wise search, where k-itemsets are used to explore (k+1)-itemsets. Direct Hashing and Pruning algorithm generates fast mining of association rules [3]. It utilizes a hash method for candidate itemset generation. In order to effectively reduce transaction database size, the proposed algorithm uses only a small number of candidate itemset. At the earlier stage of iterations itself database size is reduced, and therefore the computational cost is reduced. Perfect Hashing and Pruning algorithm [18] generates the frequent itemset of a transaction database. This algorithm has some of the features of Direct Hashing and Pruning algorithm. It employs hashing to keep the actual count of occurrence of each candidate itemset of the database. It also prunes the transactions which do not contain any frequent items, and trims non-frequent items from the transactions at each step in order to improve the accuracy of rules. Inverted Hashing and Pruning algorithm [5] identifies the candidate and frequent itemsets with the help of hash tables. The database is stored with three fields such as transaction identifier and the transaction with its corresponding hash value. Next TID Hash Table is built with items and its entries. Each entry has a number showing the number of items hashed to it. Instead of scanning the database, count of items support in hash table entries are added. Then items which are not satisfying minimum support are removed. Double hashing technique [5] stores the itemset in buckets based on hash function. At the time of hash collision, double hashing technique is used to resolve them. After hashing the candidate 1-itemset, the frequent 1-itemset is calculated by using minimum support, and then candidate 2-itemset is hashed and so on. Maximal frequent items are directly calculated from frequent items itself. Finally from all the frequent items, association rules are generated. Hash Based Maximal Frequent Itemsets -Linear Probing method avoids unnecessary database scan. Database is stored in vertical format and based on hash function items are hashed into hash table [10]. To avoid hash collisions Linear Probing sequence is handled. For first level of items a linked list is created for each item. Support count of each item is stored in first node of the linked list and other nodes stored number of transactions related to corresponding item. Instead of scanning the database, support count is taken from hash table and the next level of items is hashed. Maximal frequent items are directly calculated from frequent items itself.

Existing Work

1.1. Apriori Algorithm

It uses the Apriori property to reduce the search space: All nonempty subsets of a frequent itemset must also be frequent [1].

- P(I)<min_sup => I is not frequent
- P(I+A)<min_sup => I+A is not frequent either
- Antimonotone property – if a set cannot pass a test, all of its supersets will fail the same test as well. In many cases the Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain. However it can suffer from two nontrivial costs:
- It may need to generate a huge number of candidate sets. For example if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.
- It may need to repeatedly scan the database and check a large set of candidates by pattern matching.

Although, Apriori presented by Agrawal et al. [1] is very effective method for enumerating frequent itemsets of sparse datasets on large support threshold, but the basic algorithm of Apriori encounters some difficulties and takes large processing time on low support threshold.
Apriori encounters difficulty in mining long pattern, especially for dense datasets. For example, to find a frequent itemsets of X = {1…200} items. Apriori has to generate-and-test all 2^{200} candidates. Apriori algorithm is considered to be unsuitable for handling frequency counting, which is considered to be most expensive task in frequent itemsets mining. Since Apriori is a level-wisecandidate-generate-and-test algorithm, therefore it has to scan the dataset *200* times to find a frequent itemsets X = X1… X200. Even

for datasets with large items, determining *k*-frequent itemsets by repeated scanning the dataset with pattern matching takes a long processing time. Apriori and its variants enumerate all frequent itemsets by repeatedly scanning the dataset and checking the frequency of candidate frequent *k* itemsets by pattern matching. This whole process is costly especially if the dataset is dense and has long patterns, or a low minimum support threshold is given.

4.1 Hash-based Quadratic Probing

The existing methodology, Hash Based Frequent Itemsets-Quadratic Probing (HBFI-QP) algorithm uses the hash table structure with linked list to catch on the frequent itemset . HBFI-QP technique first converts the initial transactional database into vertical format (Itemset, Tidset). Itemset is the number of items in database and Tidset represents the list of transactions in which a particular item occurs [20]. The hash table is constructed based on the size n and is determined by the equation (1).

$$n = 2(\text{Number of items}) + 1 \quad \ldots (1)$$

First level itemsets in the vertical format are hashed based on the hash function given in the equation (2) where size n must be prime.

$$H(k) = (\text{order of item } k) \bmod n \quad \ldots (2)$$

If collisions take place, Quadratic Probing (QP) technique is employed to avoid it. QP function is given by the equation (3). H (k) is the original hash value and $1 \leq \text{probe}[i] \leq \text{table length}-1$.

$$H(k, i) = (H(k) + i2) \bmod n \quad \ldots (3)$$

After placing all the itemsets in hash table, linked list is created for each stored item in the table. Linked list consists of nodes with support count and Tidset for every item. Instead of scanning the database, frequent 1-itemset is directly captured from the hash table based on the minimum support threshold. The second level items are hashed based on the hash function given by the equation (4).

$$H(k) = ((\text{order of item X}) * 10 + \text{order of item Y}) \bmod n \quad \ldots (4)$$

Hashing with Quadratic Probing technique places all items without any collision but the probing sequence volume is high. Because of the lengthy probing sequence, it takes more time to hash the collided itemsets. Secondary clustering occurs while using QP technique. For the same initial hash value, same sequences of numbers are repeated again, this is termed as secondary clustering. QP technique mainly depends on load factor (ratio of the number of stored entries and the size of the table's array of buckets). Quadratic Probing happens when the load factor is less than or equal to 0.5. Even though hash table contains free space, the size of hashed itemset is not more than half the hash table's size and so remaining memory space is idle.

The Bloom Filter (BF)

A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. Bloom filters allow false positives but the space savings often outweigh this drawback when the probability of an error is made sufficiently low. There has been a lot of variation in the bloom filter giving rise to a number of different varieties. Bloom filters have been used widely in computing applications. Broder and Mitzenmacher have coined the Bloom filter principle [9] as: "Whenever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated."A Bloom filter is an array of m bits for representing a set S = {x1, x2 , . . .

, x_n} of n elements. Initially all the bits in the filter are set to zero. The key idea is to use k hash functions, hi(x), 1≤ I ≤ k to map items x ∈ S to random numbers uniform in the range 1, . . .m. The hash functions are assumed to be uniform. The MD5 hash algorithm is a popular choice for the hash functions. An element x ∈ S is inserted into the filter by setting the bits $h_i(x)$ to one for 1 ≤ i ≤ k. Conversely, y is assumed a member of S if the bits $h_i(y)$ are set, and guaranteed not to be a member if any bit $h_i(y)$ is not set. Fig 1 shows the insertion of elements x, y, z into a bloom filter with m=32 and three hash functions (k=3).

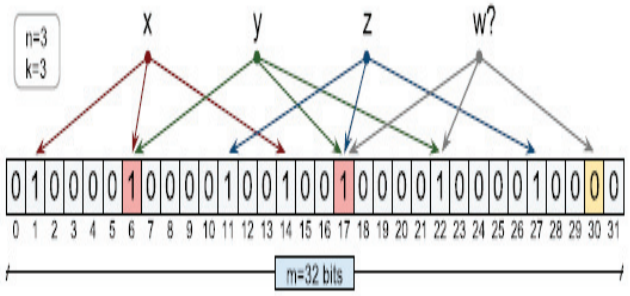

Fig 1 Insertion in Bloom Filter

One noteworthy property of Bloom filters is that the false positive performance depends only on the bit-per-element ratio (m/n) and not on the form or size of the hashed elements. As long as the size of the elements can be bounded, hashing time can be assumed to be a constant factor. Considering the trend in computational power versus memory access time, the practical bottleneck is the amount of (slow) memory accesses rather than the hash computation time.

5.1 False Positive Probability

The Bloom filter has a trade-off between memory usage (i.e., the number of bits used) and the false positive rate. When storing 'n' k-itemset in a Bloom filter of m bits, and using d hash functions, the false positive rate is approximately $(1 - e^{-dn/m})^d$. Given n and m, the optimal number of hash functions that minimizes the false positive ratio is $d \approx (m/n)\ln 2$. Fig 2 shows the effect of false positive probability for different values of n and m. In practice we may have rough idea in advance about n, the number of k-itemset, and we can select m as a fixed multiple of n. For example using m = 8n (which corresponds to storing one byte per k-itemset), and d = 5 gives a false positive ratio of 2.16%.

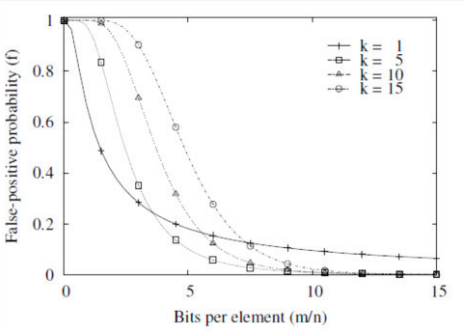

Fig 2 False Positive probability

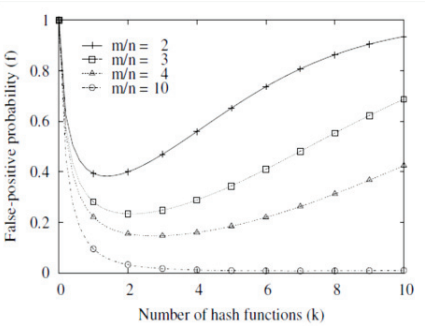

Fig 3. False positive rate (FPR) with the increase in the number of bits

2. Storing and Counting K-Itemset using BF

In order to insert a k-itemset x into the Bloom filter, we set all of the d corresponding locations in B to be 1; that is, we set $B[h_i(x)] = 1$ for i = 1, ..., d. Then, to determine whether a k-itemset y has been inserted, we simply check whether each of the corresponding hash positions is 1, i.e., whether $B[h_i(y)]$ are all set to 1 for i = 1,..., d. If this is the case, then we infer that y has probably been seen before. By construction, this procedure correctly identifies every k-itemset that is present more than once in the data. If the desired minimum support count is c we use an array of m $\lceil \log_2(c) \rceil$-bit counters. The counting Bloom filter was introduced by [19] to allow for deletions, but here we use the counts directly. To check if a k-itemset should be inserted into the hash table T we look to see if all of B[hi(x)] are equal to c -1. Otherwise we insert it into the Bloom filter. When inserting a k-itemset x, we set

$$B[hi(x)] \leftarrow \min\{B[hi(x)] + 1, c - 1\} \quad \ldots. (5)$$

for i = 1, ..., d. Note that for a k-itemset x, min{B[hi(x)]|i= 1, ..., d} gives an upper bound on the number of occurrences of x seen so far.

Table 3. Algorithm : To store and count the k-itemset

```
1.  B <= empty Bloom Filter of size M with m-bit counter
2.  T <= hash table
3.  c <= min_support
4.  for all reads s do
5.    for all k-itemset x in s do
6.        If x ∈ B then
7.          If count = c-1 then
8.              T[x] <= T[x]+1
9.          else
10.             add x to B
11.       else
12.         add x to B
13.  for all x in s
14.    if T[x] < = c then
15.       remove x from T
```

Experimental Results

Bloom Filter based algorithm along with Aprori and Hash Based-QP is implemented in Java. All the experiments are performed on a 2.27 GHz Intel Core i3 PC with 4GB memory running under Windows 7 with the same two real datasets used previously in the evaluation of frequent itemsets [5]. The characteristics of the two real datasets are shown in Table 4.

Table 4. Dataset Characteristics

Datasets	Items	Avg.Length	Transactions
Pumsb	7117	50	49,046
Mushroom	120	23	8124

The fig 4 and fig 5 shows the time taken for generating the frequent itemsets using varying support counts for the real datasets.

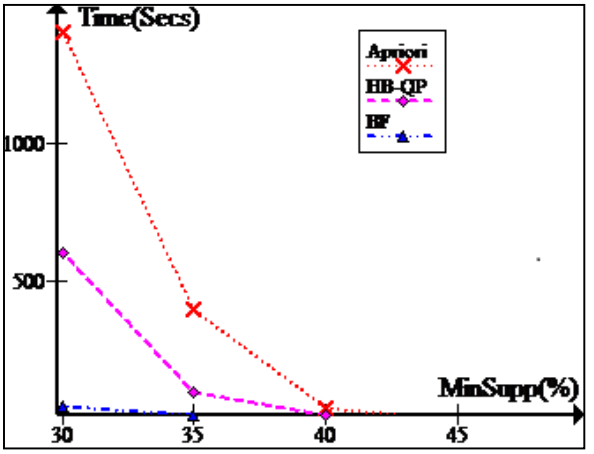

Fig 4 Total computation time taken for Pumsb dataset

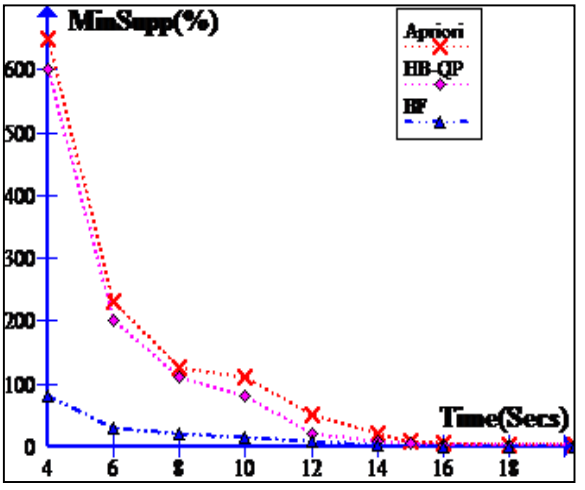

Fig 5 Total computation time for mushroom dataset

The experimental results show a sharp increase in the computational time for the apriori and BB-QP algorithms whereas it decreases slowly in the case of BF based method. But the overall time is effectively reduced by the proposed algorithm.

3. Conclusion

In this paper we have used the bloom filters to count the frequent itemsets. The bloom filter is used both for storing and counting the itemsets and it is observed from the experimental results that using bloom filters will reduce the time taken to generate frequent itemsets. The rate of false positive can be significantly

reduced by choosing the appropriate size m for the filter and k for the counter. This data structure can be effectively implemented parallel to further improve the overall performance.

References

1. Agrawal R, Imielinski T., Swami A. Mining association rules between sets of items in large databases, *ACM SIGMOD International Conference on Management of Data*, 1993.
2. Jiawei Han, Micheline Kamber. *Data Mining – Concepts and Techniques*, Morgan Kaufmann, second edition, 2006.
3. Ebrahim Ansari Chelche, Sadreddini M H., Dastghaybifard G H. Using Candidate Hashing and Transaction Trimming in Distributed Frequent Itemset Mining, *World Applied Sciences Journal*, 2010; Vol.9, No.12, 1353-1358.
4. Hassan Najadat, Amani Shatwani , Ghadeer Objedat. A New Perfect Hashing and Pruning Algorithm for Mining Association Rule, *IBIMA* , 2011.
5. John D.Holt and Soon M.Chung. Mining Association Rules using Inverted Hashing and Pruning, *ELSEVIER Information Processing letters*, 2002;211-220.
6. Lin D.I., Kedem Z.M. Pincer Search: A New Algorithm for Discovering the Maximum Frequent Set, *Lecture Notes in ComputerScience*, 1998; Vol. 1377, 105–119.
7. Shenoy P., Haritsa J R, Sundarshan S, Bhalotia G, Bawa M, and Shah D. Turbo-charging vertical mining of large databases in *SIGMOD*, 2000.
8. Krishnamurthy P, Buhler J, Chamberlain R, Franklin M, Gyang K, Jacob A, Lancaster J. Biosequence Similarity Search on the Mercury System, *The Journal of VLSI Signal Processing*, 2007.
9. Broder A, Mitzenmacher M: Network Applications of Bloom Filters: A Survey, *Internet Mathematics,* 2004; Vol. 1, No.4, 485-509.
10. Zubair Rahman A M J., Balasubramanie P ,.Venkata Krishna P. A Hash Based Mining Algorithm for Maximal Frequent Itemsets using Linear Probing, *Info Comp Journal of Computer Science*, 2009; Vol.8, No.1, 14-19.
11. Fan L, Cao P, Almeida J, Broder A Z. Summary cache: a scalable wide-area web cache sharing protocol, *IEEE/ACM Trans Netw* , 2000; 8:281-293.
12. Wang J T L, Zaki M J., Toivonen H., and Shasha D. *Data Mining in Bioinformatics*, Springer, 2005.
13. Punin J R., Krishnamoorthy M S. , Zaki M J. Web usage mining - languages and algorithms, *In Studies in Classification, Data Analysis, and Knowledge Organization*, Springer-Verlag, 2001;88-112.
14. Agrawal R., Srikant R. Fast algorithms for mining association rules, *In VLDB,* 1994;487-499.
15. Goethals B. and Zaki M J. Advances in frequent itemset mining implementations, *Report on fimi'03.SIGKDD Explorations*, 2004; 6:109-117.
16. Ostlin A., Pagh R. Uniform hashing in constant time and linear space, *in STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 2003; 622–628.
17. Zhiyong Z., Hui Y., Tao F. Using HMT and HASH -TREE to Optimize Apriori Algorithm, *International Conference on Business Computing and Global Informatization*, 2011.
18. Yang D L., Pan C. T., Chung Y. An Efficient Hash - Based Method for Discovering the Maximal Frequent Set, *Computer Software and Applications Conference*, 2011; 511-516.