Fast Vertical Mining Using Diffsets

Mohammed J. Zaki *
Department of Computer Science
Rensselaer Polytechnic Institute, Troy, NY
zaki@cs.rpi.edu

Karam Gouda
Department of Mathematics
Faculty of Science, Benha, Egypt
karam_g@hotmail.com

ABSTRACT

A number of vertical mining algorithms have been proposed recently for association mining, which have shown to be very effective and usually outperform horizontal approaches. The main advantage of the vertical format is support for fast frequency counting via intersection operations on transaction ids (tids) and automatic pruning of irrelevant data. The main problem with these approaches is when intermediate results of vertical tid lists become too large for memory, thus affecting the algorithm scalability.

In this paper we present a novel vertical data representation called *Diffset*, that only keeps track of differences in the tids of a candidate pattern from its generating frequent patterns. We show that diffsets drastically cut down the size of memory required to store intermediate results. We show how diffsets, when incorporated into previous vertical mining methods, increase the performance significantly.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Data Mining

Keywords

Diffsets, Frequent Itemsets, Association Rule Mining

1. INTRODUCTION

Mining frequent patterns or itemsets is a fundamental and essential problem in many data mining applications. These applications include the discovery of association rules, strong rules, correlations, sequential rules, episodes, multi-dimensional patterns, and many other important discovery tasks [10]. The problem is formulated as follows: Given a large data base of item transactions, find all frequent itemsets, where a frequent itemset is one that occurs in at least a user-specified percentage of the data base.

Most of the proposed pattern-mining algorithms are a variant of Apriori [1]. Apriori employs a bottom-up, breadth-first search that enumerates every single frequent itemset. The process starts by scanning all transactions in the data base and computing the frequent items at the bottom. Next, a set of potentially frequent candidate 2-itemsets is formed from the frequent items. Another database scan is made to obtain their supports. The frequent 2-itemsets are retained for the next pass, and the process is repeated until all frequent itemsets have been enumerated. The Apriori heuristic achieves good performance gain by (possibly significantly) reducing the size of candidate sets. Apriori uses the *downward closure* property of itemset support to prune the search space — the property that all subsets of a frequent itemset must themselves be frequent. Thus only the frequent k-itemsets are used to construct candidate $(k+1)$-itemsets. A pass over the database is made at each level to find the frequent itemsets among the candidates.

Apriori-inspired algorithms [18, 14, 5, 13] show good performance with sparse datasets such as market-basket data, where the frequent patterns are very short. However, with dense datasets such as telecommunications and census data, which have many, long frequent patterns, the performance of these algorithms degrades incredibly. This degradation is due to the following reasons: these algorithms perform as many passes over the database as the length of the longest frequent pattern. This incurs high I/O overhead for scanning large disk-resident databases many times. Secondly, it is computationally expensive to check a large set of candidates by pattern matching, which is specially true for mining long patterns; a frequent pattern of length m implies the presence of $2^m - 2$ additional frequent patterns as well, each of which is explicitly examined by such algorithms. When m is large, the frequent itemset mining methods become CPU bound rather than I/O bound.

There has been recent interest in mining maximal frequent patterns in "hard" dense databases, where it is simply not feasible to mine all possible frequent itemsets; in such datasets one typically finds an exponen-

*This work was supported in part by NSF CAREER Award IIS-0092978, DOE Career Award DE-FG02-02ER25538, and NSF grant EIA-0103708.

SIGKDD '03, August 24-27, 2003, Washington, DC, USA

tial number of frequent itemsets. For example, finding long itemsets of length 30 or 40 is not uncommon [4]. Methods for finding the maximal elements include *All-MFS* [9], which is a randomized algorithm to discover maximal frequent itemsets. The *Pincer-Search* algorithm [12] not only constructs the candidates in a bottom-up manner like *Apriori*, but also starts a top-down search at the same time. This can help in reducing the number of database scans. *MaxMiner* [4] is another algorithm for finding the maximal elements. It uses efficient pruning based on lookaheads to quickly narrow the search. DepthProject [2] finds long itemsets using a depth first search of a lexicographic tree of itemsets, and uses a counting method based on transaction projections along its branches. Mafia [6] also uses several pruning strategies, uses vertical bit-vector data format, and compression and projection of bitmaps to improve performance. GenMax [8] is a backtrack search based algorithm for mining maximal frequent itemsets. GenMax uses a number of optimizations to prune the search space. It uses a novel technique called progressive focusing to perform maximality checking, and uses diffset propagation to perform fast frequency computation. Finally, FPgrowth [11] uses the novel frequent pattern tree (FP-tree) structure, which is a compressed representation of all the transactions in the database. It uses a recursive divide-and-conquer and database projection approach to mine long patterns.

Another recent promising direction is to mine only closed sets. It was shown in [15, 21] that it is not necessary to mine all frequent itemsets, rather frequent *closed* itemsets can be used to uniquely determine the set of all frequent itemsets and their *exact* frequency. Since the cardinality of closed sets is orders of magnitude smaller than all frequent sets, even dense domains can be mined. The advantage of closed sets is that they guarantee that the completeness property is preserved, i.e., all valid association rules can be found. Note that maximal sets do not have this property, since subset counts are not available. Methods for mining closed sets include the Apriori-based A-Close method [15], the Closet algorithm based on FP-trees [16] and Charm [23].

Most of the previous work on association mining has utilized the traditional horizontal transactional database format. However, a number of vertical mining algorithms have been proposed recently for association mining [22, 20, 7, 23, 8, 6] (as well as other mining tasks like classification [19]). In a vertical database each item is associated with its corresponding tidset, the set of all transactions (or tids) where it appears. Mining algorithms using the vertical format have shown to be very effective and usually outperform horizontal approaches. This advantage stems from the fact that frequent patterns can be counted via tidset intersections, instead of using complex internal data structures (candidate generation and counting happens in a single step). The horizontal approach on the other hand requires complex hash/search trees. Tidsets offer natural pruning of irrelevant transactions as a result of an intersection (tids not relevant drop out). Furthermore, for databases with long transactions it has been shown using a simple cost model, that the the vertical approach reduces the number of I/O operations [7]. In a recent study on the integration of database and mining, the *Vertical* algorithm [17] was shown to be the best approach (better than horizontal) when tightly integrating association mining with database systems. Also, VIPER [20], which uses compressed vertical bitmaps for association mining, was shown to outperform (in some cases) even an *optimal* horizontal algorithm that had complete *a priori* knowledge of all frequent itemsets, and only needed to find their frequency. MAFIA [6] and SPAM [3] use vertical bit-vectors for fast itemset and sequence mining respectively.

Despite the many advantages of the vertical format, when the tidset cardinality gets very large (e.g., for very frequent items) the methods start to suffer, since the intersection time starts to become inordinately large. Furthermore, the size of intermediate tidsets generated for frequent patterns can also become very large, requiring data compression and writing of temporary results to disk. Thus (especially) in dense datasets, which are characterized by high item frequency and many patterns, the vertical approaches may quickly lose their advantages.

In this paper we present a detailed evaluation of a novel vertical data representation called *diffset*, that only keeps track of differences in the tids of a candidate pattern from its generating frequent patterns. We show that diffsets drastically cut down (by orders of magnitude) the size of memory required to store intermediate results. The initial database stored in diffset format, instead of tidsets can also reduce the total database size. Thus even in dense domains the entire working set of patterns of several vertical mining algorithms can fit entirely in main-memory. Since the diffsets are a small fraction of the size of tidsets, intersection operations are performed extremely fast! We show how diffsets improve by several orders of magnitude the running time of vertical algorithms like Eclat [22] that mines all frequent itemsets. These results have not been previously published. We also compare our diffset-based methods against Viper [20], and against FPGrowth [11]. While we have previously *used* diffsets for closed [23] and maximal pattern mining [8], the detailed experimental evaluation of diffsets has not been presented before.

2. PROBLEM SETTING AND PRELIMINARIES

Association mining works as follows. Let $\mathcal{I}$ be a set of items, and $\mathcal{T}$ a database of transactions, where each transaction has a unique identifier (*tid*) and contains a set of items. A set $X \subseteq \mathcal{I}$ is also called an *itemset*, and a set $Y \subseteq \mathcal{T}$ is called a *tidset*. An itemset with k items is called a k-itemset. For convenience we write an itemset $\{A, C, W\}$ as ACW, and a tidset $\{2, 4, 5\}$ as 245. The *support* of an itemset X, denoted $\sigma(X)$, is the number of transactions in which it occurs as a subset. An itemset is *frequent* if its support is more than or equal to a user-specified *minimum support* (*min_sup*) value, i.e., if $\sigma(X) \geq$ *min_sup*. As a running example, consider the database shown in Figure 1. There are five dif-

ferent items, $\mathcal{I} = \{A, B, C, D, E\}$ and six transactions $\mathcal{T} = \{1, 2, 3, 4, 5, 6\}$. The table on the right shows all 19 frequent itemsets contained in at least three transactions, i.e., *min_sup*= 50%. A frequent itemset is called *maximal* if it is not a subset of any other frequent itemset. A frequent itemset X is called *closed* if there exists no proper superset $Y \supset X$ with $\sigma(X) = \sigma(Y)$.

DISTINCT DATABASE ITEMS

Jane Austen	Agatha Christie	Sir Arthur Conan Doyle	Mark Twain	P. G. Wodehouse
A	C	D	T	W

DATABASE

Transcation	Items
1	A C T W
2	C D W
3	A C T W
4	A C D W
5	A C D T W
6	C D T

ALL FREQUENT ITEMSETS

MINIMUM SUPPORT = 50%

Support	Itemsets
100% (6)	C
83% (5)	W, CW
67% (4)	A, D, T, AC, AW CD, CT, ACW
50% (3)	AT, DW, TW, ACT, ATW CDW, CTW, ACTW

Figure 1: Mining Frequent Itemsets

An association rule is an expression $X \xrightarrow{s,c} Y$, where X and Y are itemsets. The rule's support s is the joint probability of a transaction containing both X and Y, and is given as $s = \sigma(XY)$. The confidence c of the rule is the conditional probability that a transaction contains Y, given that it contains X, and is given as $c = \sigma(XY)/\sigma(Y)$. A rule is frequent if its support is greater than *min_sup*, and strong if its confidence is more than a user-specified minimum confidence (*min_conf*).

Association mining involves generating all rules in the database that have a support greater than *min_sup* (the rules are frequent) and that have a confidence greater than *min_conf* (the rules are strong). The main step in this process is to find all frequent itemsets having minimum support. The search space for enumeration of all frequent itemsets is given by the powerset $\mathcal{P}(\mathcal{I})$ which is exponential (2^m) in $m = |\mathcal{I}|$, the number of items. Since rule generation is relatively easy, and less I/O intensive than frequent itemset generation, we will focus only on the first step in the rest of this paper.

2.1 Common Data Formats

Figure 2 also illustrates some of the common data formats used in association mining. In the traditional horizontal approach, each transaction has a tid along with the itemset comprising the transaction. In contrast, the vertical format maintains for each item its tidset, a set of all tids where it occurs. Most of the past research has utilized the traditional horizontal database format for mining; some of these methods include Apriori [1], that mines frequent itemsets, and MaxMiner [4] and DepthProject [2] which mine maximal itemsets. Notable exception to this trend are the approaches that

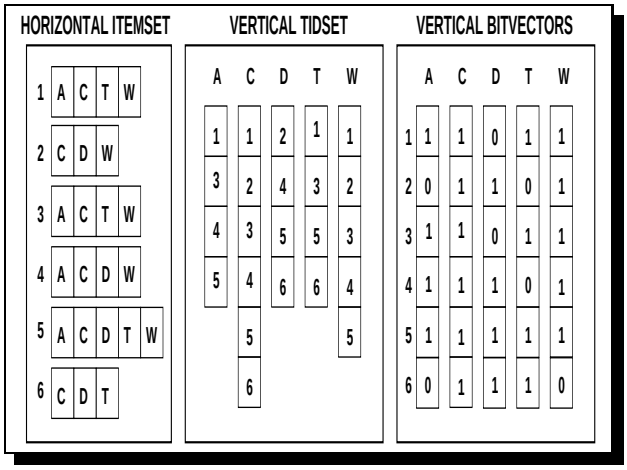

Figure 2: Common Data Formats

use a vertical database format, which include Eclat [22], Charm [21], and Partition [18]. Viper [20] and Mafia [6] use compressed vertical bitvectors instead of tidsets. Our main focus is to improve upon methods that utilize the vertical format for mining frequent patterns.

3. EQUIVALENCE CLASSES AND DIFFSETS

Let $\mathcal{I}$ be the set of items. Define a function $p : \mathcal{P}(\mathcal{I}) \times N \mapsto \mathcal{P}(\mathcal{I})$ where $p(X, k) = X[1 : k]$, the k length prefix of X. Define an equivalence relation θ_k on the subset tree as follows: $\forall X, Y \in \mathcal{P}(\mathcal{I}),\ X \equiv_{\theta_k} Y \Leftrightarrow p(X, k) = p(Y, k)$. That is, two itemsets are in the same class if they share a common k length prefix. θ_k is called a *prefix-based* equivalence relation [22].

The search for frequent patterns takes place over the subset (or itemset) search tree, as shown in Figure 3 (boxes indicate closed sets, circles the maximal sets and the infrequent sets have been crossed out). Each node in the subset search tree represents a prefix-based class. As Figure 3 shows the root of the tree corresponds to the class $\{A, C, D, T, W\}$, composed of the frequent items in the database (note: all these items share the empty prefix in common). The leftmost child of the root consists of the class $[A]$ of all subsets containing A as the prefix, i.e. the set $\{AC, AD, AT, AW\}$, and so on. At each node, the class is also called a *combine-set*. A class represents items that the prefix can be extended with to obtain a new frequent node. Clearly, no subtree of an infrequent prefix has to be examined.

The power of the equivalence class approach is that it breaks the original search space into *independent* subproblems. For the subtree rooted at A, one can treat it as a completely new problem; one can enumerate the patterns under it and simply prefix them with the item A, and so on. The branches also need not be explored in a lexicographic order; support-based ordering helps to narrow down the search space and prune unnecessary branches.

In the vertical mining approaches there is usually no distinct candidate generation and support counting phase like in Apriori. Rather, counting is simultaneous

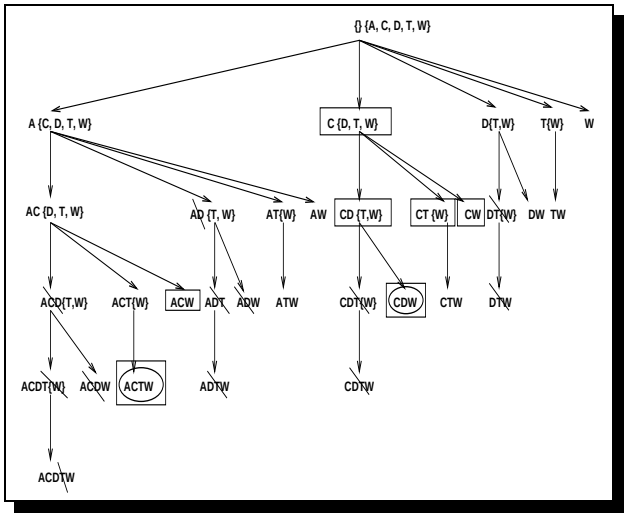

Figure 3: Subset Search Tree

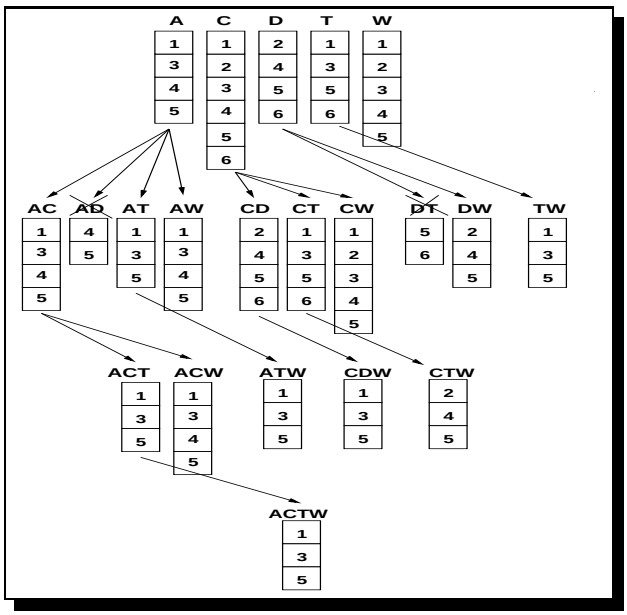

Figure 4: Tidsets for Pattern Counting

with generation. For a given node or prefix class, one performs intersections of the tidsets of all pairs of class elements, and checks if *min_sup* is met. Each resulting frequent itemset is a class unto itself with its own elements that will be expanded in the next step. That is to say, for a given class of itemsets with prefix P, $[P] = \{X_1, X_2, ..., X_n\}$, one performs the intersection of PX_i with all PX_j with $j > i$ to obtain a new class $[PX_i]$ with elements X_j where the itemset PX_iX_j is frequent. For example, from $[A] = \{C, D, T, W\}$, we obtain the classes $[AC] = \{D, T, W\}$, $[AD] = \{T, W\}$, and $[AT] = \{W\}$ (an empty class like $[AW]$ need not be explored further).

Vertical methods like Eclat [22] and Viper [20] utilize this independence of classes for frequent set enumeration. Figure 4 shows how a typical vertical mining process would proceed from one class to the next using intersections of tidsets of frequent items. For example, the tidsets of A ($t(A) = 1345$) and of D ($t(D) = 2456$) can be intersected to get the tidset for AD ($t(AD) = 45$) which is not frequent. As one can see in dense domains the tidset size can become very large. Combined with the fact that there are a huge number of patterns that exist in dense datasets, we find that the assumption that a sub-problem can be solved entirely in main-memory can easily be violated in such dense domains (particularly at low values of support). One way to solve this problem is the approach used by Viper, where they use compression of vertical bit-vectors to selectively read/write tidsets from/to disk as the computation progresses. Here we offer a fundamentally new way of processing tidsets using the concept of "differences".

3.1 Diffsets

Since each class is totally independent, in the sense that it has a list of all possible itemsets, and their tidsets, that can be combined with each other to produce all frequent patterns sharing a class prefix, our goal is to leverage this property in an efficient manner.

Our novel and extremely powerful solution (as we shall show experimentally) is to avoid storing the entire tidset of each member of a class. Instead we will keep track of only the differences in the tids between each class member and the class prefix itemset. These differences in tids are stored in what we call the *diffset*, which is a difference of two tidsets (namely, the prefix tidset and a class member's tidset). Furthermore, these differences are propagated all the way from a node to its children starting from the root. The root node's members can themselves use full tidsets or differences from the empty prefix (which by definition appears in all tids).

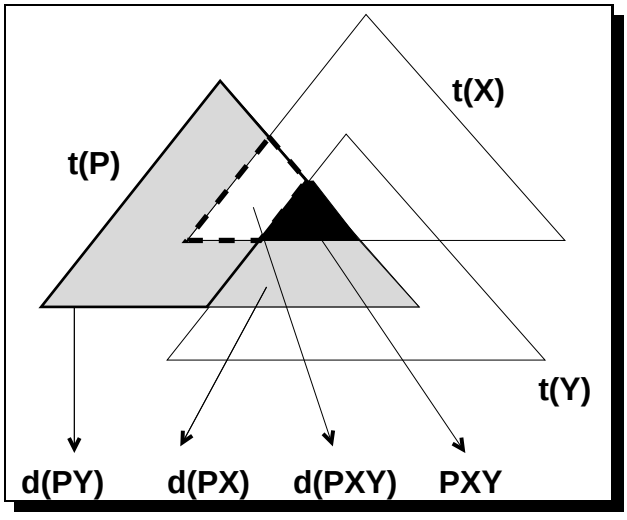

Figure 5: Diffsets Illustration

More formally, consider a given class with prefix P. Let $t(X)$ denote the tidset of element X, and let $d(X)$ the diffset of X, with respect to a prefix tidset, which is the current universe of tids. In normal vertical methods one has available for a given class the tidset for the prefix $t(P)$ as well as the tidsets of all class members $t(PX_i)$.

Assume that PX and PY are any two class members of P. By the definition of support it is true that $t(PX) \subseteq t(P)$ and $t(PY) \subseteq t(P)$. Furthermore, one obtains the support of PXY by checking the cardinality of $t(PX) \cap t(PY) = t(PXY)$.

Now suppose instead that we have available to us not $t(PX)$ but rather $d(PX)$, which is given as $t(P) - t(X)$, i.e., the differences in the tids of X from P. Similarly, we have available $d(PY)$. The first thing to note is that the support of an itemset is no longer the cardinality of the diffset, but rather it must be stored separately and is given as follows: $\sigma(PX) = \sigma(P) - |d(PX)|$. So, given $d(PX)$ and $d(PY)$ how can we compute if PXY is frequent?

We use the diffsets recursively as we mentioned above, i.e., $\sigma(PXY) = \sigma(PX) - |d(PXY)|$. So we have to compute $d(PXY)$. By our definition $d(PXY) = t(PX) - t(PY)$. But we only have diffsets, and not tidsets as the expression requires. This is easy to fix, since $d(PXY) = t(PX) - t(PY) = t(PX) - t(PY) + t(P) - t(P) = (t(P) - t(PY)) - (t(P) - t(PX)) = d(PY) - d(PX)$. In other words, instead of computing $d(PXY)$ as a difference of tidsets $t(PX) - t(PY)$, we compute it as the difference of the diffsets $d(PY) - d(PX)$. Figure 5 shows the different regions for the tidsets and diffsets of a given prefix class and any two of its members. The tidset of P, the triangle marked $t(P)$, is the universe of relevant tids. The gray region denotes $d(PX)$, while the region with the solid black line denotes $d(PY)$. Note also that both $t(PXY)$ and $d(PXY)$ are subsets of the tidset of the new prefix PX.

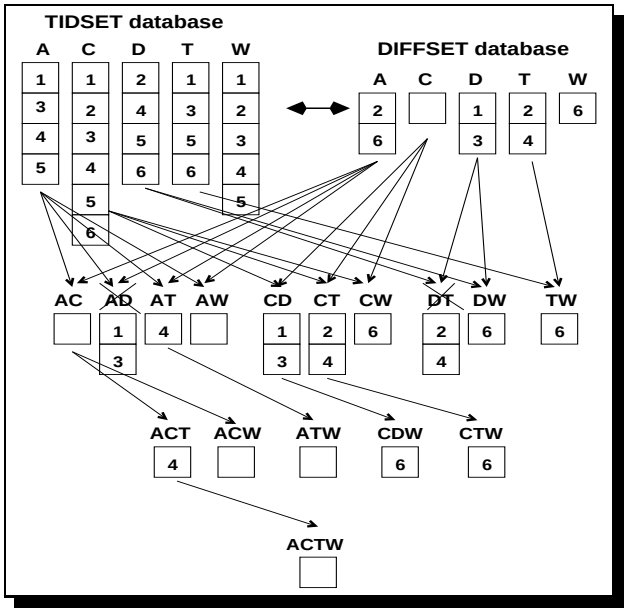

Figure 6: Diffsets for Pattern Counting

Example Consider Figure 6 showing how diffsets can be used to enhance vertical mining methods. We can choose to start with the original set of tidsets for the frequent items, or we could convert from the tidset representation to a diffset representation at the very beginning. One can clearly observe that for dense datasets like the one shown, a great reduction in the database size is achieved using this transformation (which we confirm on real datasets in the experiments below).

If we start with tidsets, then to compute the support of a 2-itemset like AD, we would find $d(AD) = t(A) - t(D) = 13$ (we omit set notation when there is no confusion). To find out if AD is frequent we check $\sigma(A) - |d(AD)| = 4 - 2 = 2$, thus AD is not frequent. If we had started with the diffsets, then we would have $d(AD) = d(D) - d(A) = 13 - 26 = 13$, the same result as before. Even this simple example illustrates the power of diffsets. The tidset database has 23 entries in total, while the diffset database has only 7 (3 times better). If we look at the size of all results, we find that the tidset-based approach takes up 76 tids in all, while the diffset approach (with initial diffset data) stores only 22 tids. If we compare by length, we find the average tidset size for frequent 2-itemsets is 3.8, while the average diffset size is 1. For 3-itemsets the tidset size is 3.2, but the avg. diffset size is 0.6. Finally for 4-itemsets the tidset size is 3 and the diffset size is 0! The fact that the database is smaller to start with and that the diffsets shrink as longer itemsets are found, allows the diffset based methods to become extremely scalable, and deliver orders of magnitude improvements over traditional approaches.

3.2 dEclat: Diffset Based Mining

dEclat($[P]$)**:**
for all $X_i \in [P]$ **do**
  for all $X_j \in [P]$, with $j > i$ **do**
    $R = X_i \cup X_j$;
    $d(R) = d(X_j) - d(X_i)$;
    if $\sigma(R) \geq min\_sup$ **then**
      $T_i = T_i \cup \{R\}$; //T_i initially empty
  if $T_i \neq \emptyset$ **then** dEclat(T_i);

Figure 7: Pseudo-code for dEclat

To illustrate the power of diffset-based mining, we have integrated diffsets with Eclat [24, 22], a state-of-the-art vertical mining algorithms. Our enhancement is called *dEclat*. We briefly discuss this algorithm below.

Figure 7 shows the pseudo-code for dEclat. Details on some optimizations, especially for computing frequent items and 2-itemsets have been omitted, which can be found in [22].

dEclat performs a depth-first search of the subset tree. Our experiments show that diffsets allow it to mine on much lower supports than other methods like Apriori and the base Eclat method. The input to the procedure is a set of class members for a subtree rooted at P. Frequent itemsets are generated by computing diffsets for all distinct pairs of itemsets and checking the support of the resulting itemset. A recursive procedure call is made with those itemsets found to be frequent at the current level. This process is repeated until all frequent itemsets have been enumerated. In terms of memory management it is easy to see that we need memory to store intermediate diffsets for at most two consecutive levels within a class. Once all the frequent itemsets for the next level have been generated, the itemsets at the

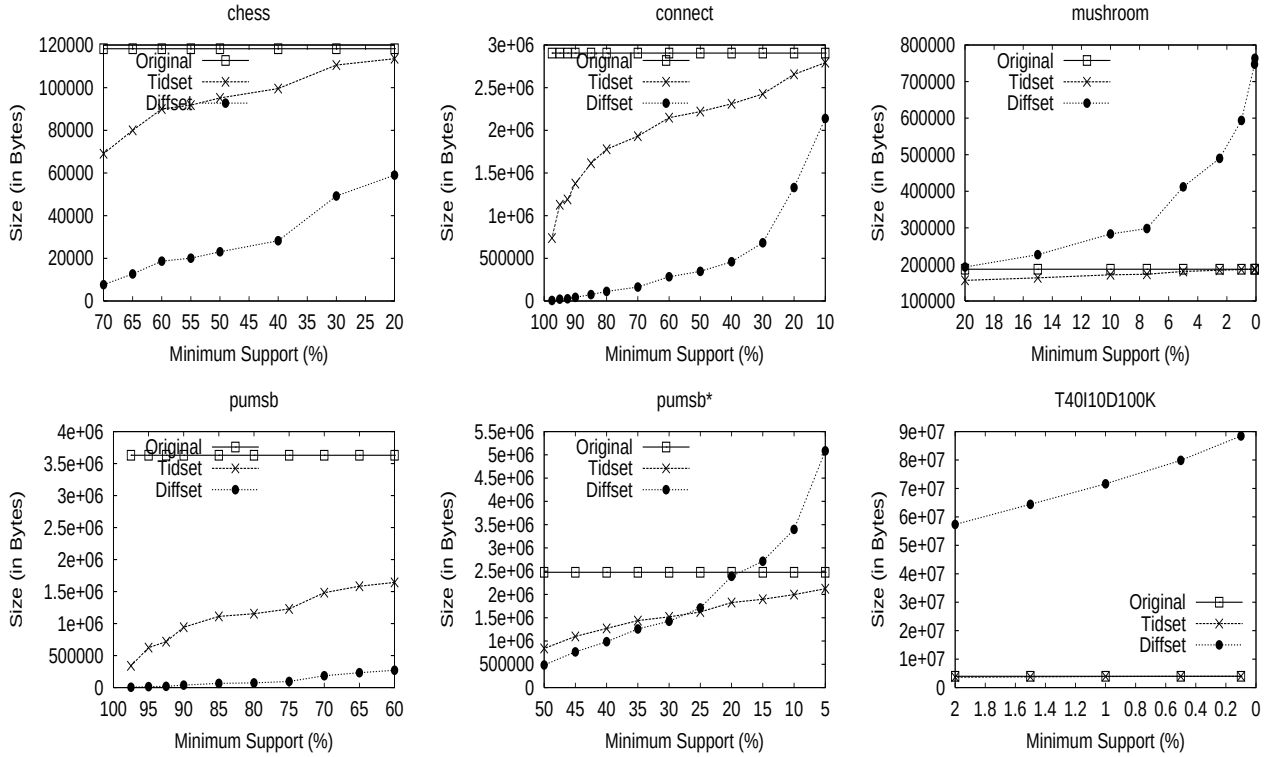

Figure 8: Size of Vertical Database: Tidset and Diffset

current level within a class can be deleted.

4. EXPERIMENTAL RESULTS

All experiments were performed on a 400MHz Pentium PC with 256MB of memory, running RedHat Linux 6.0. Algorithms were coded in C++. Furthermore, the times for all the vertical methods include all costs, including the conversion of the original database from a horizontal to a vertical format required for the vertical algorithms. We chose several real and synthetic datasets for testing the performance of algorithms. All datasets except the PUMS (pumsb and pumsb*) sets, are taken from the UC Irvine Machine Learning Database Repository. The PUMS datasets contain census data. pumsb* is the same as pumsb without items with 80% or more support. The mushroom database contains characteristics of various species of mushrooms. Finally the connect and chess datasets are derived from their respective game steps. Typically, these real datasets are very dense, i.e., they produce many long frequent itemsets even for very high values of support.

We also chose a few synthetic datasets, which have been used as benchmarks for testing previous association mining algorithms. These datasets mimic the transactions in a retailing environment. Usually the synthetic datasets are sparser when compared to the real sets.

Figure 9 shows the characteristics of the real and synthetic datasets used in our evaluation. It shows the number of items, the average transaction length and the number of transactions in each database. As one can see

Database	# Items	Avg. Length	# Records
chess	76	37	3,196
connect	130	43	67,557
mushroom	120	23	8,124
pumsb*	7117	50	49,046
pumsb	7117	74	49,046
T10I4D100K	1000	10	100,000
T40I10D100K	1000	40	100,000

Figure 9: Database Characteristics

the average transaction size for these databases is much longer than conventionally used in previous literature. We also include two sparse datasets (last two rows) to study its performance on both dense and sparse data.

4.1 Diffsets vs. Tidsets

Our first experiment is to compare the benefits of diffsets versus tidsets in terms of the database sizes using the two formats. We conduct experiment on several real (usually dense) and synthetic (sparse) datasets (see Section 4 for the dataset descriptions). In Figure 8 we plot the size of the original vertical database, the size of the database using tidsets of only the frequent items at a given level of support, and finally the database size if items were stored as diffsets. We see, for example on the dense pumsb dataset, that the tidset database at 60% support is 10^4 times smaller than the full vertical database; the diffset database is 10^6 times smaller! It is 100 times smaller than the tidset database. For the other dense datasets, connect and chess, the diffset format can be up to 100 times smaller depending on the

min_sup value.

For the sparser pumsb* dataset we notice a more interesting trend. The diffset database starts out smaller than the tidset database, but quickly grows more than even the full vertical database. For mushroom and other other synthetic datasets (results shown only for T40I10-D100K), we find that diffsets occupy (several magnitudes) more space than tidsets. We conclude that keeping the original database in diffset format is clearly superior if the database is dense, while the opposite is true for sparse datasets. In general we can use as starting point the smaller of the two formats depending on the database characteristics.

Due to the recursive dependence of a tidset or diffset on its parent equivalence class it is difficult to obtain analytically an estimate of their relative sizes. For this reason we conduct an experiment comparing the size of diffsets versus tidsets at various stages during mining. Figure 10 shows the average cardinality of the tidsets and diffsets for frequent itemsets of various lengths on different datasets, for a given minimum support. We denote by *db* a run with tidset format and by *Ddb* a run with the diffset format, for a given dataset *db*. We assume that the original dataset is stored in tidset format, thus the average tidset length is the same for single items in both runs. However, we find that while the tidset size remains more or less constant over the different lengths, the diffset size reduces drastically.

For example, the average diffset size falls below 1 for the last few lengths (over the length interval [11-16] for chess, [9-12] for connect, [7-17] for mushroom, [8-15] for pumsb*, 8 for pumsb, [9-11] for T10, [12-14] for T20). The only exception is T40 where the diffset length is 5 for the longest patterns. However, over the same interval range the avg. tidset size is 1682 for chess, 61325 for connect, 495 for mushroom, 18200 for pumsb*, 44415 for pumsb, 64 for T10, 182 for T20, and 728 for T40. Thus for long patterns the avg. diffset size is several orders of magnitude smaller than the corresponding avg. tidset size (4 to 5 orders of magnitude smaller on dense sets, and 2 to 3 orders of magnitude smaller on sparse sets). We also show in Table 11 the average diffset and tidset sizes across all lengths. We find that diffsets are smaller by one to two orders of magnitude for both dense as well as sparse datasets.

There is usually a cross-over point when a switch from tidsets to diffsets will be of benefit. For dense datasets it is better to start with the diffset format, while for sparse data it is better to start with tidset format and switch to diffsets in later stages, since diffsets on average are orders of magnitude smaller than tidsets. In general, we would like to know when it is beneficial to switch to the diffset format from the tidset format. Since each class is independent the decision can be made adaptively at the class level.

Consider a given class with prefix P, and assume that PX and PY are any two class members of P, with their corresponding tidsets $t(PX)$ and $t(PY)$. Consider the itemset PXY in a new class PX, which can either be stored as a tidset $t(PXY)$ or as a diffset $d(PXY)$. We define *reduction ratio* as $r = t(PXY)/d(PXY)$. For diffsets to be beneficial the reduction ratio should be at least 1. That is $r \geq 1$ or $t(PXY)/d(PXY) \geq 1$. Substituting for $d(PXY)$, we get $t(PXY)/(t(PX)-t(PY)) \geq 1$. Since $t(PX) - t(PY) = t(PX) - t(PXY)$, we have $t(PXY)/(t(PX) - t(PXY)) \geq 1$. Dividing by $t(PXY)$ we get, $1/(t(PX)/t(PXY) - 1) \geq 1$. After simplification we get $t(PX)/t(PXY) \leq 2$. In other words it is better to switch to the diffset format if the support of PXY is at least half of PX. If we start with a tidset database, empirically we found that for all real datasets it was better to use diffsets from length 2 onward. On the other hand, for the synthetic datasets we found that the 2-itemsets have an average support value 10 times smaller than the support of single items. Since this results in a reduction ratio less than 1, we found it better to switch to diffsets starting at 3-itemsets.

4.1.1 Comparison with Compressed Bitvectors

Viper [20] proposed using compressed vertical bitvectors instead of tidsets. Here we compare the bitvectors against diffsets. The classical way of compressing bitvectors is by using run-length encoding (RLE). It was noted in [20] that RLE is not appropriate for association mining, since it is not realistic to assume many consecutive 1's for a sparse dataset. If all 1's occur in an isolated manner, RLE outputs one word for the preceding 0 run and one word for the 1 itself. This results in a database that is double the size of a tidset database.

Viper uses a novel encoding scheme called *Skinning*. The idea is to divide runs of 1's and 0's in groups of size W_1 and W_0. Each full group occupies one bit set to 1. The last partial group ($R \bmod W_i$, where R is the run length) occupies $\lg W_i$ bits storing the partial count of 1's or 0's. Finally, a field separator bit (0) is placed between the full groups bits and the partial count field. Since the length of the count field is fixed, we know that we have to switch to 1's or 0's after having seen the count field for a run of 0's or 1's, respectively. If the minimum support is less than 50% the bitvectors for (longer) itemsets will have more 0's than 1's, thus it makes sense to use a large value for W_0 and a small value for W_1. Viper uses $W_0 = 256$ and $W_1 = 1$.

Let N denote the number of transactions, n_1 the number of 1's and n_0 the number of 0's in an itemset's bitvector. Assuming word length of 32 bits, a tidset takes $32n_1$ bits of storage. Assume (in the worst case) that all 1's are interspersed by exactly one 0, with a trailing run of 0's. In the skinning process each 1 leads to two bits of storage, one to indicate a full group and another for the separator. For n_1 1's, we get $2n_1$ bits of storage. For a compressed bitvector the number of bits used for isolated 1's is given as $2n_1$. For the n_1 isolated 0's we need $n_1(1 + \lg W_0) = 9n_1$ bits. For the remaining $n_0 - n_1 = (N - n_1) - n_1 = N - 2n_1$ 0's we need $(N - 2n_1)(1/W_0) = N/256 - n_1/128$ bits. Since $n_1 \geq N \times$ *min_sup* the total number of bits for the compressed vector is given as the sum of the number of bits required for 1's and 0's, given as $2n_1 + 9n_1 - n_1/128 + N/256 = 1407n_1/128 + N/256$. The benefit of skinning compared to tidset storage is then given as $C_w \geq 32n_1/(1407n_1/128 + N/256)$, where C_w denotes worst case compression ratio, i.e., the ra-

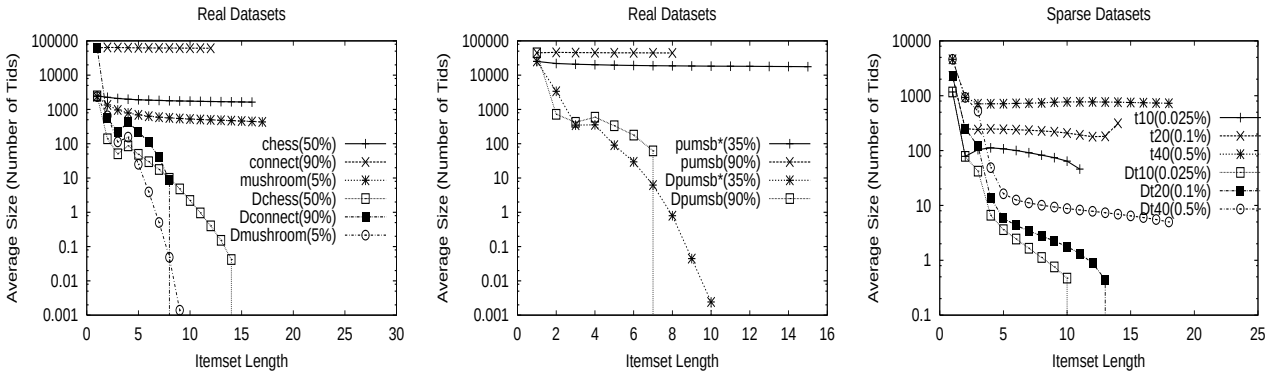

Figure 10: Average Size per Iteration: Tidset and Diffset

Database	*min_sup*	Max Length	Avg. Diffset Size	Avg. Tidset Size	Reduction Ratio
chess	0.5%	16	26	1820	70
connect	90%	12	143	62204	435
mushroom	5%	17	60	622	10
pumsb*	35%	15	301	18977	63
pumsb	90%	8	330	45036	136
T10I4D100K	0.025%	11	14	86	6
T20I16D100K	0.1%	14	31	230	11
T40I10D100K	0.5%	18	96	755	8

Figure 11: Average Tidset and Diffsets Cardinality

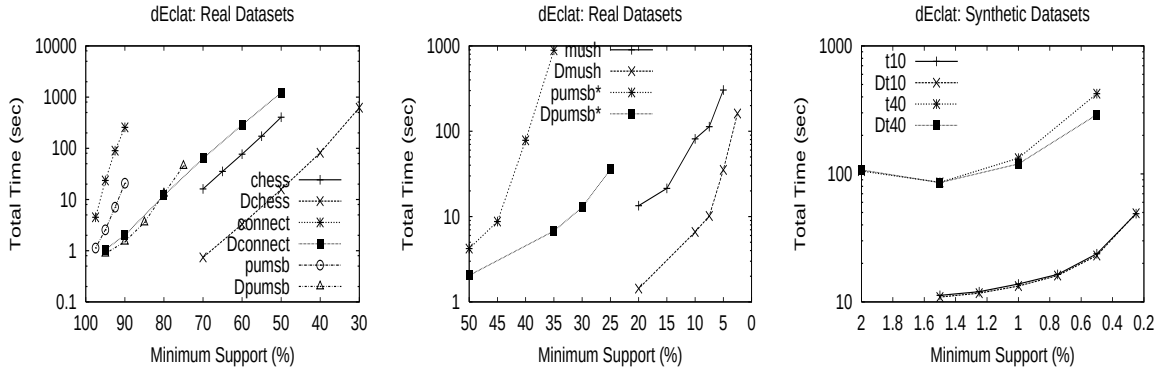

Figure 12: Improvements using Diffsets

tio of the storage required for the tidset divided by the storage required for the compressed bitvector. After substituting $n_1 \geq N \times min\_sup$ and simplifying, we get $C_w \geq \frac{1}{0.34+\frac{1}{8192\times min\_sup}}$.

For support values less than 0.02% C_w is less than 1, which means that skinning causes expansion rather than compression. The maximum value of C_w reaches 2.91 asymptotically. For *min_sup*=0.1%, $C_w = 2.14$ Thus for reasonable support values the compression ratio is expected to be between 2 and 3 compared to tidsets. Supposing we assume a best case scenario for skinning, where all the n_1 1's come before the n_0 0's (this is highly unlikely to happen). We would then need n_1 bits to represent the single run of 1's, and $n_0/W_0 = n_0/256$ bits for the run of 0's. The total space is thus $n_1 + n_0/256 = n_1 + (N - n_1)/256 = 255n_1/256 + N/256$. The best case compression ration is given as $C_b \geq 32n_1/(255n_1/256 + N/256)$. After simplification this yields $C_b \geq \frac{1}{0.03+\frac{1}{8192\times min\_sup}}$, which asymptotically reaches a value of 32. In the best case, the compression ratio is 32, while in the worst case the compression ratio is only 2 to 3. The skinning process can thus provide at most 1 order of magnitude compression ratio over tidsets. However, as we have seen diffsets provide anywhere from 2 to 5 orders of magnitude compression over tidsets. The experimental and theoretical results shown above clearly substantiate the power of diffset based mining!

4.2 Frequent Pattern Mining

Figure 12 proves the advantage of diffsets over the base method that use only tidsets. We give a thorough sets of experiments spanning all the real and synthetic datasets mentioned above, for various values of mini-

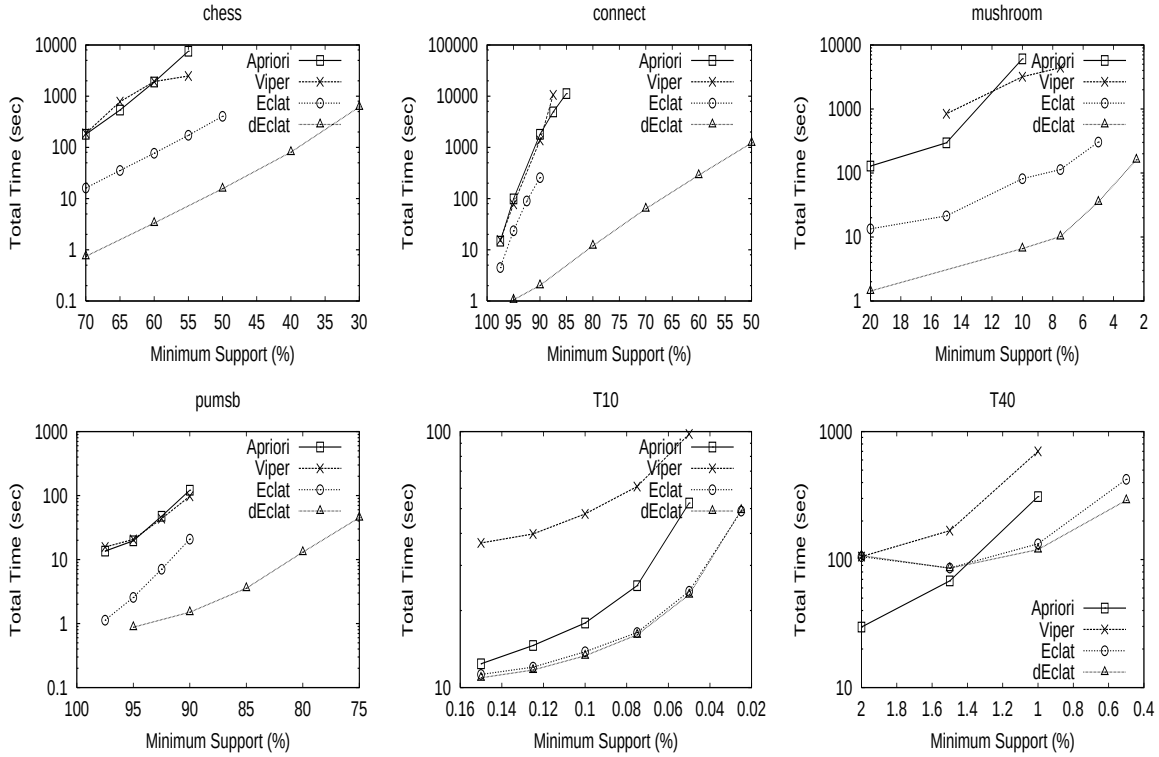

Figure 13: Comparative Performance: Apriori, Viper, Eclat, dEclat

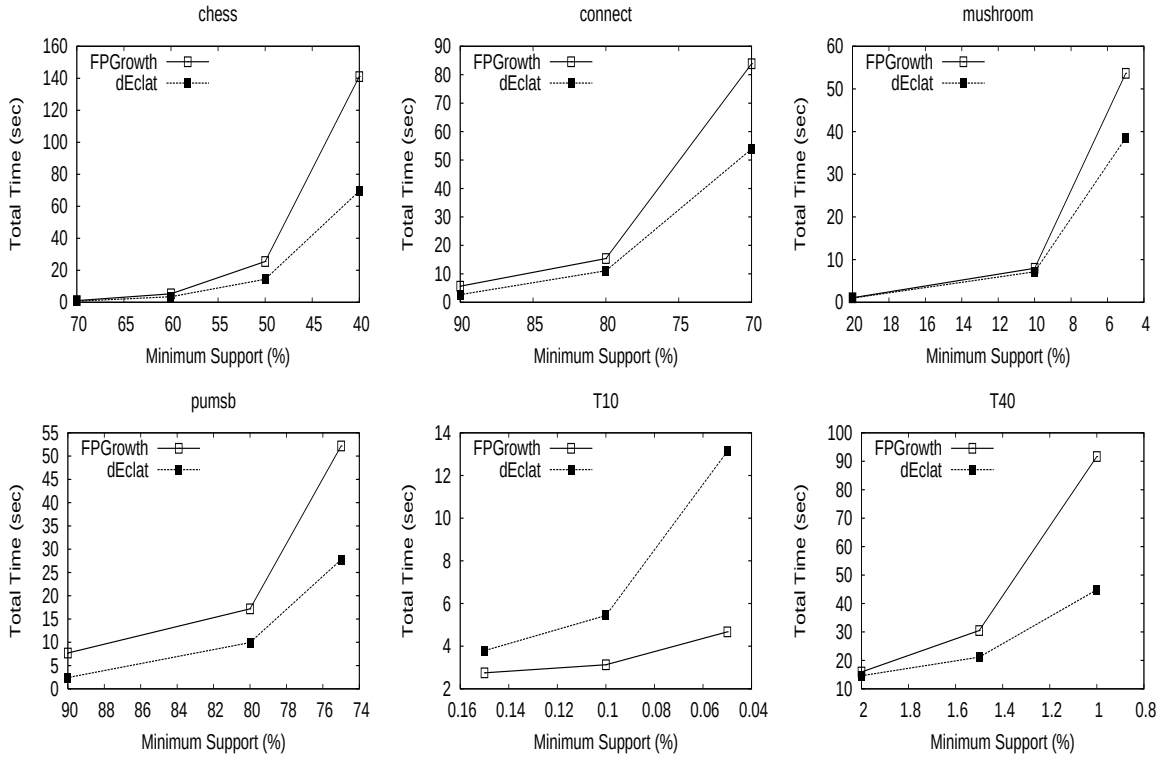

Figure 14: Comparative Performance: FPGrowth, dEclat

mum support, and compare the diffset-based algorithms against the base algorithm. We denote by *db* a run with tidset format and by *Ddb* a run with the diffset format, for a given dataset *db*. We find that one the real datasets, the diffset algorithms outperform tidset based methods by several orders of magnitude. The benefits

on the synthetic datasets are only marginal, up to a factor of 2 improvement.

In Figure 13 we compare horizontal and vertical algorithms for mining the set of all frequent patterns. We compare the new dEclat method against Eclat [22], the classic Apriori [1] and recent Viper [20] algorithm. We see the dEclat outperforms by orders of magnitude the other algorithms. One observation that can be made is that dEclat makes Eclat more scalable, allowing it to enumerate frequent patterns even in dense datasets for relatively low values of support. On dense datasets Viper is better than Apriori at lower support values, but Viper is uncompetitive with Eclat and dEclat.

Figure 14 compares dEclat with FPGrowth [11] [1]. The results are shown separately since the authors provided only a Windows executable. We tested them on a 800 Mhz, 256MB memory, Pentium III processor running Win98 and cygwin. We observe that dEclat outperforms FPGrowth by a factor of 2 for all datasets except T10I4D100K which is very sparse and has only a few long patterns. The time difference increases with decreasing support.

4.3 Conclusions

In this paper we presented a detailed evaluation of a novel vertical data representation called *Diffset*, that only keeps track of differences in the tids of a candidate pattern from its generating frequent patterns. We show that diffsets drastically cut down the size of memory required to store intermediate results. We show how diffsets, when incorporated into a previous vertical mining methods, increase the performance significantly.

5. REFERENCES

[1] R. Agrawal, et al. Fast discovery of association rules. In U. Fayyad and et al (eds.), *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 1996.

[2] Ramesh Agrawal, Charu Aggarwal, and V.V.V. Prasad. Depth First Generation of Long Patterns. In *7th Int'l Conference on Knowledge Discovery and Data Mining*, August 2000.

[3] Jay Ayres, J. E. Gehrke, Tomi Yiu, and Jason Flannick. Sequential pattern mining using bitmaps. In *SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, July 2002.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *ACM SIGMOD Conf. Management of Data*, June 1998.

[5] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *ACM SIGMOD Conf. Management of Data*, May 1997.

[6] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: a maximal frequent itemset algorithm for transactional databases. In *Intl. Conf. on Data Engineering*, April 2001.

[7] B. Dunkel and N. Soparkar. Data organization and access for efficient data mining. In *15th IEEE Intl. Conf. on Data Engineering*, March 1999.

[8] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *1st IEEE Int'l Conf. on Data Mining*, November 2001.

[9] D. Gunopulos, H. Mannila, and S. Saluja. Discovering all the most specific sentences by randomized algorithms. In *Intl. Conf. on Database Theory*, January 1997.

[10] J. Han and M. Kamber. *Data Mining: Concepts and Techniuqes*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Conf. Management of Data*, May 2000.

[12] D-I. Lin and Z. M. Kedem. Pincer-search: A new algorithm for discovering the maximum frequent set. In *6th Intl. Conf. Extending Database Technology*, March 1998.

[13] J-L. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *14th Intl. Conf. on Data Engineering*, February 1998.

[14] J. S. Park, M. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *Intl. Conf. Management of Data*, May 1995.

[15] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *7th Intl. Conf. on Database Theory*, January 1999.

[16] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *SIGMOD Int'l Workshop on Data Mining and Knowledge Discovery*, May 2000.

[17] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with databases: alternatives and implications. In *ACM Intl. Conf. Management of Data*, June 1998.

[18] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *21st VLDB Conf.*, 1995.

[19] J. Shafer, R. Agrawal, and M. Mehta. Sprint: A scalable parallel classifier for data mining. In *22nd VLDB Conference*, March 1996.

[20] P. Shenoy, et al. Turbo-charging vertical mining of large databases. In *Intl. Conf. Management of Data*, May 2000.

[21] M. J. Zaki. Generating non-redundant association rules. In *Int'l Conf. Knowledge Discovery and Data Mining*, August 2000.

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

[23] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *2nd SIAM Int'l Conf. on Data Mining*, April 2002.

[24] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, August 1997.

[1] We extend our thanks to Jiawei Han amd Jian Pei.