Vertical-format Based Frequent Pattern Mining - A Hybrid Approach

Neha Dwivedi, Srinivasa Rao Satti
Seoul National University
South Korea
nehachugh1029@gmail.com
ssrao10@gmail.com

ABSTRACT: *A Frequent pattern is a pattern (a set of items) that occurs frequently in a data set. The problem of finding such inherent regularities or associations (patterns) in a dataset is called Frequent pattern mining. This problem has been widely studied in the literature because of its numerous applications to a variety of data mining problems such as clustering and classification.*

In this paper, a compact hybrid data structure HybridDS and a corresponding time efficient vertical format mining algorithm HybridDSItr for sparse datasets has been proposed. Some new optimization techniques have been applied to reduce intermediate candidate generation and thus minimizing time and search space. Experimental studies have been performed to compare the new algorithm with FP-Growth algorithm (Trie based) and Eclat algorithm (Vertical format based). The experimental results confirm following observations for sparse datasets. The algorithm exhibits better performance in terms of time as compared to both Eclat and FP-Growth algorithms. It exhibits better performance in terms of memory as compared to FP-Growth and similar or better performance than Eclat algorithm. This research can be applied to improve memory and time efficiency of existing vertical format based mining algorithms.

Keywords: Frequent Pattern Mining, FPM, HybridDS, HybridDSItr, Eclat, FP-Growth

Received: 1 July 2015, Revised 9 August 2015, Accepted 12 August 2015

1. Introduction

Frequent pattern mining (FPM) plays an essential role in mining associations and correlations and other and other data mining tasks such as data indexing, clustering and classification. It also has numerous applications in diverse domains such as spatio-temporal data, software bug detection, and biological data. However, the task of discovering all frequent associations in very large databases is quite challenging. The search space is exponential in the number of database attributes. With millions of database objects, I/O minimization and search space minimization becomes paramount. Many researchers have proposed solutions for frequent pattern mining. *Apriori algorithm* [5] was introduced in 1994, but the algorithm is slow because of the excessive I/O operations due to repeated access of dataset to find frequent patterns and to generate candidates. The newer algorithms mostly require only one or two database scans to find the frequent patterns. The most common algorithms currently being used and researched are either *Trie* based or *Vertical Mining* based.

Among the trie-based algorithms, the FP-Growth algorithm [4] is the most popular one. It requires two passes of the database – the first pass to eliminate infrequent items, and the second pass to build the trie data structure. However, for very sparse datasets, sometimes the trie data structure does not fit in main memory due to heavy use of pointers and its large structure size.

Algorithms based on vertical format mining support fast frequency counting via intersection operations on transactionvids (tids) and automatic pruning of irrelevant data. One of the most famous vertical format mining algorithm is Eclatv[11]. Most of the existing vertical format based algorithms are modifications of Eclat. It also requires two passes,vsimilar to FP-Growth – the first pass to eliminate infrequent items, and the second, to load the transactions in vertical format. For fast intersection operations, the transaction ids per item are represented by bitsets. For sparse datasets, the size of the bitset structure becomes huge when compared to the actual file size. Also, as with most of the vertical format based algorithms, a large number of intermediate candidates are generated. The size of the bitsets gets amplified for these intermediate candidates and the search space thus becomes very large in size.

In this paper, a memory-efficient vertical format mining algorithm based on a hybrid data structure has been proposed. The idea is to minimize the size of the basic data structure and also reduce the number of candidates generated which will thus lessen the intermediate memory usage during candidate generation. The new algorithm has been compared with the FP-Growth algorithm and the Eclat algorithm in terms of time and memory. The pros and cons in comparison to the other two algorithms have also been discussed. A preliminary version of this paper has appeared as [15].

The rest of the paper is organized as follows. Section 2 defines the FPM problem and related notations. FP-Growth algorithm is explained briefly in Section 3, while Section 4 described the Eclat algorithm. In Section 5, a study of array and bitset intersection with respect to memory and time has been explained. Section 6 explains the new hybrid data structure based solution. Section 7 shows the experimental results of the new algorithm. Finally, Section 8 ends with some concluding remarks.

Transaction id	Items
1	chocolate, coffee, chips
2	coffee, beer, chips, juice
3	beer, chips
4	coffee, beer

Table 1. Small market dataset

2. Frequent Pattern Mining

Frequent Pattern Mining is defined as a mining problem to find all the frequent patterns from a given transaction database. This problem was introduced by Agrawal et al. [5] in 1993. The input consists of the list of transactions D and a minimum support value min-support. If pk is a subset of items, the number of transactions that contain p_k is its frequency or support. If the frequency of p_k is at least min-support, then pk is a frequent pattern. In other words, if the number of transactions that contain pk is at least min-support, then p_k is considered as a frequent pattern. In frequent pattern mining, our primary objective is to find all the patterns that are frequent and also their corresponding frequencies in the input dataset.

Here is an example. For a min-support = 50% and input dataset provided in Table 1, {coffee}, {beer}, {chips}, {coffee, beer} and {beer, chips} are frequent patterns, as the frequency of each of these patterns is at least 2 (50%). Following are the definitions of terms which are commonly used in the paper.

- Pattern or itemset: A set of items.

- Frequency or support: The number of occurrences of an itemset in a dataset.

- Min-support: The minimum support that an itemset should have to be frequent.

• **Frequent pattern:** An itemset whose frequency is at least min-support.

• **K-itemset:** an itemset containing k items

• **Candidate:** Any itemset that might be a frequent pattern

• **Sparse datasets:** Datasets in which transactions have very little commonality between them.

• **Dense datasets:** Datasets in which transactions have very high commonality between them.

• **Set:** A collection of elements of same type.

• **Bitset:** A set of numeric elements of fixed size where each element is represented by 0 or 1, 1 representing presence and 0 representing absence of the element respectively.

• **Array:** A fixed-size sequential collection of elements of the same type.

3. FP-Growth Algorithm

The FP-Growth algorithm [4] finds frequent patterns efficiently using a trie-based data structure called FP-tree. FP-tree is a prefix-tree that stores transactions in a dataset compactly. In FP-tree, each node represents one item and stores item-identifier and its frequency. FP-tree is a compressed version of a dataset because FP-tree stores the dataset by storing similar transactions (i.e., transactions with the same prefix) in the same path, and transactions containing specific item can be accessed directly using node-links. Once the FP-Tree is generated, mining is very fast. For each item, a conditional FP-tree is generated. A conditional FP-tree is an FP-tree of only those transactions in which the item is present. As all nodes in the tree are interconnected by pointers, the candidates patterns are generated quickly. However, memory usage of the tree-based data structure (FP-tree) might become very high. In every node of the tree, other than the item identifier, its frequency (in that path) and three references (parent pointer, child pointer, node-link). For a large sparse dataset, if number of nodes become very high, it might become too large in size to fit in memory.

4. Eclat Algorithm

A number of vertical format mining algorithms have been proposed for finding frequent patterns. In a vertical database each item is associated with its corresponding tidset, the set of transactions (or tids) where it appears. Mining algorithms using the vertical format have been shown to be very effective and usually outperform horizontal approaches. This advantage stems from the fact that frequent patterns can be counted via tidset intersections, instead of using complex internal data structures (candidate generation and counting happens in a single step). Tidsets offer natural pruning of irrelevant transactions as a result of an intersection (tids not relevant drop out).

Eclat [11] is one of the earliest vertical format based mining algorithms. Most of the algorithms in the same category are extensions of Eclat. It supports fast frequency counting via intersection operations on transaction ids (tids) and automatic pruning of irrelevant data. It requires two database passes. In the first pass, the infrequent items are removed. In the second pass, first a tidset per frequent item (implemented as bitsets for efficiency) is created. Then the tidsets are intersected to find candidate patterns. A search tree is built to mine and find the frequent itemsets in a depth first fashion with each node containing candidate pattern and corresponding tidset (implemented as bitset for efficiency).

Only if a pattern is frequent, the subtree is extended further by intersecting with remaining frequent item tidsets. Despite the many advantages of the vertical format algorithms such as *Eclat*, when the tidset cardinality gets very large (e.g., for high frequency items) the methods start to suffer, since the intersection time starts to become inordinately high. Furthermore, the size of intermediate tidsets generated for frequent patterns can also become very large, requiring data compression and writing of temporary results to disk. In dense datasets, which are characterized by high item frequency and many patterns, the vertical approaches may quickly lose their advantages.

5 . Intersection - Array vs Bitset Representation

This section describes the motivation for our hybrid data structure, and the justification for the parameter choices that we make in our implementations.

Data Structure	AB	CD	EF
Array of integers	<= 300 integers	<= 6000 integers	<= 30,000 integers
Bitset	6000 integers	6000 integers	6000 integers

Table 2. Memory usage : Array vs Bitset

Given a set of n integers from the universe $\{1, 2, \ldots, m\}$, one can represent it either as an array of integers with length n, or as a bit vector (referred henceforth as bitset) of length m (containing n ones in it). Let w be the size of integer in bits. In this section, we compare how intersection of array of integers perform as compared to intersection of bitsets. Our findings here have been used in the development of the new algorithm.

5.1 Memory

First let us compare the memory used when we intersect two arrays of integers as compared to memory used when we intersect two bitsets for the same set of integers. For example, suppose we have possible range of 1 - 64000, Set A has 100 elements, Set B has 100 elements, Set C has 1000 elements, Set D has 1000 elements, Set E has 10,000 elements and Set F has 10,000 elements.

Suppose we want to intersect Sets A and B, Sets C and D and Sets E and F. Table 2 represents the amount of memory that will be used for these intersections when we represent the sets as array of integers and bitset respectively. Here we have assumed that an integer takes 32 bits. Also the memory calculated includes two source data structures and the resultant data structure.

We can see here that representing data as array of integers will be beneficial when the set cardinality is $< m / w$ or $(0.03125) * m$, where $w = 32$. When the size of the set is large, then using a bitset is more beneficial for memory usage. Note that the above comparison is purely on a theoretical basis. Actually memory usage also depends on the machine and compiler.

5.2 Time

In this section we will compare the time used for intersection of given set of integers when represented as array of integers as compared to when represented as bitsets. As we cannot compare this theoretically as many varied machine characteristics come into play, we have compared this experimentally. We have created a program to compare the time taken by intersection of two arrays of integers to intersection of two bitsets.

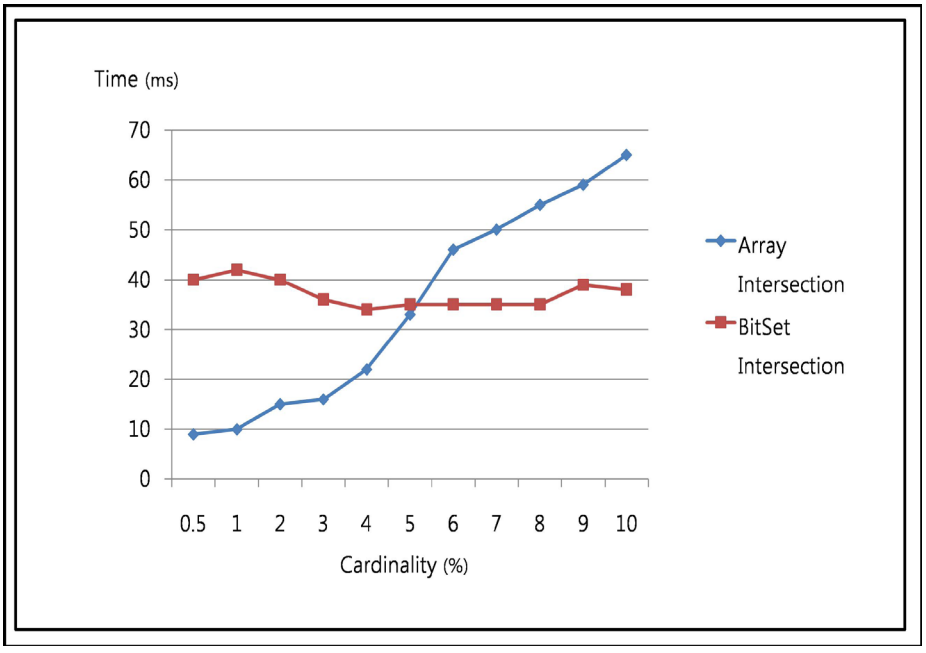


Figure 1. Array intersection vs Bitset intersection

We created two bitsets of size 100,000,000. For a cardinality varying from 500,000 (0.5 %) to 10,000,000 (10 %) with random distribution, we compared the performance of bitset intersection vs array of integers intersection where the size of array of integers is equal to bitset cardinality and values in the arrays correspond to the position of 1s in the bitsets.

Figure 1 shows how array intersection performs as compared to bitset intersection. Time has been measured in milliseconds for both the operations.

We can see that for low cardinality (< 0.05), array of integers performs better. Bitset intersection's performance remains similar even for higher cardinality, but array of integers intersection performance deteriorates.

From the above results for time and memory, we deduced that for intersection of transaction sets with cardinality $< m / 32$, using an array of integers data structure to represent the transaction ids instead of bitset might be more beneficial.

6. HybridDSItr

Most of the vertical data format based algorithms have been implemented recursively using depth-first approach. If we can develop an iterative solution, we can allocate memory in a more controlled fashion. Also, we could use the same algorithm to prune infrequent candidates earlier in the loop preventing some of the time-consuming intersection operations of the large sets of transaction ids. This has been explained later. The algorithm first builds either a pure bitset only data structure or a hybrid data structure which is a combination of bitsets and array of integers. We first describe how we build the bitset only data structure, and mention its pros and cons. We will then describe the hybrid data structure and how it helps in saving both time and memory.

6.1 Bitset Data Structure

In this approach we have only one data structure: bitset table. The bitset table stores all transactions with frequent items in bitset representation.

To generate bitset table, two scans of the dataset are needed, as in the case of *FP-Growth* and *Eclat* algorithms. Following are the steps to build bitset table.

1. Scan the dataset once to get frequency of each item. Remove all the items whose frequency is lower than minsupport.

Generate a list of frequent items by sorting remaining items in frequency ascending order.

2. Scan the dataset again and transform each transaction to vertical bitset representation.

The above procedure to build the bitset table is same as that for *Eclat*. Each transaction is transformed to vertical bitset representation for the high frequency items. If an item occurs in transactions 2, 5 and 7 out of 8 transactions, it will be represented by *01001010*. However as with all vertical data format algorithms we know that for sparse data sets, memory usage for bitset only data structure becomes very high. This is because if an item is present in very few transactions, representing its transaction set as bitset will take much more memory than if it had been represented as just an array. As shown in Section 5, we can see that instead of representing all transaction sets per frequent item as bitsets, if we save the sparse transaction sets as array of integers instead of bitsets we might be able to save memory as well as time.

6.2 Bitset and Array based Hybrid Data Structure

As explained in previous section, pure bitset data structures become less efficient for sparse datasets. For this reason, hybrid data structure was designed. In the first phase of building the data structure, we eliminate those items which are infrequent. For the rest of the items, we compute their cardinalities. For a given bitset, if *size of the bitset* > *frequency of the item* * w, then we call that item as a *low cardinality item* and if *size of the bitset* < *frequency of the item* * w, then we call that item as a *high cardinality item* (we use w to refer to the number bits per integer throughout this section). For a low cardinality item, instead of a bitset, we should represent it as an array of integers. Thus the data is represented partially by bitset and partially by array of integers, and hence the name *hybrid data structure*. Following are the steps to build the bitset and array hybrid data structure:

1. Generate a list of sorted frequent items as in pure Bitset data structure.

2. Find the item id in the sorted list of items beyond which all items are of high cardinality.

3. For each item, if it is of low cardinality, add all transaction ids where it is present to an item array. If it is of high cardinality, represent each transaction id where it is present by a bitset column.

6.3 HybridDSItr Algorithm
We have proposed an iterative algorithm based on the above hybrid data structure which enables us to prune a lot of candidates and save both time and memory.

6.4 Pruning Candidates
Vertical format mining suffers from the drawback of very high number of candidate item generation. Pruning as many candidates as possible is of utmost importance for such algorithms. As with most mining algorithms, direct supersets of infrequent candidates will not be evaluated. This, if AB was a low frequency itemset, then direct supersets of AB such as ABC, ABD, etc. will not be evaluated as they will also be low frequency itemsets.

Prune supersets of deleted items:
In this algorithm, another pruning mechanism has been used where indirect supersets are also rejected. We maintain a set of deleted items per iteration. If itemset AD is evaluated to be infrequent, it will be added to deleted items list.

In the same iteration when we evaluate candidates AB+D and AC+D, if AB - lastItem (highest frequency item in AB) + D - > AD is infrequent, then ABD and ACD are also rejected and added to deleted items. The new key set will now consist of A, AB, AC, ABC, AD and ACD. Figure 2 explains how candidates are generated and how infrequent candidates are rejected.

Prune subsets of highest frequency item: Our motivation is to eliminate as many intermediate candidates generated as possible. Again let us take the example of items A, B, C, D, E in low to high frequency order. In the first cycle, itemset combinations of A with B, C, D and E are considered. If for example, itemset ACDE is a frequent itemset. This implies that itemsets CDE and DE will also be frequent. If ABDE is a frequent itemset, then it implies that itemset BDE will also be frequent. As item sets with the high frequency item E will not be used to generate any more candidate itemsets, we can avoid intersection of bitsets or intersection arrays for these itemsets and simply report them as frequent. However, if we include this pruning mechanism in our algorithm, we will not be able to report frequencies for such patterns. Note that *FP-Growth* and *Eclat* algorithms report frequencies of all the frequent patterns. As we see later, this mechanism gives us slight improvement in time for sparse datasets. Thus depending on our requirement, we can choose to include this pruning mechanism or not.

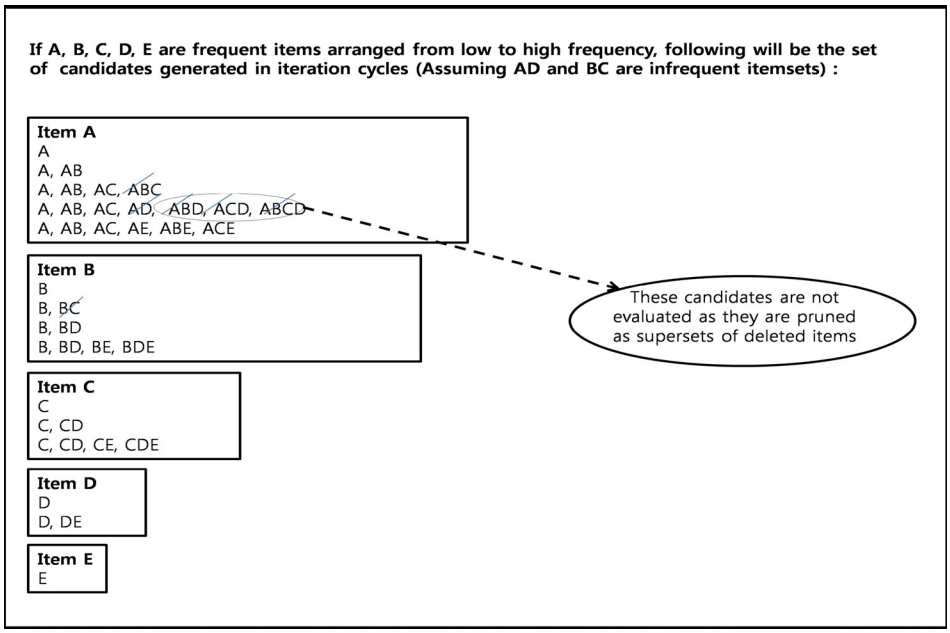


Figure 2. Iteration cycles for each item, with pruning of supersets of deleted items

Algorithm updateHighFreqSubSets(firstItem, newPattern)

1. newSubPattern = newPattern - firstItem - highestFrequencyItem

2. **for** each subSet s of newSubPattern **do**

3. create new pattern p = union (s, highestFrequencyItem)

4. l = lowestFrequencyItem of p

5. Add *p* to FrequentPatternsList for l

6.5 Algorithm

This section describes how the HybridDSItr algorithm works. We select each item in ascending order of frequency and generate candidates by combining it with items of higher frequency. A list of patterns is created that maintains all frequent patterns (which are supersets of item selected) being generated in current loop. A List of array of integers is maintained that stores arrays of integers for patterns that contain some items of low cardinality. A list of bitsets is maintained that stores bitsets for patterns that have all items of high cardinality. A map of highest frequency item supersets is maintained which maps item identifier and corresponding subsets which need not be evaluated as they are known to be frequent.

One by one, each pattern is selected from the list of patterns. A list of deleted itemsets is maintained in current loop. A new candidate pattern is created by taking a union of the pattern selected with the next item of higher frequency. If it is a superset of a deleted pattern, we add it to list of deleted patterns, else, we move forward. If the map of supersets of highest frequency pattern contains this pattern, then we simply report it as frequent and select the next pattern. Else, we move forward.

If the next item selected is of low cardinality, then we simply compute the intersection of the array of integers of the current pattern selected with the next item selected. If resulting array of integers has size > min-support, then it is added to the List of array of integers and new pattern is reported as frequent.

If next item is of high cardinality and all items in current pattern are of low cardinality, we create a new array of integers which contains intersection of array of integers for the selected pattern and bitset of the next item selected for candidate generation. If the size of this resultant array is > min-support, then it is added to the List of array of integers of the current pattern selected with the bitset of the next item selected. If the cardinality of this bitset is > min-support, then the new bitset created is added to the list of bitsets and the new pattern is reported as frequent. If it is a frequent pattern of size > three, and contains the highest frequency item as its subset, we add all its subsets to map of highest frequency item supersets.

If the frequency of the new candidate pattern is < min-support, it is added to the set of deleted patterns, else it is added to the list of patterns.

Here we describe the pseudo-code for the algorithm HybridDSItr in detail.

6.6 Analysis

The initial data structure created for HybridDSItr algorithm has the lowest memory usage as compared to the other two algorithms. However, the peak memory usage depends on the number of candidates generated during the mining procedure. For sparse datasets, the peak memory usage is less, and for dense datasets, the peak memory usage is high. This is because if very few candidates are rejected, then the search space becomes huge and thus the peak memory usage also becomes very high. The running time is less for sparse datasets where column cardinalities are very low for most of the items. For dense datasets, where column cardinalities are very high, the running time is comparatively high as there are more number of intersection operations due to high number of candidates generated. Also most of the intersection operations are between complete bitsets and not intersection between small arrays of integers.

Algorithm HybridDSAlgorithm

1: procedure HYBRIDDSALGORITHM

```
2:  Input : Hybrid Data Structure of List of Set of Integers hS, Array of Vertical
    Bitsets hB, number of frequent items N, size of hS C, Set of frequencies of frequent items F
3:  Output : Frequent patterns (FPs)
4:  Create new map L of item and corresponding keys containing item and highest frequency item
5:  for each frequent item pos in ascending order do
6:          Create new pattern nP with pos as only item
7:          Report nP as Frequent Pattern
8:          Create new List P of Patterns with nP as first element
9:          Create new List I of Array of Integers
10:         Create new List B of BitSets
11: for each frequent item i > pos in ascending order do
12:         currentListSize = P.size;
13:         Create new Set D of Deleted patterns;
14: for each k = 0, k < currentListSize do
15:         =newPattern = P.get(k);
16: chkPattern = newPattern - newPattern.lastElement + i;
17: if D !contains (chkPattern) then
18: if L.contains(i) then
19: add newPattern to P;
20: exit current loop;
21: else
22:         if i < hS.size and k < I.size then
23:                 Array a = I.get(k) intersects hS.get(i);
24:                 countFrequency = a.size;
25:         if countFrequency >= support then
26:                 Add a to I;
27:         if i > hS.size and k < I.size then
28:                 Array a = I.get(k) intersects hB.get(i - hS.size);
29:                 countFrequency = a.size;
30:         if countFrequency >= support then
31:                 Add a to I;
32:         if i > hS.size and k > I.size then
33:                 BitSet b = B.get(k - I.size) intersects hB.get(i - hS.size);
34:         if b.size >= support then
35:                 countFrequency = b.size;
36:         if countFrequency >= support then
37:                 Add b to B;
38:         if countFrequency >= support then
39:                 Add newPattern to P;
40:                 Report nP as Frequent Pattern;
```

```
41:                 if i == N - 1 and newPattern.size >= 3 then
42:                         updateHighFreqSubSets(pos, newPattern)
43:                          else
44:                                 Add newPattern to D;
45:         else
46:                         Add newPattern to D;
```

7. Experimental Study

Several experiments have been performed to analyze the time and memory usage of the new hybrid data structure based iterative algorithm with *Eclat* and *FP-Growth* algorithms. We have written the code in Java 1.7. The source code of *FP-Growth* and *Eclat* is also in Java and has been downloaded from SPMF website [7]. The code has been executed on a machine with Intel(R) Xeon(R) CPU E7450 @ 2.40GHz system with 4GB of main memory.

7.1 Dataset

We have used 8 datasets: CHESS, MUSHROOM, T10I4D100K, T40I10D100K, T20I6D100K, T20I4D100K, T60I10D100K and T80I10D100K in our experiments. Off these eight, the first four datasets have been downloaded from FIMI repository [9]. The remaining four synthetic datasets have been generated using IBM quest synthetic data generator [10]. The detailed information about generation of synthetic dataset and its characteristic are described in [5]. Table 3 shows characteristics of each dataset we have used.

Dataset	Dense/Sparse	# Items	# Trans	Avg len (%)	FileSize (MB)
T10I4D100K	Sparse	870	100,000	11 (2.04)	4.02
T20I6D100K	Sparse	980	99,944	20 (2.04)	7.44
T40I10D100K	Sparse	942	100,000	40.5 (2.04)	15.21
T20I4D100K	Sparse	980	99,944	20 (2.04)	7.63
T60I10D100K	Sparse	2000	100,000	60 (2.04)	26.46
T80I10D100K	Sparse	2000	100,000	80 (2.04)	35.28
CHESS	Dense	75	3,196	37 (49.33)	0.33
MUSHROOM	Dense	119	8,416	23 (19.16)	0.57

Table 3. Dataset characteristics

7.2 Memory Comparison

For comparing memory usage for the three algorithms, we calculated the peak memory usage reached during data structure building phase or the mining phase, whichever was greater. *Eclat* and *FP-Growth* algorithms both have been implemented using tree structures. Their peak memory usages are - maximum nodes created at any point of time * size of a node. The following variables have been used in calculating the memory usage for the algorithms:

- t = number of transactions with frequent items
- p = pointer size in bytes
- n = integer size in bytes
- N = nodes existing at time of peak memory usage
- K = Sum total of number of integers / key existing at time of peak memory usage

Assuming that items have been represented as integers, the memory usage for the three algorithms is as following:

Memory usage for FP-Growth
FP-Growth stores three pointers per node (parent pointer + 2 pointers to left and right nodes) + two integers (itemidentifier and frequency). The memory usage will be the following:
Peak memory usage in bytes = $(3 * p + 2 * n) * N$.

Memory usage for Eclat
Eclat stores 1 pointer(parent pointer) + bitset (size equal to number of transactions with frequent items) + the one itemset or pattern per node. The memory usage will be the following:
Peak memory usage in bytes = $(t/8 + p) * N + K * n$.

Memory usage for HybridDSItr
The memory usage consists of candidate itemsets and their transaction sets represented as bitsets or array of integers. Assuming :

- N_1 = Number of bitsets existing at time of peak memory usage
- N_2 = Sum total of number of integers per array of integers existing at time of peak memory usage the memory usage will be the following:

Peak memory usage in bytes = $(t/8) * N_1 + N_2 * n + K * n$.

7.3 Result
Figure 3 shows the memory comparison for HybridDSItr, FP-Growth and Eclat algorithms for T20I6D100K, T20I4D100K, T10I4D100K, T40I10D100K, T60I10D100K, T80I10D100K, CHESS and MUSHROOM datasets.

7.4 Discussion
We can see from Figure 3 that HybridDSItr performs better than both algorithms or better than FP-Growth and some- what similar to Eclat for sparse datasets. Though size of initial datastructure for iterative HybridDSItr is small, sometimes candidates existing at a given point in mining become more or less than Eclat which is a recursive algo- rithm. Due to this, peak memory usage sometimes exceeds that of Eclat algorithm and sometimes is lesser than Eclat algorithm. We observe the opposite behaviour for the dense datasets. Here number of frequent patterns generated are very high and thus the intermediate candidate generation is high for HybridDSItr. Also in tree recursions with depth first search, we are able to discard subtrees as we go deep down which does not happen in iterative phase and we retain more candidate keysets and candidate bitsets in memory for HybridDSItr algorithm.

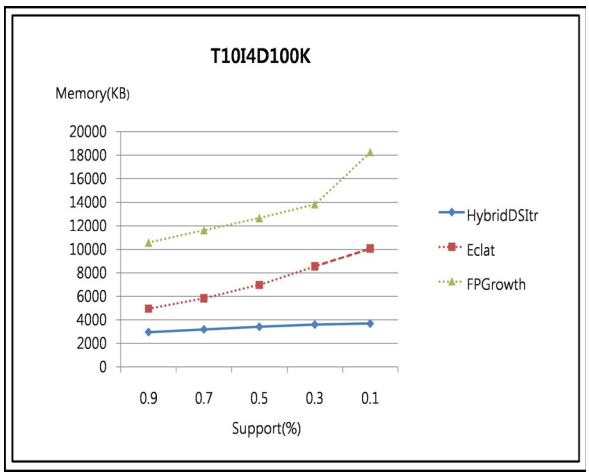
7.5 Time Comparison
In this section we compare the time performance for the three algorithms HybridDSItr, FP-Growth and Eclat for all the eight datasets. For the sparse datasets, time is in logarithmic scale of base 10. To measure time, System.currentTimeMillis function was used. Code was configured such that that frequent itemsets were stored in memory
for all the three algorithms.

7.6 Result
Figure 4 depicts the time comparison between the three algorithms. HybridDSItr performs best for sparse datasets. Hy- bridDSItr performs better than both Eclat and FP-Growth algorithms. FP-Growth performs worst for sparse datasets. HybridDSItr performs worst for dense datasets which was as expected.

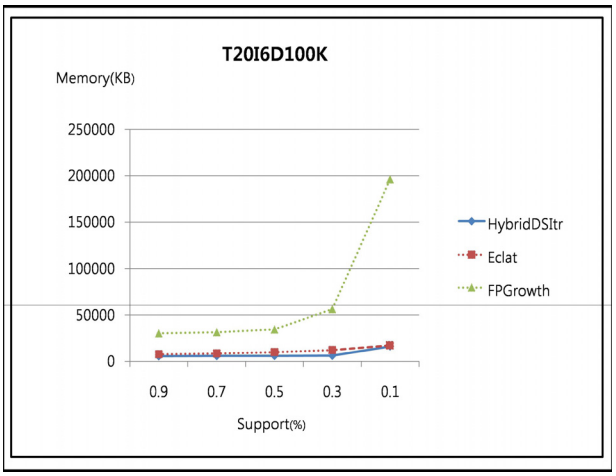
7.6.1 HybridDSItr vs PlainDSItr
Figure 5 depicts the time comparison for HybridDSItr(version with both bitsets and array of integers) with PlainDSItr (version with only bitsets). We have only used sparse datasets here as only they have items with low cardinality. For dense datasets, all items have high cardinality. For such datasets, both HybridDSItr and PlainDSItr will perform similarly.

7.6.2 HybridDSItr (Pruning supersets of deleted items)
Figure 6 depicts the time comparison for HybridDSItr with and without pruning of supersets of deleted items respec- tively. For sparse datasets, HybridDSItr performs very well with pruning of supersets of deleted items. For dense datasets, HybridDSItr

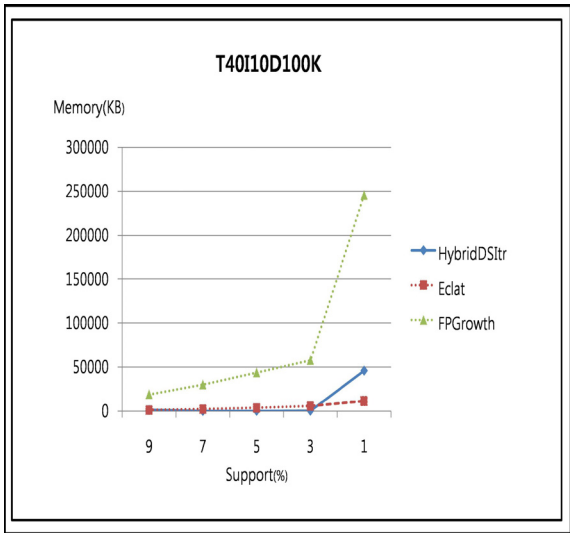
performs better without pruning of supersets of deleted items. This could be because number of deleted candidates for dense datasets is very small and thus this pruning mechanism only adds more time.



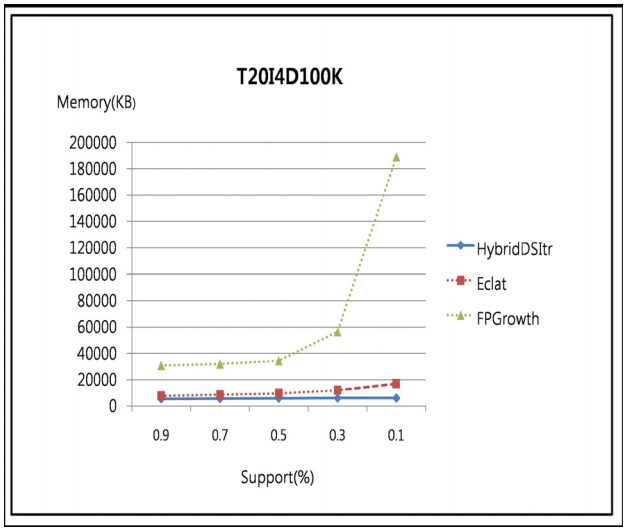
(a) T10I4D100K



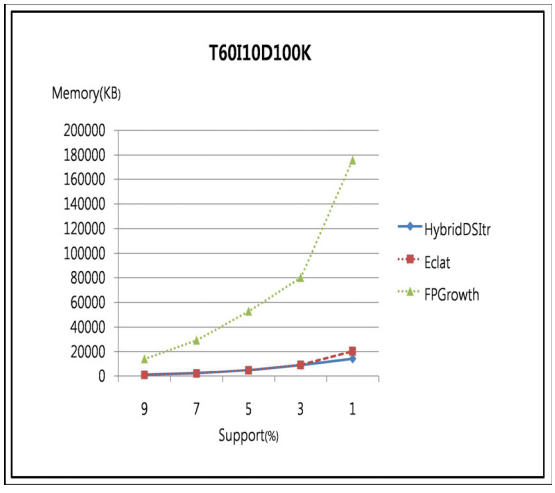
(b) T20I6D100K



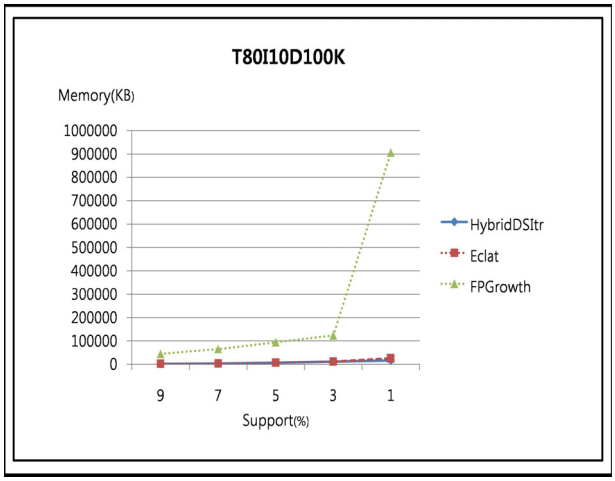
(c) T40I10D100K



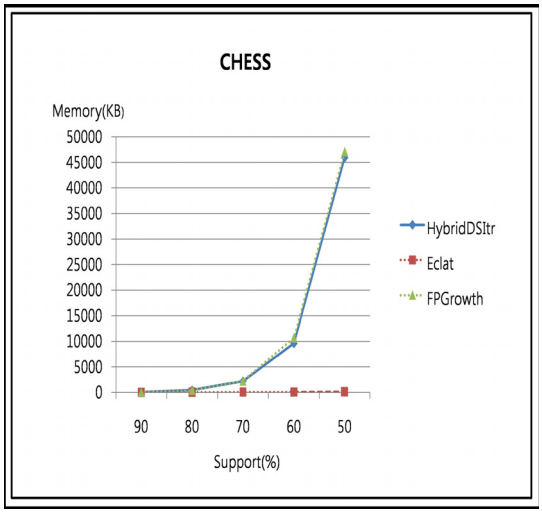
(d) T20I4D100K



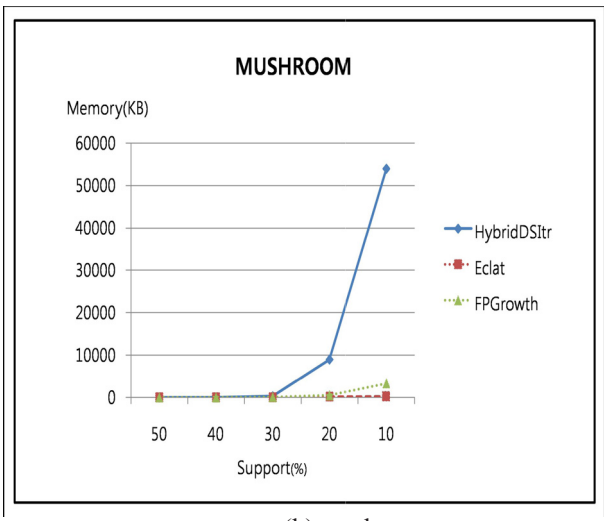
(e) T60I10D100K



(f) T80I10D100K

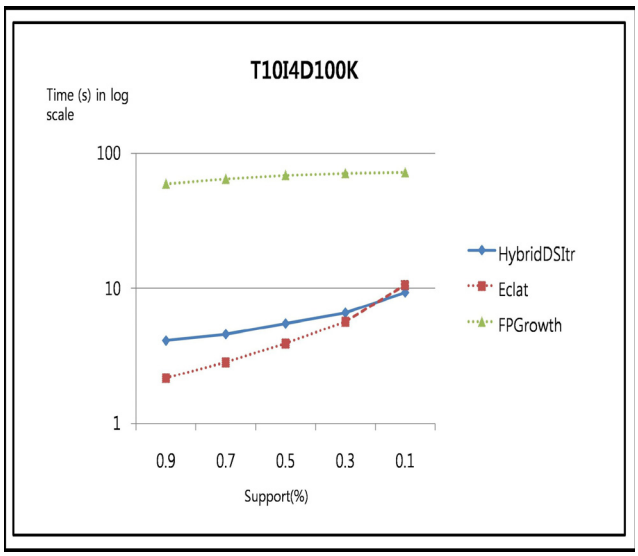


(g) chess

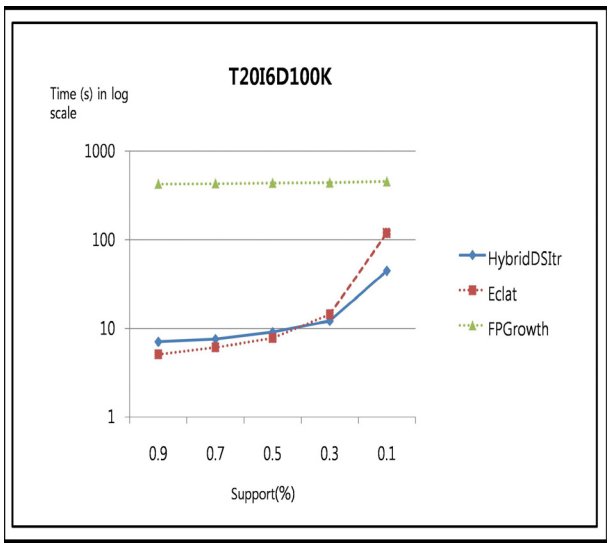


(h) mushroom

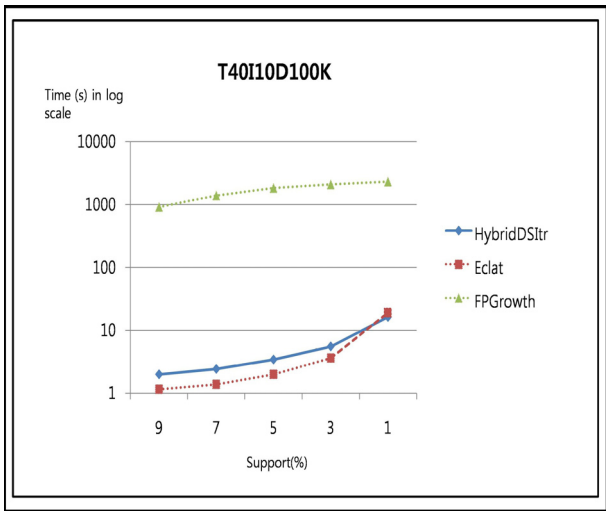
Figure 3. Memory Usage



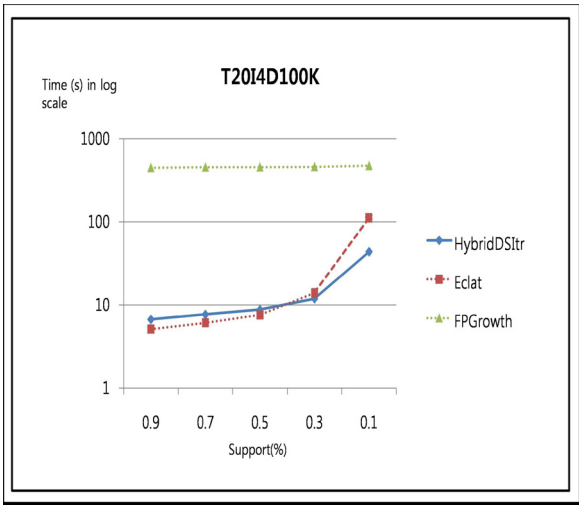
(a) T10I4D100K



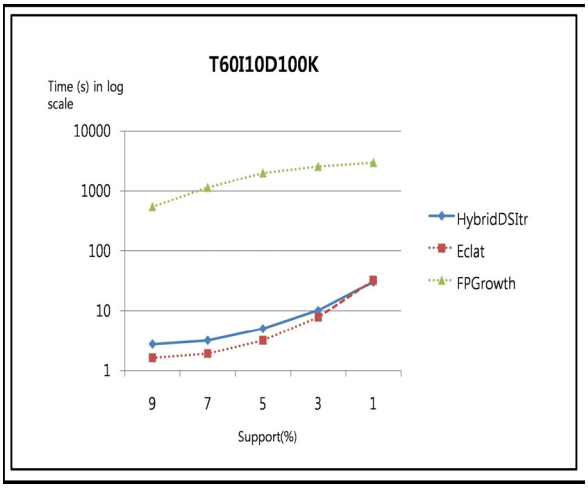
(b) T20I6D100K



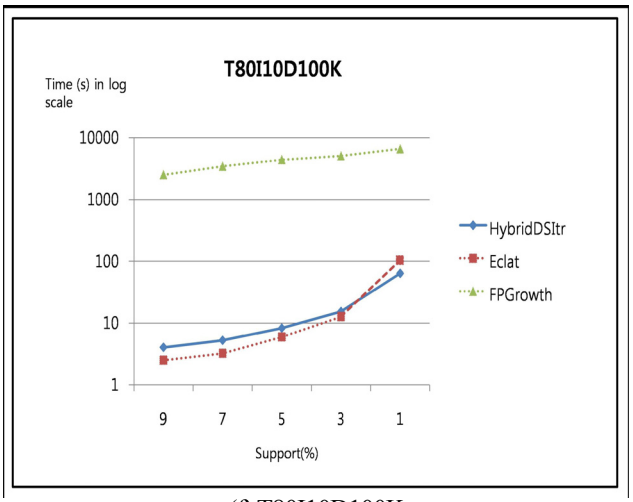
(c) T40I10D100K



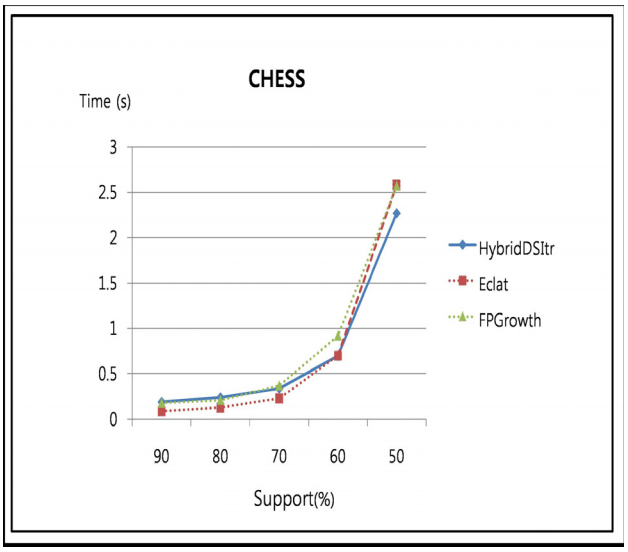
(d) T20I4D100K



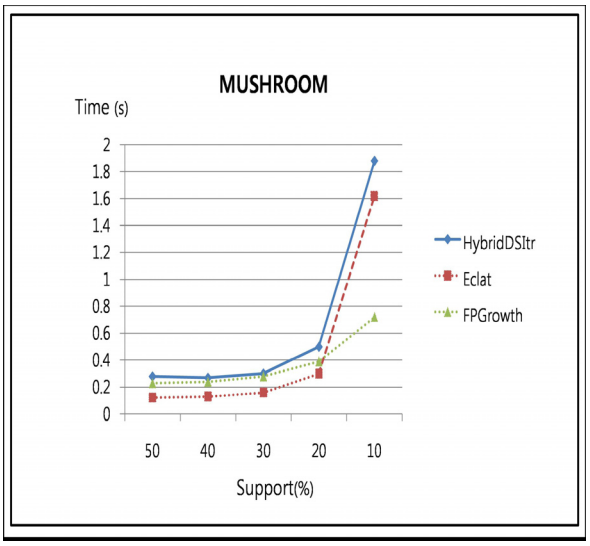
(e) T60I10D100K



(f) T80I10D100K

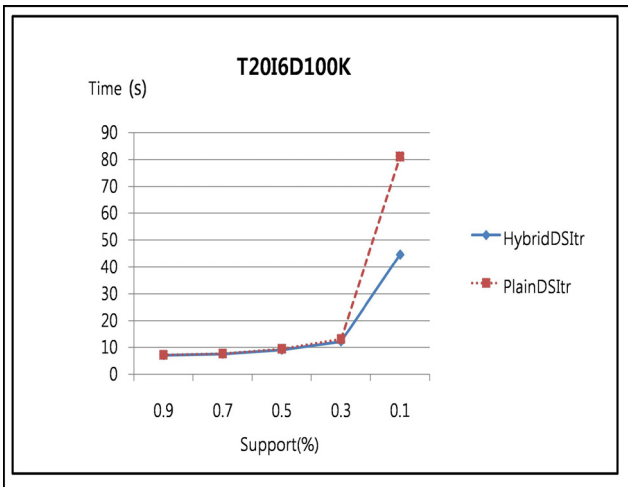


(g) CHESS

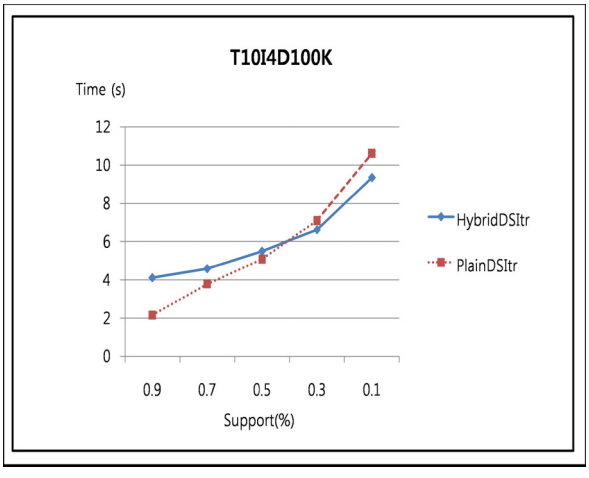


(h) MUSHROOM

Figure 4. Time usage

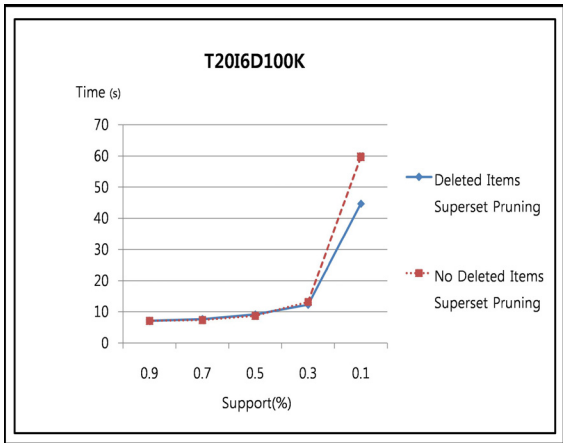


(a) T20I6D100K

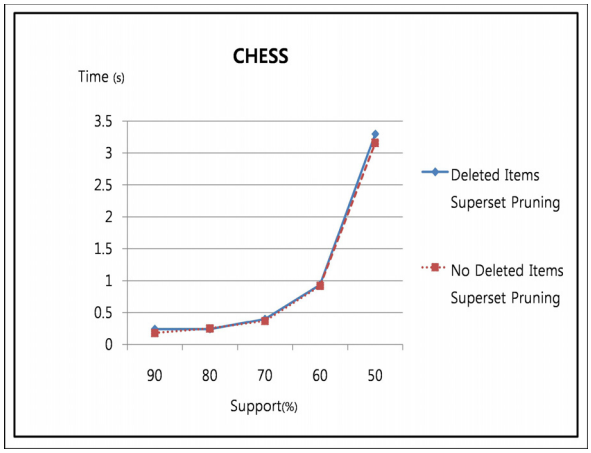


(b) T10I4D100K

Figure 5. HybridDSItr vs PlainDSItr



(a) T20I6D100K

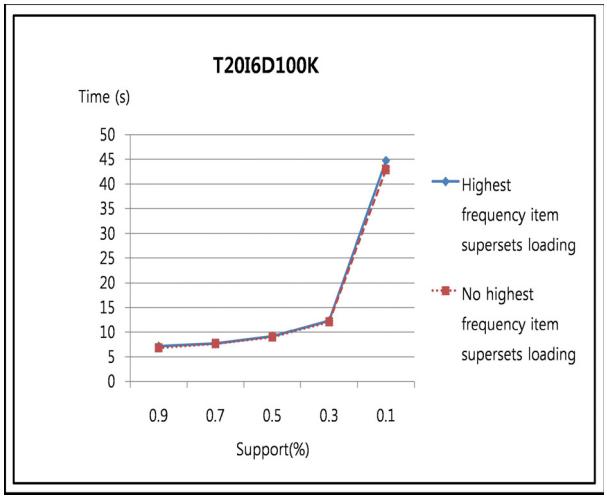


(b) CHESS

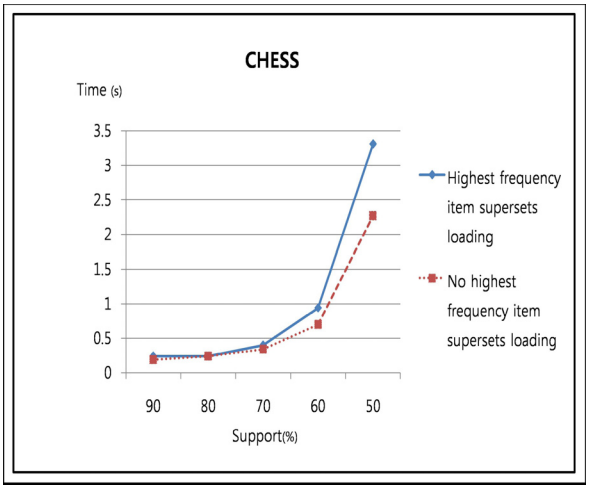
Figure 6. Pruning supersets of deleted itemsets

7.6.3 HybridDSItr (Pruning supersets of highest frequency item)

Figure 7 depicts the time comparison for *HybridDSItr* with and without code for pruning supersets of highest frequency item respectively. For the sparse datasets, there is very slight gain with this code. For dense datasets, HybridDSItr performs better without code for pruning supersets of highest frequency item. This is because time spent in calculation of such supersets is more than time gain in avoiding bitsets computation for such candidates with highest frequency item as subsets.



(a) T20I6D100K



(b) CHESS

Figure 7. Pruning supersets of highest frequency item

7.7 Discussion

HybridDSItr algorithm displays better performance in *time* for sparse datasets. This is because due to the hybrid structure, the number of intersection operations per pair of items is minimum, also candidate generation is less as we prune supersets of already deleted itemsets. For dense datasets, when number of intermediate candidate generation increases, its performance deteriorates. This is because, in dense datasets, most of the items are represented by bitsets and as very few candidates are rejected, pruning supersets of deleted itemsets is also minimum.

8. Conclusion

In this paper, we have designed a memory-efficient data structure (*HybridDS*) and a time-efficient mining algorithm (*HybridDSItr*)

for sparse datasets. We have compared the new algorithm in terms of memory and time with *FP-Growth* and *Eclat* algorithms. In terms of time usage, *HybridDSItr* performs better as compared to *Eclat* and *FP-Growth* for sparse datasets. In terms of memory usage, *HybridDSItr* performs better as compared to *FP-Growth* and similar to *Eclat* for sparse datasets. *HybridDSItr* performs worse in both memory and time in case of dense datasets. This research on implementing a space efficient data structure and time-efficient algorithm for sparse datasets can be utilized to optimize existing algorithms for different mining problems.

References

[1] Annie, Loraine Charlet., Kumar, Ashok. (2012). *Market Basket Analysis for a Supermarket based on Frequent Itemset Mining*. *International Journal of Computer Science*, 9 (5) 3 257-264.

[2] Liu., Tantan., Agrawal, Gagan. (2011)*Active Learning Based Frequent Itemset Mining Over the Deep Web*. IEEE 27th International Conference on Data Engineering.

[3] Naulaerts, Stefan., Meysman, Pieter., Bittremieux, Wout., Vu, Trung Nghia., Berghe, Wim Vanden., Goethals, Bart., Laukens, Kris (2013). *A Primer to frequent itemset mining for bioinformatics*. *Bioinformatics Advance Access*.

[4] Han, Jiawei., Pei, Jian., Yi, Yiwen. (2000). *Mining frequent patterns without candidate generation*. *In*: Proceedings of the 2000 ACM SIGMOD international conference on Management of data.

[5] Agrawal., Srikant. (1994). *Fast Algorithms for Mining Association Rules, In*: Proceedings of the 20th international conference on very large data bases, *VLDB*.

[6] Borgelt, Christian. (2005). *An implementation of the FP-Growth algorithm . In*: Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations.

[7] *SPMF : Website for open source implementation of mining algorithms*. http://www.philippe-fournierviger. com/spmf/.

[8] *HybridDSItr (2015). Source code*. https://github.com/nehadwivedi/HybridDSItr.

[9] *FIMI*. http://fimi.ua.ac.be/data/.

[10] *IBM Quest Synthetic Data Generator*. http://sourceforge.net/projects/ibmquestdatagen/.

[11] Zaki, M. J. *(*2000). Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12 (3) 372-390.

[12] Jie-Dong, Min-Han *(2007)*. BitTable-FI - An efficient mining frequent itemsets algorithm, *Knowledge Based Systems,* 20 (4) 329-335.

[13] Song, Wei., Yang, Bingru., Xu, Zhangyan (2008). Index-BitTableFI - An improved algorithm for mining frequent itemsets. *Knowledge Based Systems,* 21 (6) 507-513.

[14] Vo, Bay., Hong, Tzung-Pae., Le, Bac (2012). *DBV-Miner: A Dynamic Bit-Vector approach for fast mining frequent closed itemsets*. *Expert Systems with Applications*, 39 (8) 7196-7206.

[15] Dwivedi, Neha., Satti, Srinivasa Rao (2015). Set and Array based Hybrid Data Structure Solution for Frequent Pattern Mining. *In:* Tenth International Conference on Digital Information Management, Jeju, S.Korea, October 21-23, 2015. ICDIM 2015.