Efficient Mining of Association Rules in Text Databases

John D. Holt and Soon M. Chung

Department of Computer Science and Engineering
Wright State University
Dayton, Ohio 45435 USA
{jholt, schung}@cs.wright.edu

Abstract

In this paper, we propose two new algorithms for mining association rules between words in text databases. The characteristics of text databases are quite different from those of retail transaction databases, and existing mining algorithms cannot handle text databases efficiently because of the large number of itemsets (i.e., words) that need to be counted. Two well-known mining algorithms, Apriori algorithm and Direct Hashing and Pruning (DHP) algorithm, are evaluated in the context of mining text databases, and are compared with the new proposed algorithms named Multipass-Apriori (M-Apriori) and Multipass-DHP (M-DHP). It has been shown that the proposed algorithms have better performance for large text databases.

1 Introduction

Mining association rules in transaction databases has been demonstrated to be useful and technically feasible in several application areas [2, 3], particularly in retail sales. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of items. Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items, such that $T \subseteq \mathcal{I}$. An association rule is an implication of the form $X \Rightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \phi$. The association rule $X \Rightarrow Y$ holds in the database $\mathcal{D}$ with *confidence* c if $c\%$ of transactions in $\mathcal{D}$ that contain X also contain Y. The rule $X \Rightarrow Y$ has *support* s if $s\%$ of transactions in $\mathcal{D}$ contain $X \cup Y$. Mining association rules is to find all association rules that have support and confidence greater than or equal to the user-specified minimum support (called *minsup*) and minimum confidence (called *minconf*), respectively [1]. For example, beer and disposable diapers are items such that *beer* $\Rightarrow$ *diapers* is an association rule mined from the database if the co-occurrence rate of beer and disposable diapers (in the same transaction) is not less than *minsup* and the occurrence rate of beer in the transactions containing diapers is not less than *minconf*.

The first step in the discovery of association rules is to find each set of items (called *itemset*) that have co-occurrence rate above the minimum support. An itemset with at least the minimum support is called a *large itemset* or a *frequent itemset*. In this paper, the term *frequent itemset* will be used. The size of an itemset represents the the number of items contained in the itemset, and an itemset containing k items will be called a k-itemset. For example, {beer, disposable diapers} can be a frequent 2-itemset. Finding all frequent itemsets is very resource consuming task and has received a considerable amount of research effort in recent years. The second step of forming the association rules from the frequent itemsets is straightforward as described in [1]: For every frequent itemset f, find all non-empty subsets of f. For every such subset a, generate a rule of the form $a \Rightarrow (f - a)$ if the ratio of support$(f - a)$ to support(a) is at least *minconf*.

The association rules mined from point-of-sale (POS) transaction databases can be used to predict the purchase behavior of customers. In the case of text databases, there are several uses of mined association rules. The association rules for text can be used for building a statistical thesaurus. Consider the case that we have an association rule, $B \Rightarrow C$, where B and C are words. A search for documents containing C can be expanded by including B. This expansion will allow for finding documents using C that do not contain C as a term. A closely related use is Latent Semantic Indexing, where documents are considered *close* to each other if they share a sufficient number of associations [4]. Latent Semantic Indexing can be used to retrieve documents that do not have any terms in common with the original text search expression by adding documents to the query result set that are *close* to the documents in the original query result set.

The word frequency distribution of a text database can be very different from the item frequency distribution of a sales transaction database. Additionally, the

CIKM '99 11/99 Kansas City, MO, USA

number of unique words in a text database is significantly larger than the number of items in a transaction database. Finally, the number of unique words in a typical document is much larger than the number of unique items in a transaction. These differences make the existing algorithms, such as the Apriori [1] and Direct Hashing and Pruning (DHP) [5], ineffective in mining association rules in the text databases.

Two new algorithms suitable for the mining association rules in text databases are proposed in this paper. These algorithms are named Multipass-Apriori (M-Apriori) and Multipass-DHP (M-DHP), respectively, and are described in Section 4. The results of the performance analysis are discussed in Section 5. The new algorithms demonstrated significantly better performance than Apriori and DHP algorithms for large text databases.

2 Text Databases

Traditional domains for finding frequent itemsets, and subsequently the association rules, include retail point-of-sale (POS) transaction database and catalog order database [2]. The natural item instances are the sales transaction items, but other item instances are possible. For example, individual customer order histories could be used. An item may have a detailed identity, such as a particular brand and size, or may be mapped to a generic identity such as "bread". The number of items in a typical POS transaction is well under a hundred. The mean number of items and distribution varies considerably depending upon the retail operation. In the referenced papers that provided experimental results, the number of items per transaction ranged from 5 to 20.

The word distribution characteristics of text data present some scale challenges to the algorithms that are typically used in retail sales mining. A sample of text documents was drawn from the 1996 TReC [9] data collection. The sample consisted of the April 1990 Wall Street Journal articles that were in the TReC collection. There were 3,568 articles and 47,189 unique words. Most of those words occur in only a few of the documents. Some of the key distribution statistics are:

48.2%	of the words occur in only one document;
13.4%	of the words occur in only two documents;
7.0%	of the words occur in only 3 documents.

The mean number of unique words in a document was 207, with a standard deviation of 174.2 words. In this sample, only 6.8% of the words occurred in more than 1% of the documents. A single day sample of January 21, 1990 was taken as well. In that sample, there were 9,830 unique words, and 78.3% of the words occurred in three or fewer documents.

The characteristics of this word distribution have profound implications for the efficiency of association rule mining algorithms. The most important implications are: (1) the large number of items and combinations of items that need to be counted; and (2) the large number of items in each document in the database.

It is commonly recognized in the information retrieval community that words that appear uniformly in text database have little value in differentiating documents, and further those words occur in a substantial number of documents [6]. It is reasonable to expect that frequent itemsets composed of highly frequent words (typically considered to be words with occurrence rates above 20%) would also have little value. Therefore, text database miners will need to work with itemsets composed of words that are not too frequent, but are frequent enough. The range of minimum and maximum support suitable for word association mining is not known at this time. However, it is clear that word association mining will require using minimum support levels that are significantly lower than the ones typically used for POS transaction databases.

The relatively low minimum support required for text database mining exacerbate the problems caused by the word frequency distribution. In text documents, the preponderance of the words are of moderate or low frequency and these words are precisely the words of interest for finding frequent itemsets.

It is not yet clear how low the minimum support should be for finding effective associations. However, experimental results suggest that it should be lower than 0.5% for the April 1990 Wall Street Journal collections.

In the April 1990 Wall Street Journal articles in the TReC collection are six groups of four or more very closely related documents. There should be a significant number of frequent itemsets that are in common for a majority of the documents within each group. Using the frequent itemsets with a minimum support level of 0.5%, only one group had a frequent itemset that was common for more than 50% of the documents in that group. Thus, it is clear that the threshold must be lower than 0.5%. A lower minimum support of 0.1% was selected, because in the April 1990 collection, that threshold represents the four documents level and four documents is the smallest size of any of the six groups.

3 Existing Algorithms for Mining Association Rules

Algorithms for finding frequent itemsets make multiple passes over the data. In the first pass, the support of individual items are counted and frequent 1-itemsets are determined. Then the frequent 1-items are used to generate the potentially frequent 2-itemsets, called candidate 2-itemsets. In the second pass, we count

the support of the candidate 2-itemsets, so that we can determine the frequent 2-items. Frequent 2-itemsets are used to generate the candidate 3-itemsets, and so on. This process is repeated until there is no new frequent itemset. There are both sequential and parallel algorithms that have been developed for finding frequent itemsets in transaction databases. The four algorithms described below have both a sequential and a parallel implementation. The parallel implementation is a trivial change from the base sequential implementation.

These algorithms are all implemented as a process which interacts with a local data collection. In the parallel versions, the data collection is partitioned into a set of sub-collections each of which is local to a particular processor. In a later section, there is a brief review of alternative strategies that require all or a substantial portion of the data to be moved from the local environment to other environments for parallel processing. These methods were not pursued because of the substantial amount of data movements and replication required. A text collection local to a processor could be multiple gigabytes in size, thus it is not suitable for transmission and replication among tens or hundreds of processors.

3.1 Apriori Algorithm

The Apriori algorithm proposed by Agrawal and Srikant [1] for finding frequent itemsets where the input data consists of transactions is as follows:

```
1)  Database = set of transactions;
2)  Items = set of items;
3)  transaction = ⟨TID, {x | x ∈ Items}⟩;
4)  Comment: F1 is a set of frequent 1-itemsets
5)  F1 = φ;
6)  Comment: Read the transactions and count
                 the occurrences of each item
7)  foreach transaction t ∈ Database do begin
8)     foreach item x in t do
9)        x.count++;
10) end
11) Comment: Form the set of frequent 1-itemsets
12) foreach item i ∈ Items do
13)    if i.count/|Database| ≥ minsup
14)       then F1 = F1 ∪ i;
15) Comment: Find Fk, the set of frequent k-itemsets,
                 where k ≥ 2
16) for (k := 2; Fk-1 ≠ φ; k++) do begin
17)    Comment: Ck is the set of candidate k-itemsets
18)    Ck = φ;
19)    Comment: Fk-1 * Fk-1 is a natural join of
             Fk-1 and Fk-1 on the first k − 2 items
20)    foreach x ∈ {Fk-1 * Fk-1} do
21)       if ¬∃y | y = (k − 1)-subset of x ∧ y ∉ Fk-1
22)          then Ck = Ck ∪ x;
23)    Comment: Scan the transactions to count
                 candidate k-itemsets
24)    foreach transaction t ∈ Database do begin
25)       foreach k-itemset x in t do
26)          if x ∈ Ck
27)             then x.count++;
28)    end
29)    Comment: Fk is the set of frequent
                 k-itemsets
30)    Fk = φ;
31)    foreach x ∈ Ck do
32)       if x.count/|Database| ≥ minsup
330)         then Fk = Fk ∪ x;
34) end
35) Answer = ∪k Fk;
```

The formation of the set of candidate itemsets can be done effectively when the items in each itemset are stored in a lexical order, and itemsets are also lexically ordered. As specified in line 20, candidate k-itemsets, for $k \geq 2$, are obtained by performing the natural join operation $F_{k-1} * F_{k-1}$ on the first $k-2$ items of F_{k-1} assuming that the items are lexically ordered in each itemset [1]. For example, if F_2 includes {A, B} and {A, C}, then {A, B, C} is a potential candidate 3-itemset. Then the potential candidate k-itemsets are pruned in line 21 by using the property that all the $(k-1)$-subsets of a frequent k-itemset should be frequent $(k-1)$-itemsets. This property is *subset closure* property of the frequent itemset. Thus, for {A, B, C} to be a candidate 3-itemset, {B, C} also should be a frequent 2-itemset. This pruning step prevents many potential candidate k-itemsets from being counted in each pass k for finding frequent k-itemsets, and results in a major reduction in memory consumption. To count the occurrences of the candidate itemsets efficiently as the transactions are scanned, they can be stored in a hash tree, where the hash value of each item occupies a level in the tree [1, 5].

The Apriori algorithm can be parallelized in a trivial manner by simply distributing the transactions to processors and sharing the counts of itemsets at the end of each pass. A large scale text collection is generally distributed for search and retrieval performance. Unfortunately, the amount of text that is allocated to a processor can be still too large and can generate a very large number of candidate itemsets for the Apriori algorithm. Recall the distribution characteristics of text collections discussed in Section 2. There will be a considerable number of candidate itemsets that do not have the minimum support. In the April 1990 Wall Street Journal data, there are approximately 15,000 words that occur more than 0.1% of the documents. With Apriori, approximately 112 million candidate 2-itemsets would be generated.

3.2 Direct Hashing and Pruning (DHP) Algorithm

In Direct Hashing and Pruning (DHP) algorithm, hashing technique is used to filter out unnecessary itemsets for the generation of the next set of candidate itemsets [5]. Each $(k+1)$-itemset in transactions is hashed to a hash value while the occurrences of the candidate k-itemsets are counted by scanning the transactions. If the support count of a hash value is less than the minimum support, then all the $(k+1)$-itemsets with that hash value will not be included in the set of candidate $(k+1)$-itemsets in the next pass. Pruning candidate itemsets based upon the support counts of their hash values is safe because there may be false positives (i.e., the retained candidate itemsets that are not actually frequent) but there will be no false negatives.

Transaction pruning and transaction trimming methods are also proposed in DHP [5], so that the size of database to be scanned to count the occurrences of candidate itemsets is reduced at each pass. Transaction pruning and trimming are based on the subset closure property of frequent itemsets; that is, any subset of a frequent itemset must be a frequent itemset by itself. This property suggests that a transaction may have a candidate $(k+1)$-itemset only if it contains $(k+1)$ candidate k-itemsets obtained in the previous pass. Thus, as a transaction is scanned to count the occurrences of the candidate k-itemsets, we can determine if this transaction can be pruned from the database in the next pass. On the other hand, if a transaction contains a frequent $(k+1)$-itemset, any item contained in this $(k+1)$-itemset should appear in at least k of the candidate k-itemsets contained in this transaction. Thus, by counting how many times each item in a transaction is involved in the candidate k-itemsets in that transaction, we can decide whether the item can be eliminated from the transaction in the next pass. A transaction is trimmed by rewriting it without the items that will not contribute to forming the frequent itemsets in the next pass.

To realize Direct Hashing and Pruning (DHP), a hash table needs to be implemented. In each pass k, a hash table is created to count the hash values of $(k+1)$-items in the transactions. The hash table implemented as an object has the following methods:

- *add(itemset)* to increment the count of the occurrence of the hash value of an itemset.
- *prune(minsup)* to remove hash values that lack sufficient support.
- *hasSupport(itemset)* to see if the hash value of an itemset remain in the hash table after pruning.

The DHP algorithm is as follows:

```
1)  Database = set of transactions;
2)  Items = set of items;
3)  transaction = ⟨TID, {x | x ∈ Items}⟩;
4)  Comment: F1 is a set of frequent 1-itemsets
5)  F1 = φ;
6)  Comment: H2 is the hash table for 2-itemsets
7)  Comment: Read the transactions, and count the
        occurrences of each item, and generate H2
8)  foreach transaction t ∈ Database do begin
9)      foreach item x in t do
10)         x.count++;
11)     foreach 2-itemset y in t do
12)         H2.add(y);
13) end
14) Comment: Form the set of frequent 1-itemsets
15) foreach item i ∈ Items do
16)     if i.count/|Database| ≥ minsup
17)         then F1 = F1 ∪ i;
18) Comment: Remove the hash values without the
                minimum support
19)     H2.prune(minsup);
20) Comment: Find Fk, the set of frequent k-itemsets,
                where k ≥ 2
21) foreach (k := 2; Fk-1 ≠ φ; k++) do begin
22)     Comment: Ck is the set of candidate k-itemsets
23)     Ck = φ;
24)     Comment: Fk-1 * Fk-1 is a natural join of
                Fk-1 and Fk-1 on the first k − 2 items
25)     Comment: Hk is the hash table for k-itemsets
26)     foreach x ∈ {Fk-1 * Fk-1} do
27)         if Hk.hasSupport(x)
28)             then Ck = Ck ∪ x;
29)     Comment: Scan the transactions to count
                candidate k-itemsets and generate Hk+1
30)     foreach transaction t ∈ Database do begin
31)         foreach k-itemset x in t do
32)             if x ∈ Ck
33)                 then x.count++;
34)         foreach (k+1)-itemset y in t do
35)             if ¬∃z | z = k-subset of y ∧
                    ¬Hk.hasSupport(z)
36)                 then Hk+1.add(y);
37)     end
38)     Comment: Fk is the set of frequent
                    k-itemsets
39)     Fk = φ;
40)     foreach x ∈ Ck do
41)         if x.count/|Database| ≥ minsup
42)             then Fk = Fk ∪ x;
43) Comment: Remove the hash values without the
                minimum support from Hk+1
44) Hk+1.prune(minsup);
45) end
46) Answer = ∪k Fk;
```

There are few changes made to the Apriori algorithm to convert it to the DHP algorithm. In the initial pass, to count the occurrences of the hash values of 2-itemsets in each transaction, *add*() function is added at line 12. Pruning the hash table using the *prune*() function in preparation for the next pass is performed in lines 19 and 44. Candidate itemsets are removed if their hash values don't have the minimum support, as shown in lines 26–28. A $(k+1)$-itemset in a transaction is added to the hash table H_{k+1} if the hash values of all the k-subsets of the $(k+1)$-itemset have the minimum support in H_k, as shown in lines 34–36.

How much the DHP can reduce the number of candidate itemsets depends on the number of false positives. The false positives are generated when the hash values are identical for a group of candidate itemsets (the hash synonyms) whose individual frequency is less than minimum support count, but whose hash value frequency is not less than the threshold. The number of candidate itemsets that have the same hash value is directly related to the size of the hash table. Unfortunately, this table is in competition for memory space with the hash tree used to hold the counts for the itemsets.

The DHP algorithm is not suitable for finding the frequent itemsets in the April 1990 Wall Street Journal articles (see Section 2 for details on the collection) with the required minimum support level of 0.1%. The reason is that there are 47,000 unique words, so that the number of 2-itemsets to be hashed is about 2.2 billion. This is because the hashing of the 2-itemsets is performed before the pruning of the single items. With a minimum support count of four occurrences (0.1% minimum support), if every 2-itemset actually occurred in a document, the hash table would have to accommodate more than 500 million entries to avoid counting every 2-itemset, because the expected number of itemsets with the same hash value would be slightly more than four. For this collection of text data, a hash table of 10 million entries resulted in no pruning of 2-itemsets when the minimum support is 0.1%. There was not enough memory available to try a significantly larger hash table.

3.3 Partition Algorithm

The advantage of holding the database in memory and thereby avoiding disk I/O operations motivated Savasere, Omiecinski, and Navathe [7] to propose the algorithm Partition.

In both Apriori and DHP (see Section 3.1 and Section 3.2, respectively), there are repeated passes of the database to find frequent itemsets of different sizes. The upper bound on the passes may be the maximum size of the frequent itemsets desired, or when there is no candidate itemset for a pass. Recall that there can be no frequent $(k+1)$-itemset without all of its k-subsets being frequent. In contrast, Partition algorithm passes the database only two times.

The database is partitioned into as many partitions as are required so that all of the transactions in each partition fit in the main memory. In the first pass, each partition is mined independently to find all the frequent itemsets in the partition. Then the frequent itemsets of all the partitions are merged to generate a set of all candidate itemsets. In the second pass, all the partitions are processed against the candidate itemsets found in the first pass, and their exact occurrence counts are tabulated.

Assume that there are m partitions. If the minimum support count for itemsets in the database is s, then the minimum support count of itemsets in a partition is s/m. It is clear that in order for an itemset to have the support count s, it should have the support count not less than s/m in at least one of the m partitions. Notice that the subset closure property of frequent itemsets is quite helpful. If an itemset doesn't have the partition level support, then all of its extensions will not have the partition level support, so that none of those itemsets need be counted within the partition.

The second pass is required to weed out the false positives. Once the first pass is complete, all the partitions have been processed, and there is a single set of candidate itemsets. Some of these candidate itemsets may be false positives, but there is no false negative. In the second pass, the occurrence counts for each candidate itemsets are tabulated. Those itemsets whose occurrence counts are not less than the minimum support count are retained, but the others are false positives and discarded.

The first pass of the Partition algorithm can be easily parallelized because each partition can be processed independently. The second pass also can be parallelized by simply exchange the counting information before the beginning and at end of the second pass.

For each item in a partition, we maintain a list of transaction identifiers (TIDs) of the transactions containing the item. The lists of TIDs are used to determine how many transactions have a particular itemset. For example, let A and B be two items, each with a list of TIDs. Then the support count for {A, B} is simply the number of TIDs that are common in both lists.

This partition approach is not particularly suitable for a text collection. Consider the distribution of words in the TReC collection of April 1990 Wall Street Journal articles. To apply a minimum support of 1%, which is corresponding to appearing in 36 documents, the number of partitions should be much less than 36. Otherwise, we cannot distinguish a word that occurs in only one document from the one that occurs in 1% of the documents, i.e. 36 documents. When the number

of partitions is small, the size of each partition becomes large and a partition may not fit into the main memory. Anyway, since nearly 70% of the words occur in three or fewer documents, there will be a considerable number of false positives.

3.4 Sampling Algorithm

The idea of Sampling algorithm is to pick a small sample of the database and find all the itemsets in the sample that are potentially frequent in the whole database [8]. Then, the whole database is scanned to actually count those itemsets. Thus, compared to other mining algorithms, we can reduce the number of passes on the database. The Sampling algorithm proceeds as follows:

1. A sample of a size appropriate to the task is drawn from the database. For an itemset, its support in the whole database can be different from its support in the sample. If we consider this difference as an error, the sample size can be determined based on the maximum probability that this error is greater than certain value. (see [8] for more details.)

2. The sample is processed to discover the itemsets whose support in the sample is not less than a threshold value. This threshold value is chosen to be smaller than *minsup*, so that we can discover most of the potentially frequent itemsets.

3. For the set S of frequent itemsets discovered from the sample, determine the *negative border* of S, denoted by $NB(S)$. The negative border is the set of minimal itemsets in the database that are not in S. Thus, each itemset in $NB(S)$ is not in S, whereas all of its proper subsets are in S.

4. Read the database and count the occurrences of each itemset in $S \cup NB(S)$. Then, include only the itemsets whose support is not less than *minsup* into the set F.

5. If all the itemsets in F are also in S, then F includes all the frequent itemsets, and the algorithm is complete. However, if some itemset in F is in $NB(S)$, we may have some missing frequent itemsets. In this case, proceed to the next step.

6. repeat
   compute $F = F \cup NB(F)$
   until F does not grow;

7. Read the database to count the occurrences of each itemset in F, and keep only the itemsets whose support is not less than *minsup*. Now F includes all the frequent itemsets of the database.

Like the Partition algorithm, Sampling algorithm also has difficulties in mining text databases due to the frequency distribution of words. In the Partition algorithm the difficulty is too many false positives or too few partitions. In Sampling, the problem is too many false negatives, that is, too many single items in the negative border that must be counted both individually and extended with themselves as well as with the frequent itemsets.

3.5 Other Methods

A clear alternative to partitioning the documents or transactions is the partitioning of the itemsets to be counted. For a parallel implementation, each partition of itemsets is allocated to a processing node and the database is replicated at each node. However, the replication need not be complete. Zaki et al. [10] proposed to transmit to each node only those transactions that contain the items that are members of the itemsets assigned to the node. Further, the itemsets can be clustered so that the number of replicated transactions is minimized.

These methods were not pursued because they all require data movement and at least some data replication. With the size of commercial text collections in the range of hundreds to thousands of gigabytes, replication and data movement are not viable options.

3.6 Motivation for New Mining Algorithms

Each of the existing algorithms discussed above was shown to be unsuitable for the task of mining frequent itemsets from text databases. The unsuitability was a consequence of not being capable of handling the large number of potential frequent itemsets in an effective manner. The requirement of mining frequent itemsets from a database with a large number of candidate itemsets motivates the development of new mining algorithms

4 Multipass-Apriori and Multipass-DHP Algorithms

The Multipass-Apriori (M-Apriori) and the Multipass-DHP (M-DHP) algorithms for mining association rules are direct descendent of the Apriori and DHP algorithms discussed in Section 3.1 and Section 3.2, respectively. Both Apriori and DHP are not suitable for mining frequent itemsets in text databases because of the high memory space requirement for counting the occurrences of large number of potential frequent itemsets. The Multipass approach directly reduces the required memory space by partitioning the frequent 1-itemsets, and processes each partition separately. Each partition of items contains a fraction of the set of all items in the database, so that the memory space required for counting the occurrences of the sets of items within a partition will be much less than the case of counting the occurrences of the sets of all the items in the database.

The M-Apriori and M-DHP algorithms are described as follows:

1. Count the occurrences of each item in the database to find the frequent 1-itemsets.

2. Partition the frequent 1-itemsets into p partitions, $P_1, P_2, \ldots, P_p$.

3. Use Apriori or DHP algorithm to find all the frequent itemsets in each partition, in the order of $P_p, P_{p-1}, \ldots, P_1$, by scanning the database. When partition P_p is processed, we can find all the frequent itemsets whose member items are in P_p. When the next partition P_{p-1} is processed, we can find all the frequent itemsets whose member items are in P_{p-1} and P_p. This is because, when P_{p-1} is processed, the frequent itemsets we found from P_p are extended with the items in P_{p-1} and then counted. This procedure is continued until P_1 is processed.

Assume, without loss of generality, that the items are ordered lexically. The items are partitioned into p partitions, $P_1, P_2, \ldots, P_p$, such that for every $i < j$, every item $a \in P_i$ is less than every item $b \in P_j$. Thus the itemsets under consideration for some partition P_i have a particular range of item prefixes. Notice that if the partitions have the same number of items, the potential number of itemsets that will be formed by expanding a lexically lower ordered partition will be larger than the potential number of itemsets from a lexically higher ordered partition.

Since the items are ordered lexically, it is important to process the partitions in sequence from the highest ordered one to the lowest ordered one. This processing order is required to support the pruning of candidate itemsets based on the subset closure property of frequent itemsets. Figure 1 shows an example of partitions, where items 0, 1, 2, 3, 4, and 5 are frequent 1-itemsets and are partitioned into P_1, P_2, and P_3.

For example, if an itemset $\{2,3\}$ is found frequent as a result of processing the items in partition P_3, then we may count the itemset $\{1,2,3\}$ when the partition P_2 is being processed. When the itemset $\{1,2,3\}$ is considered for a potential candidate itemset, the occurrence count of the itemset $\{2,3\}$ is already available, because all itemsets beginning with item 2 were counted when partition P_3 was processed.

In practice, if the estimated number of candidate itemsets to be generated is small after processing a certain number of partitions, then we can merge the remaining partitions into a single partition, so that we can reduce the number of database scanning.

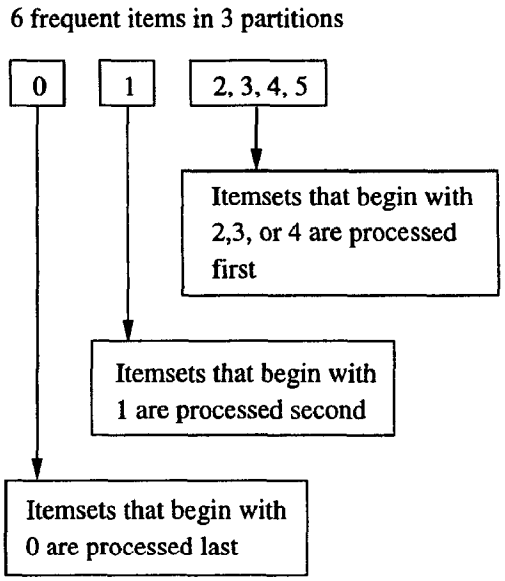

Figure 1: Partitioning a set of 6 frequent items for M-Apriori and M-DHP

5 Performance Analysis of M-Apriori and M-DHP

Some performance tests have been done with M-Apriori and M-DHP. The first objective was to assess the effect of the hash table on the performance. The second objective was to assess whether the multipass approach would improve performance. To meet these objectives, we studied the performance of four miners, Apriori, DHP, M-Apriori, and M-DHP. All four of these miners were derived from the same code base. All of the test runs were made on a 400 MHz Pentium-II machine with 384 Mbytes of memory. All of the miners were written in Java, and the IBM 1.1.7A JVM was used. The JVM memory for objects was constrained (via the *mx* parameter) to 256 Mbytes. The partition size used for the M-Apriori and M-DHP was 100 items, and the hash table size for M-DHP and DHP was 50,000 entries.

The performance measurements were taken in different data contexts. The contexts varied by the number of documents in the database and the number of items with sufficient support. Three databases were used for the experiments.

A small data collection, one day of the Wall Street Journal with 182 documents in 596 Kbytes, was used for the first test case. In this test the minimum support was 5%. There were 9,825 items with 769 of them being frequent items. Three passes were made against this data collection, and 5,231 frequent 2-itemsets and 3,521 frequent 3-itemsets were found.

A medium sized data collection, one week of the Wall Street Journal with 838 documents in 2.7 Mbytes, was used for the second test case. In this test the minimum support was 2.5%. There were 22,712 items with

1,463 of them being frequent items. Two passes were made against this data collection and 30,235 frequent 2-itemsets were found.

A large data collection, two weeks of the Wall Street Journal with 1,738 documents in 5.7 Mbytes, was used for the third test case. In this case, the minimum support was also 2.5%. There were 32,541 items with 1,427 of them being frequent items. Three passes were made against this data collection, and 24,791 frequent 2-itemsets and 22,715 frequent 3-itemsets were found.

In Figure 2, we find that for a small number of text documents, Apriori strongly outperforms DHP, and that M-Apriori and M-DHP perform at a similar level as Apriori. The overhead of the hash table used in DHP is not offset by the efficiency gained from the filtering of candidate itemsets. In Apriori, there was a combined total of 314,067 candidate itemsets counted in all three passes. DHP and M-DHP counted 314,060 candidate itemsets, which means that the hash table was not effective. M-Apriori takes almost the same time as Apriori because the time consumed by reading the database additional times for different partitions of items is not offset by the reduction in memory use.

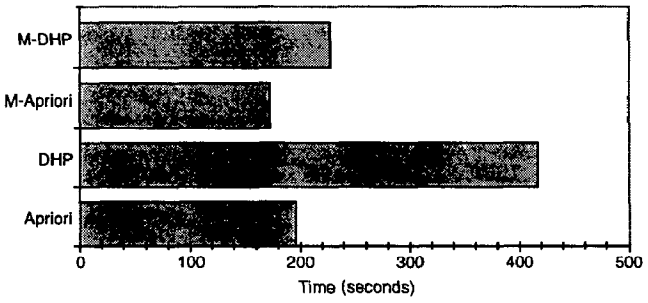

Figure 2: 182 documents and 769 frequent items

In Figure 3, we see that the effect of the hash table and performing multiple passes becomes apparent. In this situation, the overhead of using the hash table in DHP is completely offset by the savings generated by counting fewer itemsets. There are enough frequent itemsets, so that the memory savings generated by performing multiple passes also has a positive impact. Notice that M-Apriori performs better than DHP. This should not be surprising, because the multiple pass approach has a much better opportunity to reduce memory consumption than DHP alone.

In Figure 4, we can see that with a much larger set of documents, M-DHP slightly outperforms M-Apriori. M-Apriori takes about 15% longer to execute than M-DHP. M-Apriori counted a combined total of 1,939,868 candidate itemsets and M-DHP counted 1,919,465 candidate item sets which is about 20,000 fewer candidate itemsets. A hash table of only 50,000

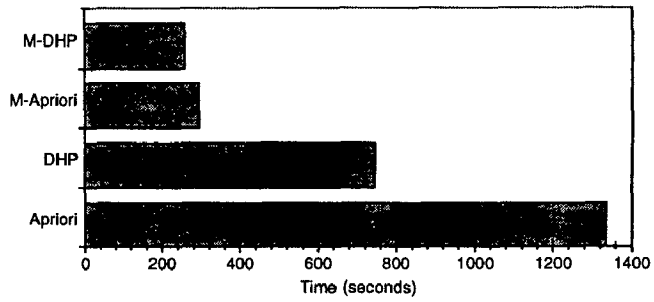

Figure 3: 838 documents and 1,463 frequent items

entries is not large enough to enable significant filtering of the candidates in this case.

Both Apriori and DHP couldn't run successfully against the database of 1,738 documents because they exceeded the JVM memory limit of 256 Mbytes. The reason for this is the large number of candidate itemsets generated. Recall that there are more than 1.9 million candidate 2-itemsets to be counted in the second pass. Both Apriori and DHP require all of those candidate itemsets to be in memory during the pass.

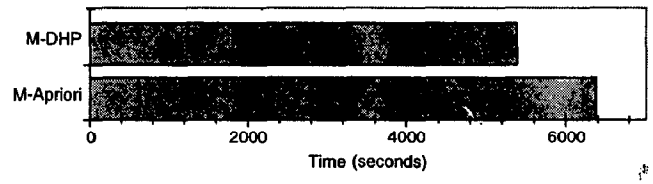

Figure 4: 1,738 documents and 1,427 frequent items

Our performance analyses show that the multipass approach can be effective even though the amount of data is not very large. For example, Figure 3 shows the case of 2.7 Mbytes of data.

The key performance differentiation appears to be the reduction in the amount of required memory space. The multipass approach directly reduces the number of objects in memory, and hence reduce the memory management overhead. These results suggest that memory management overhead is one of the dominant performance factors.

6 Conclusions

The main conclusions that can be drawn from this study are centered around the nature of the databases and the use of the mined association rules. The distribution of words in text document collections and the number of unique words in a document make the problem of finding frequent itemsets (i.e., sets of words) in text databases very different from the problem of finding frequent itemsets in traditional point-of-sale transaction databases. The differences

in distribution characteristics between text databases and transaction databases motivate different mining algorithms to handle the text database. The first difference is that the candidate itemsets must be formed in a lazy manner. There are simply too many combinations of items that do not occur in the documents, hence the computation of the candidate itemsets in advance of the counting pass is not a worthwhile approach. The second difference is that shear number of potential frequent itemsets to be counted requires that there should be some way to divide the work of counting such that only a limited number of itemsets are considered at a time.

The association rules mined from text databases are used in a quite different manner compared to those from transaction databases. This difference in usage necessitates much lower minimum support level for the association rules in text databases. It can be so low that the traditional approach of partitioning the database may not work well. The reason is because, at the partition level, it is not possible to distinguish between an item that occurs just once in the complete database and an item that occurs just once in the partition but a sufficient number of times in the database.

References

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. of the 20th VLDB Conf.*, 1994, pp. 487–499.

[2] S. Brin, R. Motwani, J. Ullman, and S. Tsur, "Dynamic Itemset Counting and Implication Rules for Market Basket Data," *Proc of the ACM SIGMOD Int'l Conf. on Management of Data*, 1997, pp. 255–264.

[3] M. S. Chen, J. Han, and P. S. Yu, "Data Mining: An Overview from a Database Perspective," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 8, No. 6, Dec. 1996, pp. 866–883.

[4] M. Gordon and S. Dumais, "Using Latent Semantic Indexing for Literature Based Discovery," *Journal of the Amer. Soc. of Info Science*, Vol. 49, No. 8, June 1998, pp. 674–685.

[5] J. S. Park, M. S. Chen, and P. S. Yu, "Using a Hash-Based Method with Transaction Trimming for Mining Association Rules," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 9, No. 5, Sep/Oct 1997, pp. 813–825.

[6] G. Salton, *Automatic Text Processing: the transformation, analysis, and retrieval of information by computer*, Addison-Wesley Publishing, 1988.

[7] A. Savasere, E. Omiecinski, and S. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases," *Proc. of the 21st VLDB Conf.*, 1995, pp. 432–444.

[8] H. Toivonen, "Sampling Large Databases for Association Rules," *Proc. of the 22nd VLDB Conf.*, 1996, pp. 134–145.

[9] E.M. Voorhees and D.K. Harmon (editors), *The Fifth Text Retrieval Conference*, National Institute of Standards and Technology, 1997.

[10] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New Algorithms for Fast Discovery of Association Rules," *Technical Report 651*, Computer Science Department, University of Rochester, July 1997.