Efficient Indexing Structures for Mining Frequent Patterns

Bin Lan, Beng Chin Ooi, Kian-Lee Tan

Department of Computer Science

National University of Singapore

3 Science Drive 2, Singapore 117543

Abstract

In this paper, we propose a variant of signature file, called *Bit-Sliced Bloom-Filtered Signature File* (BBS), as the basis for implementing filter-and-refine strategies for mining frequent patterns. In the filtering step, the candidate patterns are obtained by scanning BBS instead of the database. The resultant candidate set contains a superset of the frequent patterns. In the refinement phase, each algorithm refines the candidate set to prune away the false drops. Based on this indexing structure, we study two filtering and two refinement mechanisms, thus giving rise to four different strategies. For the filtering mechanisms, we study a single filter and a dual filter mechanisms. The latter mechanism essentially introduces a second filter on the result obtained from the first filter (which is the same as the single filter method). The crux of the second filter lies in its ability to obtain accurate counts on the number of occurrences for some patterns by maintaining some additional information. For the refinement step, we also explore two mechanisms: sequential scan and probe. While the former sequentially scans the database to verify the validity of each candidate pattern, the latter only examines relevant tuples in the database that match each candidate pattern. We conducted an extensive performance study to study the effectiveness of BBS, and compared the four proposed processing schemes with the traditional Apriori algorithm and the recently proposed FP-tree scheme. Our results show that BBS, as a whole, outperforms the Apriori strategy. Moreover, one of the schemes that is based on dual filter and probe refinement performs the best in all cases.

Keywords: Frequent patterns, bloom filter, signature file, Apriori, FP-tree, BBS.

1 Introduction

Mining frequent patterns is a fundamental step in data mining. Almost all important data mining tasks, such as association rule mining, correlations and causality, require frequent patterns to be mined first. As such, this process has received much attention from the research community [1, 2, 4, 5, 6, 7, 8, 9, 11, 13, 14, 17, 18, 19].

The process of mining frequent patterns involve generating the candidate patterns and testing whether the patterns are frequent (i.e., the number of occurrences of the pattern is larger than a predetermined threshold). This process, unfortunately, cannot be avoided, and is independent of the implementation techniques. Most of the existing works are largely based on the Apriori heuristics [1]. The basic idea

behind this scheme is to iteratively generate the set of candidate patterns of length $(k + 1)$ from the set of frequent patterns of length k $(k \geq 1)$, and check their corresponding occurrence frequencies in the database. While the heuristic reduces the size of the candidate sets, the process is still expensive as it requires multiple scans over the database.

More recently, Han, Pei and Yin proposes a *frequent pattern tree* (FP-tree) structure, which is an extended prefix-tree structure, to store compressed, crucial information about frequent patterns [8]. Frequent patterns can then be mined from the FP-tree instead of the database. The paper proposes an efficient method, called FP-growth, that exploits the FP-tree to mine the *complete set of frequent patterns* by pattern fragment growth. While the approach is shown to outperform the Apriori scheme, the cost incurred to build the FP-tree and other auxiliary structures is a significant component of the overall cost of the algorithm. This is because the FP-tree is not a dynamic structure, i.e., it has to be constructed whenever the database is mined for frequent patterns. Another way of looking at this is that the scheme is effective for static databases, and any updates require rebuilding the FP-tree as it is difficult to support incremental updates to the FP-tree.

In this paper, we propose a dynamic and persistent data structure, *Bit-Slice Bloom-Filtered Signature File* (BBS), which integrates concepts from bloom filters and signature file, as an indexing structure for facilitating frequent pattern mining. Each transaction is mapped into a m-bit vector based on multiple hash functions, and the collection of m-bit vectors becomes an approximate representation of the database in a compact form. Operations on the bit vectors are performed on the m bit-slices derived from the vectors. More importantly, because BBS is persistent, new transactions added to the database simply correspond to insertions into the BBS – there is no need to reconstruct the BBS upon every update.

To efficiently mine frequent patterns, we study four novel filter-and-refine algorithms that use BBS as the underlying index. The proposed algorithms comprise two phases: a filtering phase and a refinement phase. In the first phase, each algorithm identifies a set of candidate frequent patterns from BBS (instead of the database). Because of the approximate representation of the database, the resultant candidate set contains a superset of the frequent patterns, i.e., it contains all the frequent patterns plus some false drops[1] (i.e., some infrequent patterns are identified as frequent). This phase is computationally inexpensive because BBS is typically small (and will not take too many scans if it does not fit into the memory) and operations on BBS are bitwise operations. In the second phase, the refinement phase, each algorithm refines the candidate set to prune away the false drops.

We study two filtering and two refinement mechanisms, thus giving rise to four schemes. For the filtering mechanisms, we study a single filter and a dual filters mechanisms. In the single filter mechanism, the candidate set is obtained by scanning the BBS. In the dual filters technique, the first filter is the same as the single filter scheme. The second filter further removes some of the false drops from the candidate set obtained from the first filter. The crux of the second filter lies in its ability to identify some patterns with 100% guarantee to be in the final answer set. This is achieved by providing an accurate count of the number of occurrences for these patterns by maintaining some additional information. For the

---

[1] *False drop* is also called false positive, or false hit, or Type II error. We use the term for conventional reason.

refinement step, we also explore two mechanisms: sequential scan and probe. Under the sequential scan scheme, the database is sequentially scanned to verify the validity of each candidate pattern. On the other hand, the probe scheme only examines relevant tuples in the database that match each candidate pattern. We also examine how the proposed scheme can be used to support more general ad-hoc queries involving constraints.

There is also a tradeoff between the size of a bit vector used to represent a transaction (i.e., the value of m) and the degree of false drops. A large value of m will minimize the number of false drops, but may increase the processing cost since a large BBS will incur higher I/O cost. On the contrary, a small m value will lead to an increase in the number of false positives. While a smaller BBS may incur lower I/O cost, it may take a longer time to prune away a larger number of false positives. Tuning the size of BBS for optimal performance is critical.

We conducted an extensive performance study on BBS, and compared the four proposed schemes with the traditional Apriori algorithm and the recently proposed FP-tree scheme. Our results show that BBS is scalable in terms of the number of transactions and items. For the proposed scheme, a reasonably small number of bits (relative to the total number of items in the database) is sufficient to represent each transaction to give good performance. Comparatively, our study shows that all the proposed processing schemes are more efficient than the Apriori strategy. One of the schemes that is based on dual filter and probe refinement performs the best in all cases. Moreover, the proposed schemes can maintain its effectiveness for dynamic workload, which FP-tree is unable to handle.

The remainder of this paper is organized as follows. In the next section, we present the proposed BBS structure, and how it can be used to estimate the number of occurrences of a given pattern. In Section 3, we present the filtering and refinement schemes, and the four proposed algorithms. Section 4 presents the experimental study conducted, and reports the performance results, and finally, we conclude in Section 5 with directions for future work.

2   The Bit-Slice Bloom Filtered Signature File

Let $I = \{i_1, i_2, \cdots, i_N\}$ be a set of N distinct literals called *items*. The database, D, consists of a set of variable length transactions over I. Each transaction contains a set of items $i_1, i_2, \cdots, i_n \in I$. A transaction has an associated unique identifier called TID. In this section, we shall present a dynamic and persistent structure, called Bit-Slice Bloom Filtered Signature File (BBS), that serves as a compact but lossy representation of the database. Moreover, we shall present an algorithm that operates on BBS to *estimate* the number of transactions containing a given item set.

2.1   The BBS Structure

BBS is based on two concepts: signature file [15, 16] and bloom filter [3]. Like the signature file, it comprises a set of fixed-length *signatures*, where each signature is a m-bit vector corresponding to a transaction in the database. However, BBS adopts the bloom filter concept to represent each transaction.

Transaction ID	Item set	Bit Vector
100	$0, 1, 2, 3, 4, 5, 14, 15$	11111111
200	$1, 2, 3, 5, 6, 7$	01110111
300	$1, 5, 14, 15$	01000111
400	$0, 1, 2, 7$	11100001
500	$1, 2, 5, 6, 11, 15$	01110111

Table 1: A transaction database as running example.

0	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
0	1	1	1	0	1	1	1
0	1	0	0	0	1	1	1
1	1	1	0	0	0	0	1
0	1	1	1	0	1	1	1

Table 2: The BBS of Example 1.

The method works as follows. Let a transaction consist of n items[2], say $i_1, i_2, \ldots, i_n$, and the vector v be m bits. Initially, all the m bits of v are set to 0. k independent hash functions, h_1, h_2, $\ldots$, h_k, are then applied on each item i_j ($1 \leq j \leq n$). The hash functions are picked such that $h_j(a)$ ($1 \leq j \leq k$) returns a value in the range $\{1, \ldots, m\}$. For each item, i_j ($1 \leq j \leq n$), the bits at positions $h_1(i_j), h_2(i_j), \ldots, h_k(i_j)$ in v are set to 1. Clearly, a particular bit might be set to 1 multiple times.[3] For efficient computation, the array of vectors can be transposed into an array of *bit-slices*, and the file is stored as slices. This explains the name given to the structure – *bit-slice bloom-filtered signature file*.

Example 1 (Running Example) *We shall use the following example as a running example in this paper. For simplicity, let us assume that there is only one hash function of the form $h(x) = x \bmod 8$. Furthermore, assume that we represent each transaction with a 8-bit vector. Table 1 shows a database of 5 transactions, a total of 16 items (numbered from 0 to 15) in the database, and the corresponding vectors of the transactions. Here, we note that the first bit of the vector corresponds to a hash value of 0, and the second bit of the vector corresponds to a hash value of 1, and so on. Table 2 shows the corresponding BBS. As shown, there are a total of 8 bit-slices in this example. From the two tables, we observe that BBS is a lossy representation of the database. For example, both transaction 200 and 500 are represented using the same bit vector, so that given the bit vector, we are not able to distinguish between the two.* □

---

[2] We note that different transactions may have different number of items.

[3] An alternative method to obtain the m-bit vector, which is employed by the signature file method, is to hash each item into an m-bit vector and superimpose (inclusive-OR) all the vectors of all items of the transaction. The bloom filter approach is preferred here because it allows us to control the number of bits to be set. Technically speaking, we can always replace the single hash function of the signature file method by multiple hash functions that behave in the same way as the bloom filter scheme. However, since this idea was formally known as bloom filter, we shall give it its due credit.

2.2 Counting Occurrences of Items in BBS

Although BBS is a lossy representation of the database, its compact representation and support for bitwise operations allows it to come up with an *estimate* of the number of transactions containing a set of items quickly. The algorithm, called CountItemSet, is shown in Figure 1. As shown in the algorithm, CountItemSet first obtains the bit vector of the set of items, v, to be counted (step 1). Next, it picks up the bit slices of BBS whose corresponding bits in v are set. These bit slices are then logically ANDed to obtain a resultant bit vector, s (step 2). Finally, we can determine the number of bits that are set in s. This value corresponds to the estimated number of occurrences of the itemset in the database (obtained from BBS).

Input: a set of items $I = \{i_1, i_2, \ldots, i_n\}$
Output: the number of occurrences of I in BBS
    1.   Obtain the m-bit vector for I, v.
    2.   Logical-ANDed all the bit-slices of BBS whose corresponding
            bits in v are set. Let the resultant bit set be s.
    3.   Return the number of bits set in s.

Figure 1: Algorithm CountItemSet

It is also worth nothing that the algorithm can be applied to identify both frequent patterns and non-frequent patterns. That is, we can estimate the count of any patterns using the algorithm. We shall derive several interesting properties of algorithm CountItemSet here.

Lemma 1 *Given an item a and a transaction with item set I. Let the corresponding vectors of a and I be v and v' respectively. If $\exists j$ such that the jth bit of v is set while the jth bit of v' is not set, then $a \notin I$, and the bit in the resultant bit string of algorithm CountItemSet corresponding to the transaction will not be set.*

    Proof. (Reduction to absurdity) *Suppose $a \in I$. This means that for every bit in v that is set, the corresponding bit in v' should also be set. Since the jth bit of v is set, therefore the jth bit of v' is also set. This contradicts the assumption. Thus $a \notin I$.*

    Moreover, since the jth bit of v is set, it means that one of the bit-slices that algorithm CountItemSet uses in the computation will be the jth bit-slice. However, since the jth bit of v' is not set, it follows that the bit for the transaction corresponding to the resultant bit set (of algorithm CountItemSet) will be 0.   □

Lemma 2 *Given an item set I and a transaction in the database with item set I', and their corresponding vectors be v and v'. If $\exists j$ such that the jth bit of v is set while the jth bit of v' is not set, then $I \not\subset I'$. Moreover, the bit in the resultant bit string of algorithm CountItemSet corresponding to the transaction will not be set.*

    Proof. *Because the jth bit of v is set, then there is one item $a \in I$ with one hashing function h_i which satisfies*

$$h_i(a) = j$$

According to Lemma 1, $a \notin I'$. Then $I \not\subset I'$.

Moreover, since the jth bit of v is set, it means that one of the bit-slices that algorithm *CountItemSet* uses in the computation will be the jth bit-slice. However, since the jth bit of v' is not set, it means that the bit for the transaction corresponding to the resultant bit set (of algorithm *CountItemSet*) will be 0.

$\square$

Lemma 3 *Given a set of items $I = i_1$, i_2, ..., i_n, and a transaction containing a set of items $I' = i_1$, i_2, ..., i_n, i_{n+1}, ..., i_j $(j \geq n)$. Then, algorithm CountItemSet will not miss the transaction in its computation of the number of occurrences of I in BBS.*

Proof. *Let v and v' be the m-bit vectors of I and I' respectively. Since $I \subset I'$, it means that for every bit in v that is set, the corresponding bit in v' is also set. Now, algorithm CountItemSet logically-ANDed the set bits of all corresponding bit-slices. This means that the bit in the resultant bit array of algorithm CountItemSet corresponding to v' will also be set. Thus, the transaction is not missed in the computation.* $\square$

Lemma 4 *Given a database D and its BBS, and a set of items $I = \{i_1$, i_2, ..., $i_n\}$. The number of occurrences of I in BBS determined by algorithm CountItemSet is no smaller than the actual number of occurrences of I in the database D.*

Proof. *This follows from Lemma 3. Since algorithm CountItemSet does not miss any transactions that contain I, it means that the answer returned by algorithm CountItemSet should at least be as large as the actual count obtained from D.* $\square$

The implications of the above Lemmas are:

- From Lemmas 1 and 2, we can see that algorithm CountItemSet can be effectively used to prune away transactions that have different bit representation from that of the item set that we are counting.

- From Lemma 3 , we understand that CountItemSet will not have any *false misses*. In other words, if any transaction contains the item set that we are counting, that transaction will be counted.

- From Lemma 4, we see that it is possible to over-estimate the number of transactions containing an item set. In other words, CountItemSet may generate *false drops*. This follows from the fact that BBS is a lossy representation and multiple transactions containing different item sets may be mapped to the same bit vector.

- From Lemmas 3 and 4, we can prune away certain small item sets quickly, i.e., an item set whose estimated count is below the support threshold is certainly not a large item set.

Example 2 *We shall illustrate CountItemSet using our running example. Suppose we want to determine the number of transactions containing item set $I = \{0, 1\}$. Using the same hash function, we note that the vector for I is 11000000. As such, we only need to examine bit slices 1 and 2 of the BBS. From Table 2, we note that the two slices are 10010 and 11111. By logically ANDing the two slices, we get the resultant bit vector of 10010 which indicates that there are two transactions containing I. Here, the answer obtained is accurate, reflecting the actual number of transactions containing I. On the other hand, if we were to determine the number of transactions containing $I = \{1, 3\}$, we will obtain a value of 3. This value turns out to be larger than the actual count of 2. Thus, algorithm CountItemSet can over-estimate the number of transactions containing an item set.* □

We also note that there is a tradeoff between the size of the bit vector, m, and the number of false drops. When $m = 1$, we have one extreme case of BBS returning the cardinality of the database as the answer for all item sets to be counted. On the other extreme, when m is equal to the number of items in the database, all transactions containing different item sets will be mapped to different vectors[4]. Thus, we can obtain a precise and accurate count on the number of transactions containing an item set. Essentially, a large value of m will minimize the number of false drops, but may increase the processing cost since the size of the BBS is large. On the contrary, a small m value will lead to an increase in the number of false drops. While a smaller BBS will incur smaller number of I/Os, it may take a longer time to prune away a larger number of false drops. Tuning the size of BBS for optimal performance is critical. Fortunately, as we shall see in our experimental results, the size of the bit vector can be kept to a sufficiently small value without excessive number of false drops.

3   Filter-and-Refine Algorithms for Mining Frequent Patterns

In this section, we present four novel algorithms that can efficiently mine frequent patterns. The proposed algorithms comprise two *logical*[5] phases: a filtering phase and a refinement phase. We shall first look at the filtering schemes, followed by the refinement strategies. Finally, we shall present the proposed algorithms.

3.1   Filtering Algorithms

In the filtering phase, the objective is to determine a superset of the frequent itemsets *quickly*. To realize this, we scan the BBS instead of the database. Since BBS is compact and bitwise operations are fast, this phase is computationally inexpensive. We propose two filtering algorithms in this section.

---

[4]Here, the hash function essentially maps each distinct item to a different bit in the vector.

[5]As we shall see, in the implementation of some of the schemes, we have integrated the two phases, rather than treating them as two distinct and separate phases. Such integrated approach has several advantages which we shall discuss in the respective algorithm.

Algorithm SingleFilter

The first algorithm, algorithm SingleFilter, is shown in Figure 2. The algorithm comprises two parts, an *initialization and invocation* routine, and a recursive *generate and filter* routine. Routine Generate-AndFilter operates as follows. It essentially examines all itemsets obtainable from an item first, before moving on to examine all itemsets for the second item, and so on (lines 4-9). Note that if itemsets for an item a_1 includes an item a_2, then when itemsets a_2 are generated, those that involve item a_1 will not be reexamined again. For each itemset examined, the number of occurrences is estimated using algorithm CountItemSet (see Section 2.2). Only those itemsets with estimated counts larger than or equal to the threshold will be accepted as potential frequent patterns. The patterns are accumulated as they are generated (lines 1-3). Routine InitialAndInvoke essentially performs the initialization (initially, the set of frequent patterns is an empty set, etc.) and initiates the invocation of routine GenerateAndFilter.

Routine InitialAndInvoke:

Input: set of all items $I = \{i_1,\ i_2,\ \ldots,\ i_n\}$, τ the threshold (minimum support)
Output: a superset of frequent item sets, F

    1.   *itemset* $\leftarrow$ NULL
    2. $F \leftarrow$ NULL
    3.   GenerateAndFilter(I, *itemset*)
    4.   return F

Routine GenerateAndFilter(I, *itemset*):

    1.   if (*itemset* $\neq$ NULL) {
    2. $F \leftarrow F \cup itemset$
    3.   }
    4.   while ($I \neq$ NULL) {
    5. $I \leftarrow I - \{i\}$ // i is an item from I
    6.   if CountItemSet($\{i\} \cup itemset$) $\geq \tau$ {
    7.   GenerateAndFilter(I, $\{i\} \cup itemset$)
    8.   }
    9.   }

Figure 2: Algorithm SingleFilter to generate frequent patterns.

Algorithm DualFilter

While algorithm SingleFilter is efficient, it is unable to provide any guarantee on the quality of the candidate patterns generated. As such, all candidate patterns have to be examined in the refinement step to verify their validity. Here, we propose another algorithm, algorithm DualFilter, that partitions the candidate patterns into two groups – one with 100% guarantee to be frequent patterns that will appear in the final answer set, and the other with no such guarantee. The crux of the algorithm lies in

its ability to determine, for *some* itemsets, the exact count of the number of transactions that contain the itemsets by maintaining some additional information.

We first examine the following important results:

Lemma 5 *Let $actCount(I_1)$ and $actCount(I_2)$ denote the actual counts of the number of transactions containing itemsets I_1 and I_2 respectively. Let $estCount(I_1)$ and $estCount(I_2)$ denote the estimated counts of the number of transactions containing itemsets I_1 and I_2 obtained from algorithm CountItemSet respectively. Then, if*

$$actCount(I_1) = estCount(I_1)$$

we have

$$estCount(I_1 \cup I_2) \geq actCount(I_1 \cup I_2) \geq estCount(I_1 \cup I_2) - (estCount(I_2) - actCount(I_2)).$$

Proof. *From Lemma 4, we have*

$$estCount(I_1 \cup I_2) \geq actCount(I_1 \cup I2)$$

Let $actCount(\mathcal{I})$ denote the actual counts of the number of transactions containing either itemsets I_1 or I_2, and $estCount(\mathcal{I})$ denote the estimated counts of the number of transactions containing either itemsets I_1 or I_2. From Lemma 4, we have

$$estCount(\mathcal{I}) \geq actCount(\mathcal{I})$$

We are given that

$$actCount(I_1) = estCount(I_1)$$

Thus, we have

$$estCount(\mathcal{I}) - estCount(I_1) \geq actCount(\mathcal{I}) - actCount(I_1)$$

We note that, for itemsets I_1 and I_2, we have (by set theory)

$$estCount(\mathcal{I}) - estCount(I_1) = estCount(I_2) - estCount(I_1 \cup I_2)$$

and

$$actCount(\mathcal{I}) - actCount(I_1) = actCount(I_2) - actCount(I_1 \cup I_2)$$

Therefore, we have

$$estCount(I_2) - estCount(I_1 \cup I_2) \geq actCount(I_2) - actCount(I_1 \cup I_2)$$

This implies

$$actCount(I_1 \cup I_2) \geq estCount(I_1 \cup I_2) - (estCount(I_2) - actCount(I_2)).$$

Thus, we have the lemma. □

The lemma gives the upper and lower bounds of $estCount(I_1 \cup I_2)$. In fact, we have actually used the former in algorithm SingleFilter. Now, we shall see how we can exploit the lower bound via the following corollary.

Corollary 1 *Let $actCount(I_1)$ and $actCount(I_2)$ denote the actual counts of the number of transactions containing itemsets I_1 and I_2 respectively. Let $estCount(I_1)$ and $estCount(I_2)$ denote the estimated counts of the number of transactions containing itemsets I_1 and I_2 obtained from algorithm CountItemSet respectively. Then, if*

$$actCount(I_1) = estCount(I_1)$$

and

$$actCount(I_2) = estCount(I_2)$$

we have

$$actCount(I_1 \cup I_2) = estCount(I_1 \cup I_2)$$

Proof. *Since,*

$$actCount(I_1) = estCount(I_1)$$

from Lemma 5, we obtain

$$estCount(I_1 \cup I_2) \geq actCount(I_1 \cup I_2) \geq estCount(I_1 \cup I_2) - (estCount(I_2) - actCount(I_2)).$$

Furthermore,

$$actCount(I_2) = estCount(I_2)$$

Therefore

$$estCount(I_1 \cup I_2) \geq actCount(I_1 \cup I_2) \geq estCount(I_1 \cup I_2)$$

And hence,

$$actCount(I_1 \cup I_2) = estCount(I_1 \cup I_2)$$

□

From the results, we see that all we need is to maintain the actual counts of some frequent patterns, and we can potentially determine the actual counts of subsequent patterns that contain these itemsets. For space efficiency, we only maintain the counts of all 1-itemsets.

Figure 3 shows an algorithmic description of a routine, algorithm CheckCount, that can be used to check if an itemset is frequent with 100% guarantee or frequent with some uncertainty, and the count is actual or estimated by the following values:

1. $flag = -1$

10

itemset is non-frequent

2. $flag = 0$

   itemset is frequent with some uncertainty and estimated count

3. $flag = 1$

   itemset is frequent with 100% guarantee and actual count

4. $flag = 2$

   itemset is frequent with 100% guarantee and estimated count

Algorithm CheckCount:

Input:two itemsets,$I_1(= \{i\}), I_2$,count (of the I_2), flag ,τ the threshold (minimum support)
Output: (flag,count)

Algorithm CheckCount(I_1, I_2,$count$,$flag$):

```
1.    if(I₂ = NULL) {
2.        if(actCount(I₁) < τ) return (-1,actCount(I₁)) else return (1,actCount(I₁));
3.    }
4.    if(flag = 1) {
6.        if(estCount(I₁) = actCount(I₁)&&count = estCount(I₂))
7.            return (1,estCount(I₁ ∪ I₂);     //Corollary 1
8.        else if(estCount(I₁) = actCount(I₁))&&estCount(I₁ ∪ I₂) − (estCount(I₂) − count) ≥ τ)
9.            return(2,estCount(I₁ ∪ I₂);     // The lower bound in the Lemma 5
10.       else if(estCount(I₂) = count&&estCount(I₁ ∪ I₂) − (estCount(I₁) − actCount(I₁)) ≥ τ)
11.           return(2,estCount(I₁ ∪ I₂);     // The lower bound in the Lemma 5
12 .  }
13.   return(0,estCount(I₁ ∪ I₂));
```

Figure 3: Algorithm CheckCount to determine if the actual count of an itemset can be determined.

We are now ready to look at algorithm DualFilter. The algorithmic description is shown in Figure 4. The algorithm is very similar to algorithm SingleFilter except for lines 2,7-8 of GenerateAndFilter, where we use the variable $flag$. Here, as soon as an itemset is determined as frequent (based on BBS), it is further examined to see if it is certain to be in the final answer set and its exact count can be determined. Here, we have two resultant sets − F, the set of frequent patterns that are certain to be in the final answer set, and F', the set of patterns whose validity is uncertain. We note that only patterns in F' need to be refined.

Adaptive Filtering

For large databases (with huge number of transactions), the size of BBS can be too large to fit into the memory. In this case, both algorithms SingleFilter and DualFilter are expected to perform poorly

Algorithm DualFilter:

Input: set of all items $I = \{i_1,\ i_2,\ \ldots,\ i_n\}$, τ the threshold (minimum support)
Output: F, the set of frequent patterns, and F', the set of candidate patterns with some uncertainty

    1. $itemset \leftarrow$ NULL
    2. $F \leftarrow$ NULL
    3. $F' \leftarrow$ NULL
    4. $count \leftarrow 0$
    5. $flag \leftarrow 1$
    6.   GenerateAndFilter(I, $itemset$, $count$, $flag$)
    7.   return F and F'

Routine GenerateAndFilter(I, $itemset$, $count$, $flag$):

    1.   if ($itemset \neq$ NULL) {
    2.   if($flag > 0$) $F \leftarrow F \cup itemset$ else $F' \leftarrow F' \cup itemset$
    3.   }
    4.   while ($I \neq$ NULL) {
    5. $I \leftarrow I - \{i\}$ // i is an item from I
    6.   if $CountItemSet(\{i\} \cup itemset) \geq \tau$ {
    7.   (flag,count)=CheckCount($\{i\}$, itemset,count,flag)
    8.   if($flag \geq 0$) GenerateAndFilter(I, $\{i\} \cup itemset$, $count$, $flag$)
    9.   }
   10. }

Figure 4: Algorithm DualFilter to generate frequent patterns.

because of repeated passes over BBS. To bound the I/O cost, we propose sandwiching the filtering algorithm (either SingleFilter or DualFilter) by a pre- and post- processing phases. We shall present this three-phase approach for small-memory systems (relative to database size) here.

Let DB be the database of transactions, and BBS be the corresponding bit-slice bloom-filtered signature file of DB. Let there be m bit-slices. The algorithm works as follows:

1. **Preprocessing Phase.** In this phase, we construct a new bit-slice bloom-filtered signature file for DB that fits in the memory. We shall denote this newly created structure as MemBBS. This is done by reading the first k slices of BBS that fits into memory, and "rehashing" the remaining $m - k$ slices to any of these k slices.[6]

2. **Filtering Phase.** In this phase, we can apply either algorithm SingleFilter or algorithm DualFilter on MemBBS. Let the results (i.e., the set of potential frequent itemsets) be denoted F. Clearly, the size of F generated under MemBBS will have more false drops than that under BBS. However, since MemBBS is memory-resident, the computation cost is much faster.

---

[6]An alternative approach is to create multiple BBSs with different memory requirement. At runtime, we only need to load into memory the appropriate BBS that fits in the memory. This method, however, incurs higher storage overhead as well as maintenance overhead.

3. **Postprocessing Phase.** In this phase, we want to prune away some of the frequent itemsets generated in phase 2. This can be done in a single pass over the original BBS as follows: We read sufficient vectors of BBS that fit into the memory. We estimate the counts of the itemsets in F. We repeat this process by reading the next portion of BBS, and accumulating the counts of the itemsets in F each time. When the BBS is completely scanned, we can prune away those itemsets whose counts are smaller than the minSupport threshold. The remaining itemsets in F may still contain false drops and can be further refined (see the refinement strategies in the next subsection).

We note that the three-phase scheme requires only two passes over BBS, and hence bounds the number of I/Os necessary. For simplicity, we shall not distinguish the memory-constrained versions from their memory-resident counterparts. We shall use the same name (i.e., SingleFilter or DualFilter for both versions). Essentially, whenever memory is scarce, the adaptive version will be used; otherwise the memory-resident version will be used. The context in which the specific scheme is used should be sufficient to recognize the versions used.

3.2 Refinement Algorithms

The results of the filtering phase is a set of candidate patterns that contains the actual frequent patterns. In the refinement phase, we will prune away the false drops in order to obtain the final answer set. We have identified two schemes for the refinement step:

1. **Algorithm SequentialScan.** The first scheme, algorithm SequentialScan, essentially scans the database to verify the validity of the candidate patterns. The algorithm is straightforward. We first load the memory with as many candidate patterns as can fit into the memory. Next, we scan the database and for each tuple examined, we check against the candidate patterns in the memory. This process is repeated until all the candidate patterns are examined. Since the number of candidate patterns are typically small (relative to the total number of patterns), the number of scans of the database is expected to be small.

2. **Algorithm Probe.** The second scheme, algorithm Probe, avoids scanning the database by retrieving only the relevant tuples that match each pattern. This scheme assumes that an index exists on the database. The key of the index is the relative position of the transaction from the beginning of the file. The algorithm operates as follows. Given an itemset (pattern), the resultant bit vector of CountItemSet allows us to determine the set of transactions that contribute to the count. As such, we can use the index to retrieve these records, and check if they contain the itemset.

Comparing the two schemes, we expect SequentialScan to perform well if the average estimated number of transactions containing an itemset is large. On the other hand, we expect Probe to be more efficient when the average estimated number of transactions containing an itemset is small.

3.3 The Mining Algorithms

From the above discussion, we can derive the following four filter-and-refine mining algorithms:

1. *Algorithm SFS (Single Filter, SequentialScan).* This algorithm employs algorithm SingleFilter in the filtering phase, and uses scheme SequentialScan to verify the validity of all the candidate patterns.

2. *Algorithm SFP (Single Filter, Probe).* This algorithm adopts algorithm SingleFilter in the filtering phase, and algorithm Probe in the refinement phase. However, for greater efficiency, instead of implementing them as separate phases, we integrate the two phases. In other words, as soon as SingleFilter identifies a candidate pattern, the database is probed. This implementation has two advantages. First, the initial response time is short, since as soon as a frequent pattern is identified, it can be returned to the user. Second, the scheme is more efficient as some false drops can be avoided. This is because some false drops that trigger other false drops can be identified earlier, thus the subsequent false drops (had a two-phase approach been adopted) are not triggered.

3. *Algorithm DFS (Dual Filter, SequentialScan).* This algorithm employs algorithm DualFilter in the filtering phase, and uses scheme SequentialScan to prune away the false drops. Note that unlike SFS, only those candidate patterns that have not been picked out to be in the final answer set need to be examined.

4. *Algorithm DFP (Dual Filter, Probe).* This algorithm adopts algorithm DualFilter in the filtering phase, and algorithm Probe in the refinement phase. Like SFP, we also integrate the two phases, and hence, it has the same advantages as SFP's. However, the scheme is expected to be even more efficient than SFP since we know the actual counts of all patterns that have been identified, and these can be used to identify subsequent patterns without looking up the database.

We note that SFS and DFS will have the same number of false drops. Similarly, SFP and DFP will also have the same number of false drops. Moreover, as mentioned, the probe-based schemes (i.e., SFP and DFP) will have fewer false drops that the sequential-scan-based schemes (i.e., SFS and DFS).

3.4 Discussion

In this section, we shall discuss how the proposed schemes facilitate efficient and ease of mining in two important contexts. The first context concerns a dynamic database where the database grows as more transactions or new items are added. We note that new items are only added for new transactions. The proposed schemes can be applied to dynamic databases as they are without additional complexity. Moreover, there is no need to reconstruct the BBS. For new transactions, whenever a transaction is added into the database, the corresponding bit vector of the transaction is added into BBS. For new items, since the bit vector is obtained by hashing on the items, the new items does not affect BBS either. Moreover, since bitwise operations are fast, the difference of a few bits does not affect the computation time significantly.

We note that for the FP-tree scheme, it requires reconstructing the FP-tree since the data has to be ordered. The Apriori-like schemes can be applied as they are, but with more records, it can only be more computationally expensive.

The second context concerns mining with constraints. Constraints are essentially selection predicates on the database. For example, a query of the form "Is the itemset {1,2,3} frequent during the month of October?" To answer such queries, we only need to generate a bit slice such that a bit is set if the corresponding transaction falls in the month of October and it is not set if it is not in October. The proposed schemes remain the same, except that the resultant bit slice obtained from algorithm CountItemSet is logically ANDed with the constraint bit slice. Again, this process can be done very quickly.

4 Performance Study

In this section, we present a performance study of the four filter-and-refine schemes. As comparison, we also implemented the Apriori strategy (denoted APS) and the FP-tree scheme (denoted FPS). All the experiments are performed on a 167-MHz SUN Ultra 1 with 64 megabytes main memory, running on Sun-OS. All the programs are written in C++ and compiled with g++ 2.95.2.

For the proposed strategies, the hash function used is the following: we take the four disjoint groups of bits from the 128-bit MD5 signature [12] of the item name; if more bits are needed, we calculate the MD5 signature of the item name concatenated with itself. In practice, the computational overhead of MD5 is negligible.

We evaluated the schemes on two performance metrics: false drop ratio and response time. The false drop ratio only applies to the proposed schemes and is computed as follows:

$$FDR = \frac{F_{fd}}{F}$$

where F_{fd} denotes the number of false drops, and F denote the actual number of frequent patterns. The response time of an algorithm is obtained by an average of several runs of the algorithms.

The synthetic data sets which we used for our experiments were generated using the procedure described in [1]. There are four components to a dataset – the number of transactions D, the number of distinct items in the database V, the average number of items per transaction T, and the average maximal potentially frequent item set size (I). As default, we set $D = 10K$, $V = 10K$, $T = 10$ and $I = 10$ (i.e., T10.I10.D10K with 10K items). The default value of τ(minimum support) is set to 0.3%.

4.1 Effect of Size of Bit Vector

We first study the effect of the size of a bit vector, m, on the performance of the filter-and-refine schemes. This is done by varying m from 400 to 6400 bits. Figure 5 shows the results of the experiment.

Figure 5(a) shows that as m increases, the false drop ratio FDR decreases for all the four proposed schemes. In fact, when m is small ($m \leq 1600$), increasing m reduces FDR significantly. However, as m becomes large enough ($m > 1600$), the number of false drops reduces at a very slow rate as m increases. As such, we can expect the optimal m value to be the point where the reduction starts to slow down.

From the figure, we also observe that SFP and DFP have much lower FDR than the other two schemes (i.e., SFS, DFS) without probing. Our investigation shows that the probe-based schemes have

no more than 10% of the false drops of the sequential-scan-based schemes. This result is expected since once the probe-based schemes have identified a candidate pattern, they can determine if it is frequent with 100% confidence. Hence, they are not exposed to the chain effect of false drops triggering other false drops that the sequential-scan-based schemes encountered.

Figure 5(b) shows the response time results of the four proposed schemes. Like the false drop results, when m is small, the response time of the four schemes reduce significantly as m increases. This is because as m increases, the number of false drops deceases significantly. Thus, the effort spent in removing the false drops is significantly reduced for increasing m values. However, there exists a certain value of m whereby further increase in m leads to longer response time. This is because, as mentioned in the FDR results, the number of false drops reduces at a very slow rate. As such, the vector size begins to have its effect on computational cost, albeit gradually.

We also observe that the probe-based schemes are much more efficient than their sequential-scan-based counterparts. This is clearly due to the smaller number of false drops. Comparing SFP and DFP, and SFS and DFS, we observe that the dual-filter-based schemes outperform the single-filter-based strategies. The reason is that the dual-filter-based schemes are capable of determining the frequent patterns based on Lemma 5 and Corollary 1. For example, we determine about 80% of the frequent patterns without probing the database by using the CheckCount algorithm of Figure 3 when $m = 1600$. Similarly, comparing SFS and SFP, and DFS and DFP, we note that the probe-based schemes are much faster than the sequential-scan-based schemes. As mentioned, this is due to the lower FDR of probe-based schemes.

The results of both figures imply that we can tune m to get a low FDR and fast response time. For the data sets in this paper, we shall choose $m = 1600$ as our default value.

4.2 Comparative Study on the Default Settings

In this experiment, we compare the four proposed schemes against the Apriori (APS) and FP-tree (FPS) schemes on the default parameter settings. Figure 6 shows the results of the experiment. First, we note that all the four proposed schemes outperform APS. While SFS consumes about 90% of the running time of APS, DFP's running time is less than 20% that of APS. This clearly demonstrates the effectiveness of the proposed strategies.

Second, we note that FPS remains fairly competitive - it outperforms SFS and DFS. We found that this is due to the large number of false drops for SFS and DFS.

Third, we see that SFP and DFP outperform FPS by virtue of the fact that their false drops are much lower. In particular, we see that DFP performs the best. DFP outperforms FPS since it is able to determine with certainty some patterns without having to probe the database.

4.3 Effect of Minimum Support Threshold

In this study, we would like to see how the various schemes perform as the minimum support threshold varies from 0.1% to 1.2%. The results of this experiment is shown in Figure 7. First, as the minimum support threshold increases, the response time decreases for all the schemes. This is expected since
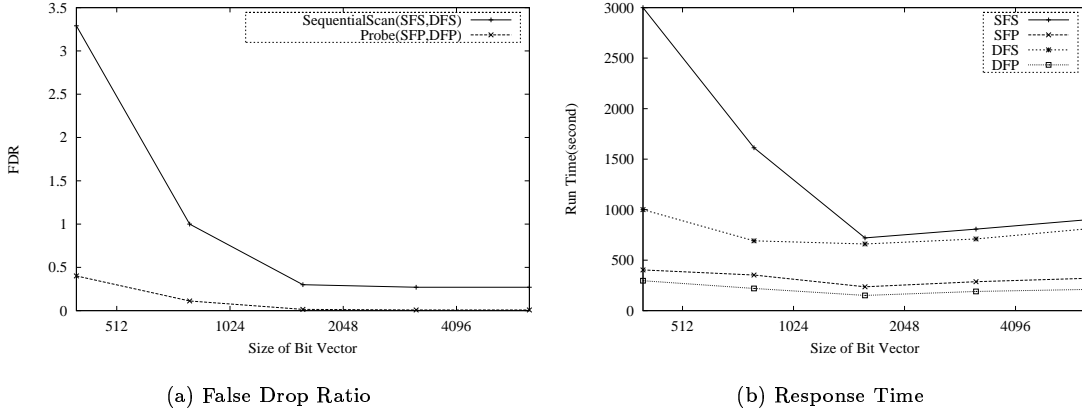
16

(a) False Drop Ratio

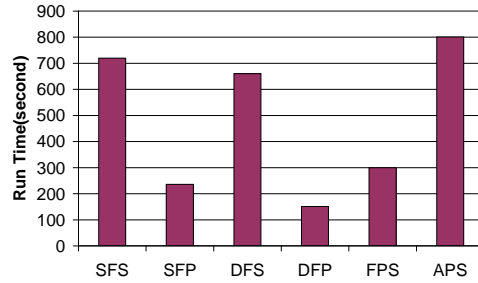(b) Response Time

Figure 5: Effect of Size of Bit Vector

Figure 6: Comparison with APS, FPS when m=1600

a larger minimum support threshold means fewer number of candidate patterns. Second, the relative performance of the various schemes remain the same: APS is the least efficient, SFS and DFS are inferior to FPS, SFP outperforms FPS, and DFP performs the best. In fact, we observe for DFP, as the minimum support threshold changes, although the number and the maximum size of frequent itemsets change dramatically, the FDR is still below 3%, and $80 - 90\%$ of the candidate frequent patterns can be determined without probing the database based on Lemma 5 and Corollary 1.

4.4   Effect of Number of Transactions

In this experiment, we evaluate how well the various schemes scale with respect to the database size. We vary the number of transactions from 10K to 100K. The results of this study is shown in Figure 8. As shown, all the schemes show linear scalability with respect to the number of transactions. However, SFP and DFP are the least affected because of the low FDR and most candidate itemsets can be determined by algorithm CheckCount based on Lemma 5 and Corollary 1. The relative performance of the schemes remains the same in the following order of efficiency: DFP (most efficient), SFP, FPS, DFS, SFS, APS (least efficient).

In summary, SFP and DFP are able to exploit the smaller candidate itemsets and be able to determine
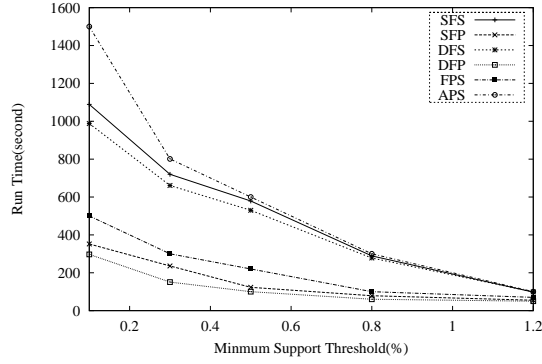
17

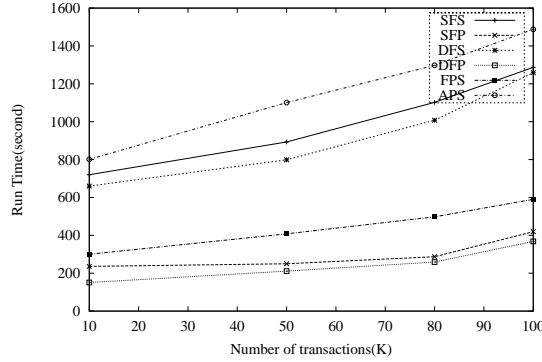Figure 7: Effect of minimum support threshold.

Figure 8: Effect of number of transactions.

most of the candidate itemsets with 100% confidence by using BBS and fast bitwise operations. Hence they are more efficient than the other schemes.

4.5 Effect of Number of Distinct Items

In this experiment, we study the effect of the number of distinct items on the various schemes. We vary V from $10K$ to $100K$ while keeping other parameters fixed.

Figure 9 shows the result of the experiment. As V increases from 10K to 100K, the number and maximum size of frequent itemsets decrease dramatically, therefore APS's response time decreases quicker than the other schemes. On the contrary, in spite of the reduction in the number of frequent itemsets, the increase of 1-frequent-itemset with the increase of V affects the structure of FP-tree, which is not a *balanced* tree. As such, the response time decreases at a slower rate. For the proposed strategies, increasing V potentially reduces the number of false drops (since m is kept constant). Thus, the response time also decreases as V increases. Since SFS and DFS have larger number of false drops, the reduction is more significant for them. Again, the relative performance of the various schemes remain the same.
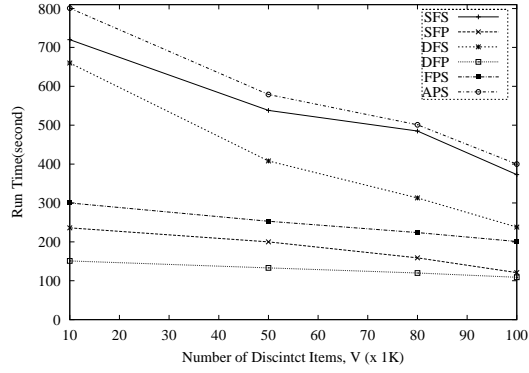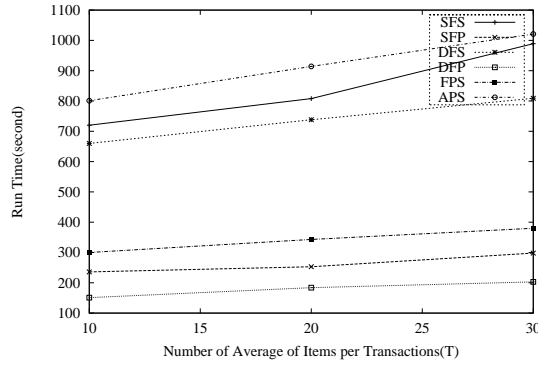
18

Figure 9: Effect of V.

Figure 10: Effect of T.

4.6 Effect of Average Number of Items Per Transaction

In this experiment, we vary the average number of items per transaction from 10 to 30 to study its effect on the schemes.

Figure 10 shows the effect of average number of items per transaction(T). When T varies from 10 to 30, the number of frequent itemsets increases, and since the support, τ, does not change, therefore the time to get the frequent patterns also increases for all the schemes. In particular, for the proposed schemes, larger T values potentially increase the number of false drops. However, as shown in the results, the proposed strategies remain effective with DFP performing the best comparing with the other schemes.

4.7 Effect of Memory Size

Since DFP performs the best among all the proposed schemes, for this and the remaining experiments, we shall just focus on DFP and ignore the other proposed algorithms. In this experiment, we study how DFP, APS and FPS perform under small memory systems. We vary the memory size from 250K to 2M. The results of this experiment is shown in Figure 11.

As shown in the figure, all schemes incur higher running time as the memory size decreases. For
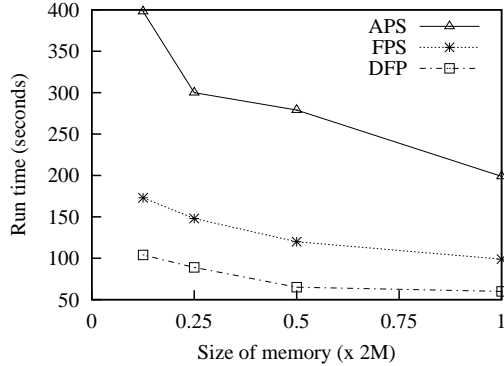
Figure 11: Effect of memory size.

DFP, this is because of the additional pre- and post- processing overheads incurred. For FPS, the size of the FP-tree is determined by the number of 1-large items. When the FP-tree does not fit into the memory, the database will have to be scanned multiple times. For APS, smaller memory means fewer data can be reused in memory, and so the database has to be scanned multiple times also. On the whole, we note that DFP remains superior over APS and FPS.

4.8   On Dynamic Databases

In this experiment, we study how DFP, APS and FPS may perform in a *dynamic* database where the database grows as more transactions or new items are added. The data set we used is the dynamic transaction database for the web server log described in [10]. The problem can be simplified as follows. There are 5000 files on the *dynamic* Web server, where 10% of the "hot" files in the previous day will be "cold" the next day. The total number of transactions is about $6,550,000$. D_0 is the transaction database in the previous day, and in the next day more records are inserted, whose database is denoted as $D_i(i = 1, \ldots, n)$.

Figure 12 shows the results. We note that DFP remains the most efficient strategy, while APS remains the worst. For DFP, the main cost, which comes from the need to update BBS, is small since new transactions are simply added into it. Evaluating a larger BBS (compared to the initial one) does not incur much higher cost because of the bitwise operations. APS has to rescan the whole database ($\sum_{i=0}^{j} D_i$, where $j = 1, \ldots, n$) and hence the increase in the mining cost (compared to DFP) is much larger. As for FPS, it has to reconstruct the FP-tree which incurs the main bulk of the additional overhead. It is also worth noting that as n gets larger (i.e., the next day's database is split into more small databases), the total difference among the three schemes will also increase.

4.9   On Ad Hoc Queries with Constraints

Queries related to frequent patterns can be answered using the frequent itemsets determined (e.g., Is itemset I frequent?). However, there are some queries that may involve more than frequent patterns, such as queries on *non*-frequent patterns, and queries with constraints. We call these queries *ad hoc*
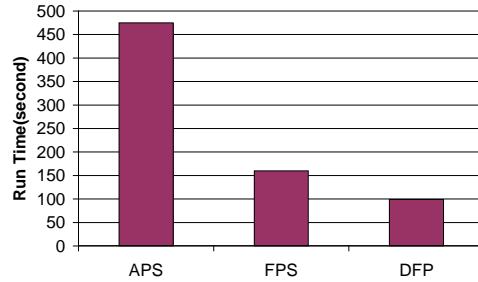
Figure 12: On Dynamic Web Log Database

queries.

In this study, we demonstrate how ad hoc queries can be addressed using two examples:

1. **Query 1.** What is the count of a particular *non*-frequent-pattern I?

2. **Query 2.** What is the number of occurrences of itemset I for transactions whose TIDs are divisible by 7?

The first query may happen in real life when users are interested in some patterns which are *not* frequent, while the second query may be a query on Sunday transactions if we suppose the TID is the number of the day since one specific Monday.

APS cannot answer these queries directly from the frequent item sets and has to rescan the database. FPS, even with the reconstruction of the FP-tree, cannot answer them because FP-tree does not store the information of the *non*-frequent patterns. Moreover, since it is not dynamic, it is hard to implement constraints in it. However, DFP can give the answers very quickly. For the first query, algorithm CountItemSet can estimate the *non*-frequent-pattern from the BBS and probe the database for the relevant transactions. For the second query, we only need an additional bit slice for the constraint[7]. The bit slice is constructed such that the corresponding bits for the transactions whose TID is divisible by 7 are set, while the other bits are not set. Figure 13 shows the results. We note that the the difference between the time to process queries 1 and 2 is negligible since the process is almost the same. As shown, DFP performs better than APS. As noted, FP-tree cannot be used to answer the queries and hence there is no performance result for FPS.

5 Conclusion

In this paper, we have proposed a dynamic and persistent data structure, called BBS, to compactly represent the original transactions as bit-vectors for facilitating efficient frequent pattern mining. Operations on BBS are efficient as they are bitwise operations. Moreover, it incurs much lesser I/Os to

---

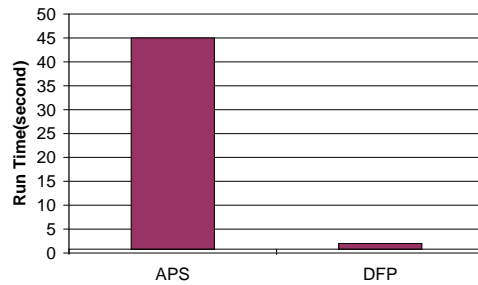[7]It is beyond our study to create the bit slices for arbitary constraints.

Figure 13: On Ad Hoc Queries 1(2)

examine bit-slices in BBS as compared to scanning the original database. Based on this structure, we have also studied four filter-and-refine algorithms. The first phase of the algorithms works on BBS to identify a superset of the candidate frequent patterns. The second phase of the algorithms refines this large set to prune away the false drops. We implemented the schemes, and compared them against the Apriori scheme and the FP-tree scheme. Extensive experimental results showed the effectiveness of the proposed schemes. Moreover, one of the schemes is shown to remain superior in all cases.

We are extending this work by exploring the possibility of doing away with phase 2. This would shorten the running time even further. However, it would mean that the answer patterns are but an approximate set of the actual answers. Such an approach will be useful if the users require only approximate answers. For the results to be meaningful, we are looking into mechanisms to provide some kind of probability on the likelihood of a pattern to be a frequent pattern.

References

[1] AGRAWAL, R., AND SRIKANT, R. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large Data Bases* (Santiago, Chile, Sept. 1994), pp. 487–499.

[2] BAYARDO, R. J., AGRAWAL, R., AND GUNOPULOS, D. Constraint-based rule mining in large, dense databases. In *Proceedings of 15th International Conference on Data Engineering* (Sydney,Australia, Mar. 1999), pp. 188–197.

[3] BLOOM, B. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM 13*, 7 (July 1970), 422–426.

[4] BRIN, S., MOTWANI, R., ULLMAN, J. D., AND TSUR, S. Dynamic intemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Tuscon,Arizona, May 1997), pp. 255–264.

[5] COHEN, E., DATAR, M., FUJIWARA, S., GIONIS, A., INDYK, P., MOTWANI, R., ULLMAN, J. D., AND YANG, C. Finding interesting associations without support pruning. In *Proceedings of 16th International Conference on Data Engineering* (San Diego, CA, Mar. 2000), pp. 489–499.

[6] FANG, M., SHIVAKUMAR, N., GARCIA-MOLINA, H., MOTWANI, R., AND ULLMAN, J. Computing iceberg queries efficiently. In *Proceedings of 24th International Conference on Very Large Data Bases* (New York City, USA, 1998), pp. 299–310.

[7] GRAHNE, H., LAKSHMANAN, L. V. S., AND WANGL, X. Efficient mining of constrained correlated sets. In *Proceedings of 16th International Conference on Data Engineering* (San Diego, CA, Mar. 2000), pp. 512–521.

[8] HAN, J., PEI, J., AND YIN, Y. Mining frequent patterns without candidata generation. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Dallas,Texas, May 2000), pp. 1–12.

[9] KLEMETTINEN, M., MANNILA, H., RONKAINEN, P., TOIVONEN, H., AND VERKAMO, A. I. Finding interesting rules from large sets of discovered association rules. In *Proceedings of CIKM'94: The 3rd ACM International Conference on Information and Knowledge Management* (1994), pp. 401–408.

[10] LAN, B., BRESSAN, S., OOI, B. C., AND TAN, K.-L. Rule-assisted prefetching in web-server caching. In *Proceedings of CIKM'00: The 9th ACM International Conference on Information and Knowledge Management* (McLean,VA,USA, Nov. 2000), pp. 504–511.

[11] LENT, B., SWAMIN, A., AND WIDOM, J. Clustering association rules. In *Proceedings of 13th International Conference on Data Engineering* (1997), pp. 220–231.

[12] MENEZES, A. J., OORSCHOT, P. C. V., AND VANTONE, S. A. *Handbook of Applied Cryptography*. CRC Press, 1997.

[13] NG, R., LAKSHMANAN, L. V. S., HAN, J., AND PANG, A. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Seattle,Washington ,USA, June 1998), pp. 13–24.

[14] PARK, J. S., CHEN, M.-S., AND YU., P. S. An effective hash-based algorithm for mining association rules. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (San Jose,CA, May 1995), pp. 175–186.

[15] ROBERTS, C. Partial-match retrieval via the method of superimposed code. *Proceeding of the IEEE 67*, 12 (1979), 1624–1642.

[16] SACKS-DAVIS, R., KENG, A., AND RAMMAMOHANARAO, K. Multi-key access methods based on superimposed coding techniques. *ACM transaction on Database systems 12*, 4 (1987), 665–696.

[17] SARAWAGI, S., THOMAS, S., AND AGRAWAL, R. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Seattle,Washington ,USA, June 1998), pp. 343–354.

[18] SAVASERE, A., OMIECINSKI, E., AND NAVATHE, S. An efficient algorithm for mining association rules in large database. In *Proceedings of 21st International Conference on Very Large Data Bases* (Zurich, Sept. 1995), pp. 432–443.

[19] SRIKANT, R., VU, Q., AND AGRAWAL, R. Mining association rules with item constraints. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining(KDD-97)* (1997), pp. 63–73.