CHARM: An Efficient Algorithm for Closed Itemset Mining

Mohammed J. Zaki * ***and Ching-Jui Hsiao*** †

Abstract

The set of frequent *closed* itemsets uniquely determines the exact frequency of all itemsets, yet it can be orders of magnitude smaller than the set of all frequent itemsets. In this paper we present CHARM, an efficient algorithm for mining all frequent closed itemsets. It enumerates closed sets using a dual itemset-tidset search tree, using an efficient hybrid search that skips many levels. It also uses a technique called *diffsets* to reduce the memory footprint of intermediate computations. Finally it uses a fast hash-based approach to remove any "non-closed" sets found during computation. An extensive experimental evaluation on a number of real and synthetic databases shows that CHARM significantly outperforms previous methods. It is also linearly scalable in the number of transactions.

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 [11]. 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 database.

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. 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 [5, 13, 16] 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, where there are 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. Secondly, a frequent pattern of length l implies the presence of $2^l - 2$ additional frequent patterns as well, each of which is explicitly examined by such algorithms. When l is large, the frequent itemset mining methods become CPU bound rather than I/O bound. In other words, it is practically unfeasible to mine the set of all frequent patterns for other than small l. On the other hand, in many real world problems (e.g., patterns in biosequences, census data, etc.) finding long itemsets of length 30 or 40 is not uncommon [4].

*Computer Science Department, Rensselaer Polytechnic Institute, Troy NY 12180. Email: zaki@cs.rpi.edu.
This work was supported in part by NSF CAREER Award IIS-0092978, and NSF Next Generation Software Program grant EIA-0103708.
†The work was done while the author was at RPI.

There are two current solutions to the long pattern mining problem. The first one is to mine only the maximal frequent itemsets [2, 4, 6, 10, 12], which are typically orders of magnitude fewer than all frequent patterns. While mining maximal sets help understand the long patterns in dense domains, they lead to a loss of information; since subset frequency is not available maximal sets are not suitable for generating rules. The second is to mine only the frequent closed sets [3, 14, 15, 18]. Closed sets are *lossless* in the sense that they uniquely determine the set of all frequent itemsets and their *exact* frequency. At the same time closed sets can themselves be orders of magnitude smaller than all frequent sets, especially on dense databases.

Our past work [18] addressed the problem of non-redundant rule generation, provided that closed sets are available; *an algorithm to efficiently mine the closed sets was not described.* A natural question to ask is whether existing methods can be used to generate closed sets? It is not feasible to generate this set using Apriori-like [1] bottom-up search methods that examine all subsets of a frequent itemset (except when patterns are very short). Neither is it possible to use maximal patterns [2, 4] to find the closed itemsets, since all subsets of the maximal frequent itemsets would again have to be examined. We thus need methods that directly enumerate the closed patterns.

Contributions We introduce CHARM [1], an efficient algorithm for enumerating the set of all frequent closed itemsets. There are a number of innovative ideas employed in the development of CHARM; these include: 1) CHARM simultaneously explores both the itemset space and transaction space, over a novel *IT-tree* (itemset-tidset tree) search space. In contrast, most previous methods exploit only the itemset search space. 2) CHARM uses a highly efficient hybrid search method that skips many levels of the IT-tree to quickly identify the frequent closed itemsets, instead of having to enumerate many possible subsets. 3) It uses a fast hash-based approach to eliminate non-closed itemsets during subsumption checking. CHARM also utilizes a novel vertical data representation called *diffset* [20], recently proposed by us, for fast frequency computations. Diffsets keep track of differences in the tids of a candidate pattern from its prefix pattern. Diffsets drastically cut down (by orders of magnitude) the size of memory required to store intermediate results. Thus the entire working set of patterns can fit entirely in main-memory, even for large databases.

We assume in this paper that the initial database is disk-resident, but that the intermediate results fit entirely in memory. Several factors make this a realistic assumption. First, CHARM breaks the search space into small independent chunks (based on prefix equivalence classes [19]). Second, diffsets lead to extremely small memory footprint (this is experimentally verified). Finally, CHARM uses simple set difference (or intersection) operations, and requires no complex internal data structures (candidate generation and counting happens in a single step). The current trend toward large (gigabyte-sized) main memories, combined with the above features, makes CHARM a practical and efficient algorithm for reasonably large databases.

We compare CHARM against previous methods for mining closed sets such as Close [14], Closet [15], Mafia [6] and Pascal [3]. Extensive experiments confirm that CHARM provides significant improvement over existing methods for mining closed itemsets, for both dense as well as sparse datasets.

2 Frequent Pattern Mining

Let $\mathcal{I}$ be a set of items, and $\mathcal{D}$ a database of transactions, where each transaction has a unique identifier (*tid*) and contains a set of items. The set of all tids is denoted as $\mathcal{T}$. 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. For an itemset X, we denote its corresponding tidset as $t(X)$, i.e., the set of all tids that contain X as a subset. For a tidset Y, we denote its corresponding itemset as $i(Y)$, i.e., the set of items common to all the tids in Y. Note that $t(X) = \bigcap_{x \in X} t(x)$, and $i(Y) = \bigcap_{y \in Y} i(y)$. For example, $t(ACW) = t(A) \cap t(C) \cap t(W) = 1345 \cap 123456 \cap 12345 = 1345$ and $i(12) = i(1) \cap i(2) = ACTW \cap CDW = CW$. We use the notation $X \times t(X)$ to refer an itemset-tidset pair, and call it an *IT-pair*.

The *support* [1] of an itemset X, denoted $\sigma(X)$, is the number of transactions in which it occurs as a subset, i.e., $\sigma(X) = |t(X)|$. 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*. 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

[1] CHARM stands for Closed Association Rule Mining; the 'H' is gratuitous

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	ACTW
2	CDW
3	ACTW
4	ACDW
5	ACDTW
6	CDT

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. *Example DB*

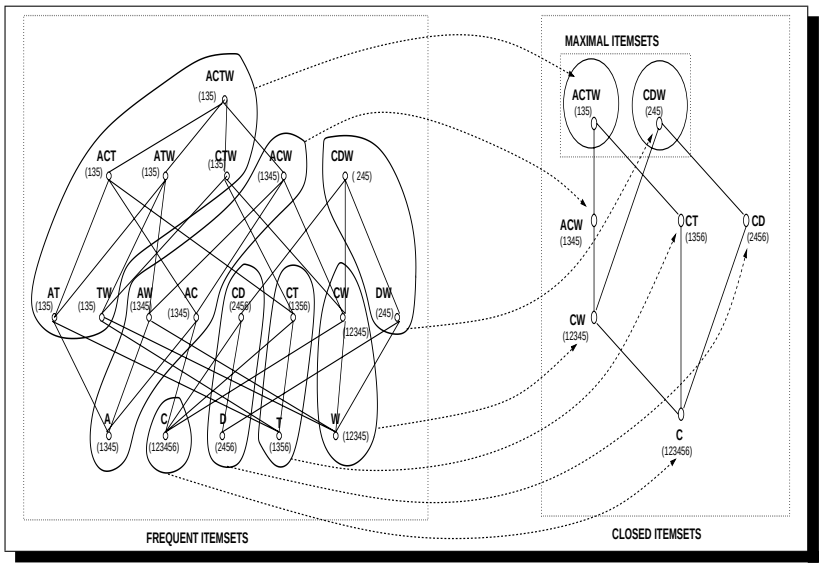

Figure 2. *Frequent, Closed and Maximal Itemsets*

superset $Y \supset X$ with $\sigma(X) = \sigma(Y)$. As a running example, consider the database shown in Figure 1. There are five different items, $\mathcal{I} = \{A, C, D, T, W\}$ 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%. Consider Figure 2 which shows the 19 frequent itemsets organized as a subset lattice; their corresponding tidsets have also been shown. The 7 closed sets are obtained by collapsing all the itemsets that have the same tidset, shown in the figure by the circled regions. Looking at the closed itemset lattice we find that there are 2 maximal frequent itemsets (marked with a circle), $ACTW$ and CDW. As the example shows, in general if $\mathcal{F}$ denotes the set of frequent itemsets, $\mathcal{C}$ the set of closed ones, and $\mathcal{M}$ the set of maximal itemsets, then we have $\mathcal{M} \subseteq \mathcal{C} \subseteq \mathcal{F}$. Generally, the $\mathcal{C}$ can be orders of magnitude smaller than $\mathcal{F}$ (especially for dense datasets), while $\mathcal{M}$ can itself be orders of magnitude smaller than $\mathcal{C}$. However, the closed sets are lossless in the sense that the exact frequency of all frequent sets can be determined from $\mathcal{C}$, while $\mathcal{M}$ leads to a loss of information. To find if a set X is frequent we find the smallest closed set that is a superset of X. If no superset exists, then X is not frequent. For example, to check if ATW is frequent, we find that $ACTW$ is the smallest closed set that contains it; ATW is thus frequent and has the same frequency as $ACTW$. On the other hand, DT is not frequent since there is no frequent closed set that contains it.

Related Work Our past work [18] addressed the problem of non-redundant rule generation, provided that closed sets are available; an algorithm to efficiently mine the closed sets was not described in that paper. There have been several recent algorithms proposed for this task.

Close [14] is an Apriori-like algorithm that directly mines frequent closed itemsets. There are two main steps in Close. The first is to use bottom-up search to identify *generators*, the smallest frequent itemset that determines a closed itemset. For example, consider the frequent itemset lattice in Figure 2. The item A is a generator for the closed set ACW, since it is the smallest itemset with the same tidset as ACW. All generators are found using a simple modification of Apriori. After finding the frequent sets at level k, Close compares the support of each set with its subsets at the previous level. If the support of an itemset matches the support of any of its subsets, the itemset cannot be a generator and is thus pruned. The second step in Close is to compute the closure of all the generators found in the first step. To compute the closure of an itemset we have to perform an intersection of all transactions where it occurs as a subset. The closures for all generators can be computed in just one database scan, provided all generators fit in memory. Nevertheless computing closures this way is an expensive operation.

The authors of Close recently developed Pascal [3], an improved algorithm for mining closed and frequent sets. They introduce the notion of *key patterns* and show that other frequent patterns can be inferred from the key patterns without access to the database. They showed that Pascal, even though it finds both frequent and closed sets, is typically twice as fast as Close, and ten times as fast as Apriori.

Since Pascal enumerates all patterns, it is only practical when pattern length is short (as we shall see in the experimental section). The *Closure* algorithm [7] is also based on a bottom-up search. It performs only marginally better than Apriori, thus CHARM should outperform it easily.

Recently two new algorithms for finding frequent closed itemsets have been proposed. Closet [15] uses a 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. We will show later that CHARM outperforms Closet by orders of magnitude as support is lowered. Mafia [6] is primarily intended for maximal pattern mining, but has an option to mine the closed sets as well. Mafia relies on efficient compressed and projected vertical bitmap based frequency computation. At higher supports both Mafia and CHARM exhibit similar performance, but as one lowers the support the gap widens exponentially. CHARM can deliver over factor of 10 improvements over Mafia for low supports.

There have been several efficient algorithms for mining maximal frequent itemsets such as MaxMiner [4], DepthProject [2], Mafia [6], and GenMax [10]. It is not practical to first mine maximal patterns and then to check if each subset is closed, since we would have to check 2^l subsets, where l is length of the longest pattern (we can easily have patterns of length 30 to 40 or more; see experimental section). In [21] we tested a modified version of MaxMiner to discover closed sets in a post-processing step, and found it to be too slow for all except short patterns.

3 Itemset-Tidset Search Tree and Equivalence Classes

Let $\mathcal{I}$ be the set of items. Define a function $p(X,k) = X[1:k]$ as the k length prefix of X, and a *prefix-based* equivalence relation θ_k [19] on itemsets as follows: $\forall X, Y \subseteq \mathcal{I},\ X \equiv_{\theta_k} Y \Leftrightarrow p(X,k) = p(Y,k)$. That is, two itemsets are in the same class if they share a common k length prefix.

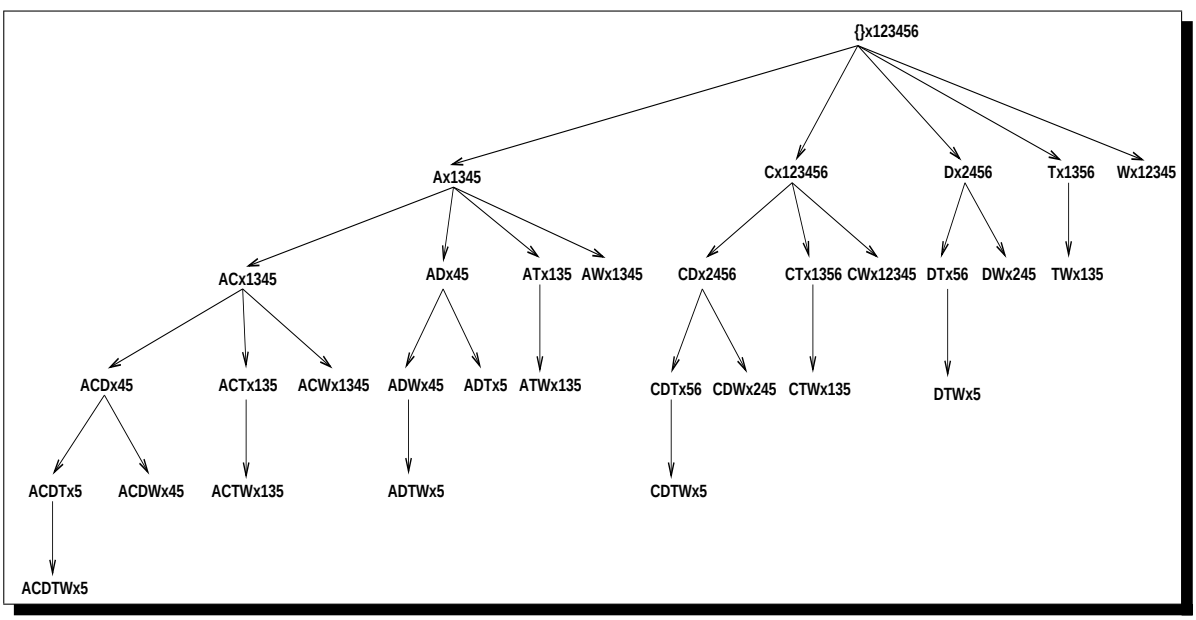

Figure 3. *IT-Tree: Itemset-Tidset Search Tree*

CHARM performs a search for closed frequent sets over a novel IT-tree search space, shown in Figure 3. Each node in the IT-tree, represented by an itemset-tidset pair, $X \times t(X)$, is in fact a prefix-based class. All the children of a given node X, belong to its equivalence class, since they all share the same prefix X. We denote an equivalence class as $[P] = \{l_1, l_2, \cdots, l_n\}$, where P is the parent node (the prefix), and each l_i is a single item, representing the node $Pl_i \times t(Pl_i)$. For example, the root of the tree corresponds to the class $[] = \{A, C, D, T, W\}$. The leftmost child of the root consists of the class $[A]$ of all itemsets containing A as the prefix, i.e. the set $\{C, D, T, W\}$. As can be discerned, each class member represents one child of the parent node. 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* sub-problems. For any subtree rooted at node X, one can treat it as a completely new problem; one can enumerate the patterns under it and simply prefix them with the item X, and so on.

```
Enumerate-Frequent([P]):
  for all l_i ∈ [P] do
    [P_i] = ∅;
    for all l_j ∈ [P], with j > i do
      I = Pl_i l_j;
      T = t(Pl_i) ∩ t(Pl_j);
      if |T| ≥ min_sup then
        [P_i] = [P_i] ∪ {I × T};
    if (DFS) then Enumerate-Frequent([P_i]); delete [P_i];
  if (BFS) then
    for all [P_i] do Enumerate-Frequent([P_i]); delete [P_i];
```

Figure 4. *Pattern Enumeration*

Frequent pattern enumeration is straightforward in the IT-tree framework. For a given node or prefix class, one can perform intersections of the tidsets of all pairs of elements in a class, and check if *min_sup* is met; support counting is simultaneous with generation. Each resulting frequent itemset is a class unto itself, with its own elements, that will be recursively expanded. That is to say, for a given class of itemsets with prefix P, $[P] = \{l_1, l_2, \cdots, l_n\}$, one can perform the intersection of $t(Pl_i)$ with all $t(Pl_j)$ with $j > i$, to obtain a new class of frequent extensions, $[Pl_i] = \{l_j \mid j > i \text{ and } \sigma(Pl_il_j) \geq \ min\_sup\}$. For example, from the null root $[] = \{A, C, D, T, W\}$, with *min_sup*=50% we obtain the classes $[A] = \{C, T, W\}$, $[C] = \{D, T, W\}$, and $[D] = \{W\}$, and $[W] = \{\}$. Note that class $[A]$ does not contain D since AD is not frequent. Figure 4 gives a pseudo-code description of a breadth-first (BFS) and depth-first (DFS) exploration of the IT-tree for all frequent patterns. CHARM improves upon this basic enumeration scheme, using the conceptual framework provided by the IT-tree; it is not assumed that all the tidsets will fit in memory, rather CHARM materializes only a small portion of the tree in memory at any given time.

Basic Properties of Itemset-Tidset Pairs We use the concept of a closure operation [9, 18] to check if a given itemset X is closed or not. We define a *closure* of an itemset X, denoted $c(X)$, as the the smallest closed set that contains X. Recall that $i(Y)$ is the set of items common to all the tids in the tidset Y, while $t(X)$ are tids common to all the items in X. To find the closure of an itemset X we first compute the image of X in the transaction space to get $t(X)$. We next map $t(X)$ to its image in the itemset space using the mapping i to get $i(t(X))$. It is well know that the resulting itemset must be closed [9], i.e., $c(X) = i \circ t(X) = i(t(X))$. It follows that an itemset X is closed if and only if $X = c(X)$. For example the itemset ACW is closed since $c(ACW) = i(t(ACW)) = i(1345) = ACW$. The support of an itemset X is also equal to the support of its closure, i.e., $\sigma(X) = \sigma(c(X))$ [14, 18].

For any two nodes in the IT-tree, $X_i \times t(X_i)$ and $X_j \times t(X_j)$, if $X_i \subseteq X_j$ then it is the case that $t(X_j) \subseteq t(X_i)$. For example, for $ACW \subseteq ACTW$, $t(ACW) = 1345 \supseteq 135 = t(ACTW)$. Let us define $f : \mathcal{P}(\mathcal{I}) \mapsto N$ to be a one-to-one mapping from itemsets to integers. For any two itemsets X_i and X_j, we say $X_i \leq_f X_j$ iff $f(X_i) \leq f(X_j)$. The function f defines a total order over the set of all itemsets. For example, if f denotes the lexicographic ordering, then itemset $AC \leq AD$, but if f sorts itemsets in increasing order of their support, then $AD \leq AC$ if $\sigma(AD) \leq \sigma(AC)$. There are four basic properties of IT-pairs that CHARM leverages for fast exploration of closed sets. Assume that we are currently processing a node $P \times t(P)$ where $[P] = \{l_1, l_2, \cdots, l_n\}$ is the prefix class. Let X_i denote the itemset Pl_i, then each member of $[P]$ is an IT-pair $X_i \times t(X_i)$.

Theorem 1. *Let $X_i \times t(X_i)$ and $X_j \times t(X_j)$ be any two members of a class $[P]$, with $X_i \leq_f X_j$, where f is a total order (e.g., lexicographic or support-based). The following four properties hold:*

1. *If $t(X_i) = t(X_j)$, then $c(X_i) = c(X_j) = c(X_i \cup X_j)$*
2. *If $t(X_i) \subset t(X_j)$, then $c(X_i) \neq c(X_j)$, but $c(X_i) = c(X_i \cup X_j)$*

3. *If* $t(X_i) \supset t(X_j)$, *then* $c(X_i) \neq c(X_j)$, *but* $c(X_j) = c(X_i \cup X_j)$

4. *If* $t(X_i) \neq t(X_j)$, *then* $c(X_i) \neq c(X_j) \neq c(X_i \cup X_j)$

PROOF AND DISCUSSION:
1. If $t(X_i) = t(X_j)$, then obviously $i(t(X_i)) = i(t(X_j)$, i.e., $c(X_i) = c(X_j)$. Further $t(X_i) = t(X_j)$ implies that $t(X_i \cup X_j) = t(X_i) \cap t(X_j) = t(X_i)$. Thus $i(t(X_i \cup X_j)) = i(t(X_i))$, giving us $c(X_i \cup X_j) = c(X_i)$. This property implies that we can replace every occurrence of X_i with $X_i \cup X_j$, and we can remove the element X_j from further consideration, since its closure is identical to the closure of $X_i \cup X_j$.
2. If $t(X_i) \subset t(X_j)$, then $t(X_i \cup X_j) = t(X_i) \cap t(X_j) = t(X_i) \neq t(X_j)$, giving us $c(X_i \cup X_j) = c(X_i) \neq c(X_j)$. Thus we can replace every occurrence of X_i with $X_i \cup X_j$, since they have identical closures. But since $c(X_i) \neq c(X_j)$ we cannot remove element X_j from class $[P]$; it generates a different closure.
3. Similar to case 2 above.
4. If $t(X_i) \neq t(X_j)$, then clearly $t(X_i \cup X_j) = t(X_i) \cap t(X_j) \neq t(X_i) \neq t(X_j)$, giving us $c(X_i \cup X_j) \neq c(X_i) \neq c(X_j)$. No element of the class can be eliminated; both X_i and X_j lead to different closures. ■

4 CHARM: Algorithm Design and Implementation

We now present CHARM, an efficient algorithm for mining all the closed frequent itemsets. We will first describe the algorithm in general terms, independent of the implementation details. We then show how the algorithm can be implemented efficiently. CHARM simultaneously explores both the itemset space and tidset space using the IT-tree, unlike previous methods which typically exploit only the itemset space. CHARM uses a novel search method, based on the IT-pair properties, that skips many levels in the IT-tree to quickly converge on the itemset closures, rather than having to enumerate many possible subsets.

The pseudo-code for CHARM appears in Figure 5. The algorithm starts by initializing the prefix class $[P]$, of nodes to be examined, to the frequent single items and their tidsets in Line 1. We assume that the elements in $[P]$ are ordered according to a suitable total order f. The main computation is performed in CHARM-EXTEND which returns the set of closed frequent itemsets $\mathcal{C}$.

CHARM-EXTEND is responsible for considering each combination of IT-pairs appearing in the prefix class $[P]$. For each IT-pair $X_i \times t(X_i)$ (Line 4), it combines it with the other IT-pairs $X_j \times t(X_j)$ that come after it (Line 6) according to the total order f. Each X_i generates a new prefix class $[P_i]$ which is initially empty (Line 5). At line 7, the two IT-pairs are combined to produce a new pair $\mathbf{X} \times \mathbf{Y}$, where $\mathbf{X} = X_i \cup X_j$ and $\mathbf{Y} = t(X_i) \cap t(X_j)$. Line 8 tests which of the four IT-pair properties can be applied by calling CHARM-PROPERTY. Note that this routine may modify the current class $[P]$ by deleting IT-pairs that are already subsumed by other pairs. It also inserts the newly generated IT-pairs in the new class $[P_i]$. Once all X_j have been processed, we recursively explore the new class $[P_i]$ in a depth-first manner (Line 9). We then insert the itemset $\mathbf{X}$, an extension of X_i, in the set of closed itemsets $\mathcal{C}$ (Line 11), provided that $\mathbf{X}$ is not subsumed by a previously found closed set (we describe later how to perform fast subsumption checking). At this stage any closed itemset containing X_i has already been generated. We then return to Line 4 to process the next (unpruned) IT-pair in $[P]$.

4.1 Dynamic Element Reordering

We purposely let the IT-pair ordering function in Line 6 remain unspecified. The usual manner of processing is in lexicographic order, but we can specify any other total order we want. The most promising approach is to sort the itemsets based on their support. The motivation is to increase opportunity for pruning elements from a class $[P]$. A quick look at Properties 1 and 2 tells us that these two cases are preferable over the other two. For Property 1, the closure of the two itemsets is equal, and thus we can discard X_j and replace X_i with $X_i \cup X_j$. For Property 2, we can still replace X_i with $X_i \cup X_j$. Note that in both these cases we do not insert anything in the new class $[P_i]$! Thus the more the occurrence of cases 1 and 2, the fewer levels of search we perform. In contrast, the occurrence of cases 3 and 4 results in additions to the set of new nodes, requiring additional levels of processing.

Since we want $t(X_i) = t(X_j)$ (Property 1) or $t(X_i) \subset t(X_j)$ (Property 2) it follows that we should sort the itemsets in increasing order of their support. Thus larger tidsets occur later in the ordering and we

```
CHARM (D, min_sup):
1.  [P] = {X_i × t(X_i) : X_i ∈ I ∧ σ(X_i) ≥ min_sup}
2.  CHARM-EXTEND ([P], C = ∅)
3.  return C //all closed sets

CHARM-EXTEND ([P], C):
4.  for each X_i × t(X_i) in [P]
5.      [P_i] = ∅ and X = X_i
6.      for each X_j × t(X_j) in [P], with X_j ≥_f X_i
7.          X = X ∪ X_j and Y = t(X_i) ∩ t(X_j)
8.          CHARM-PROPERTY([P], [P_i])
9.      if ([P_i] ≠ ∅) then CHARM-EXTEND ([P_i], C)
10.     delete [P_i]
11.     C = C ∪ X //if X is not subsumed

CHARM-PROPERTY ([P], [P_i]):
12. if (σ(X) ≥ minsup) then
13.     if t(X_i) = t(X_j) then //Property 1
14.         Remove X_j from [P]
15.         Replace all X_i with X
16.     else if t(X_i) ⊂ t(X_j) then //Property 2
17.         Replace all X_i with X
18.     else if t(X_i) ⊃ t(X_j) then //Property 3
19.         Remove X_j from [P]
20.         Add X × Y to [P_i] //use ordering f
21.     else if t(X_i) ≠ t(X_j) then //Property 4
22.         Add X × Y to [P_i] //use ordering f
```

Figure 5. *The* CHARM *Algorithm*

maximize the occurrence of Properties 1 and 2. By similar reasoning, sorting by decreasing order of support doesn't work very well, since it maximizes the occurrence of Properties 3 and 4, increasing the number of levels of processing. Note also that elements are added in sorted order to each new class $[P_i]$ (lines 20 and 22). Thus the reordering is applied recursively at each node in the tree.

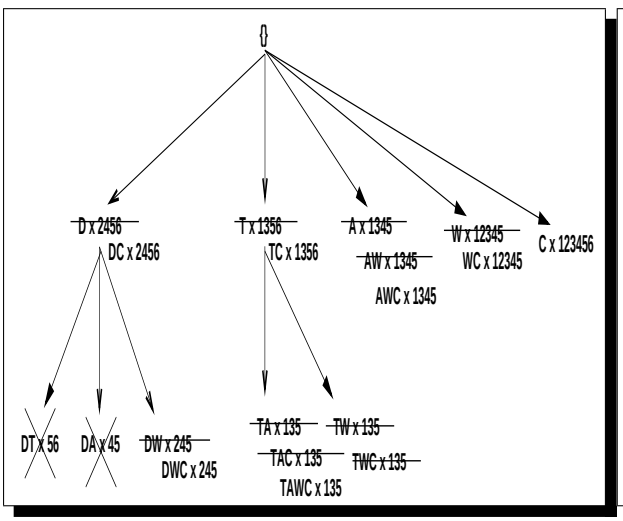

Figure 6. *Search Process using Tidsets*

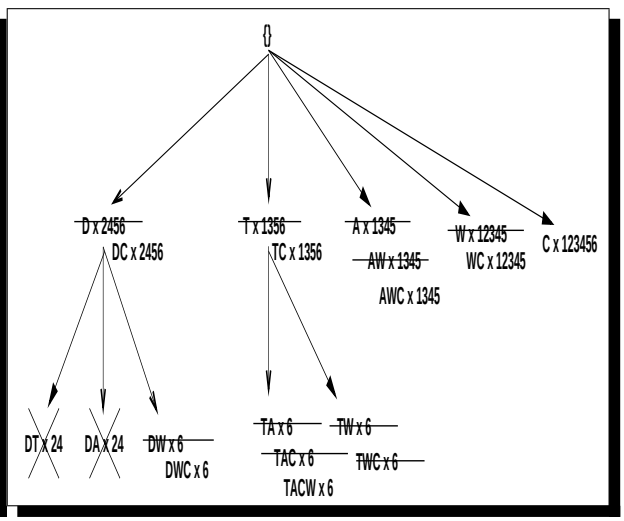

Figure 7. *Search Process using Diffsets*

Example 1: Figure 6 shows how CHARM works on our example database if we sort itemsets in increasing order of support. We will use the pseudo-code to illustrate the computation. Define the weight of an item X as $w(X) = \sum_{XY \in F_2} \sigma(XY)$, i.e., the sum of the support of frequent 2-itemsets that contain the item. At the root level, we sort the items in increasing order of their weights. For example, if we look at the support of 2-itemsets containing A, we find that AC and AW have support 4, while AT has support 3, thus $w(A) = 4 + 4 + 3 = 11$. The final sorted order of items is then D, T, A, W, and C (their weights are 7,10,11,15, and 17, respectively).

We initialize the root class as $[\emptyset] = \{D \times 2456, T \times 1356, A \times 1345, W \times 12345, C \times 123456\}$ in Line 1. At Line 4 we first process the node $D \times 2456$ (we set $\mathbf{X} = D$ in Line 5); it will be combined with the remaining elements in Line 6. DT and DA are not frequent and are pruned. We next look at D and W; since $t(D) \neq t(W)$, property 4 applies, and we simply insert DW in $[D]$ (line 22). We next find that $t(D) \subset t(C)$. Since property 2 applies, we replace all occurrences of D with DC, which means that we also change $[D]$ to $[DC]$, and the element DW to DWC. We next make a recursive call to CHARM-EXTEND with class $[DC]$. Sine there is only one element, we jump to line 11, where DWC is added to the frequent closed set $\mathcal{C}$. When we return the D (now DC) branch is complete, thus DC itself is added to $\mathcal{C}$.

When we process T, we find that $t(T) \neq t(A)$, thus we insert TA in the new class $[T]$ (property 4). Next we find that $t(T) \neq t(W)$ and we get $[T] = \{TA, TW\}$. When we find $t(T) \subset t(C)$ we update all occurrences of T with TC (by property 2). We thus get the class $[TC] = \{TAC, TWC\}$. CHARM then makes a recursive call on Line 9 to process $[TC]$. We try to combine TAC with TWC to find $t(TAC) = t(TWC)$. Since property 1 is satisfied, we replace TAC with $TACW$, deleting TWC at the same time. Since $TACW$ cannot be extended further, we insert it in $\mathcal{C}$, and when we are done processing branch TC, it too is added to $\mathcal{C}$. All other branches satisfy property 2, and no new recursion is made; the final $\mathcal{C}$ consists of the uncrossed IT-pairs shown in Figure 6.

4.2 Fast Subsumption Checking

Let X_i and X_j be two itemsets, we say that an itemset X_i *subsumes* another itemset X_j, if and only if $X_j \subset X_i$ and $\sigma(X_j) = \sigma(X_i)$. Recall that before adding a set $\mathbf{X}$ to the current set of closed patterns $\mathcal{C}$, CHARM makes a check in Line 11 (see Figure 5) to see if $\mathbf{X}$ is subsumed by some closed set in $\mathcal{C}$. In other words, it may happen that after adding a closed set Y to $\mathcal{C}$, when we explore subsequent branches, we may generate another set X, which cannot be extended further, with $X \subseteq Y$ and with $\sigma(Y) = \sigma(X)$. In this case, X is a non-closed set subsumed by Y, and it should not be added to $\mathcal{C}$. Since $\mathcal{C}$ dynamically expands during enumeration of closed patterns, we need a very fast approach to perform such subsumption checks.

Clearly we want to avoid comparing $\mathbf{X}$ with all existing elements in $\mathcal{C}$, for this would lead to a $O(|\mathcal{C}|^2)$ complexity. To quickly retrieve relevant closed sets, the obvious solution is to store $\mathcal{C}$ in a hash table. But what hash function to use? Since we want to perform subset checking, we can't hash on the itemset. We could use the support of the itemsets for the hash function. But many unrelated itemsets may have the same support. Since CHARM uses IT-pairs throughout its search, it seems reasonable to use the information from the tidsets to help identify if $\mathbf{X}$ is subsumed. Note that if $t(X_j) = t(X_i)$, then obviously $\sigma(X_j) = \sigma(X_i)$. Thus to check if $\mathbf{X}$ is subsumed, we can check if $t(\mathbf{X}) = t(C)$ for some $C \in \mathcal{C}$. This check can be performed in $O(1)$ time using a hash table. But obviously we cannot afford to store the actual tidset with each closed set in $\mathcal{C}$; the space requirements would be prohibitive.

CHARM adopts a compromise solution. It computes a hash function on the tidset and stores in the hash table a closed set along with its support (in our implementation we used the C++ STL – standard template library – hash_multimap container for the hash table). Let $h(X_i)$ denote a suitable chosen hash function on the tidset $t(X_i)$. Before adding $\mathbf{X}$ to $\mathcal{C}$, we retrieve from the hash table all entries with the hash key $h(\mathbf{X})$. For each matching closed set C is then check if $\sigma(\mathbf{X}) = \sigma(C)$. If yes, we next check if $\mathbf{X} \subset C$. If yes, then $\mathbf{X}$ is subsumed and we do not add it to hash table $\mathcal{C}$.

What is a good hash function on a tidset? CHARM uses the sum of the tids in the tidset as the hash function, i.e., $h(\mathbf{X}) = \sum_{T \in t(\mathbf{X})} T$ (note, this is not the same as support, which is the cardinality of $t(\mathbf{X})$). We tried several other variations and found there to be no performance difference. This hash function is likely to be as good as any other due to several reasons. Firstly, by definition a closed set is one that does not have a superset with the same support; it follows that it must have some tids that do not appear in any other closed set. Thus the hash keys of different closed sets will tend to be different. Secondly, even if there are several closed sets with the same hash key, the support check we perform (i.e., if $\sigma(\mathbf{X}) = \sigma(C)$) will eliminate many closed sets whose keys are the same, but they in fact have different supports. Thirdly, this hash function is easy to compute and it can easily be used with the diffset format we introduce next.

4.3 Diffsets for Fast Frequency Computations

Given that we are manipulating itemset-tidset pairs, CHARM uses a *vertical* data format, where we maintain a disk-based tidset for each item in the database. Mining algorithms using the vertical format have shown to be very effective and usually outperform horizontal approaches [8, 16, 17, 19]. The main benefits of using a vertical format are: 1) Computing the supports is simpler and faster. Only intersections on tidsets is required, which are also well-supported by current databases. The horizontal approach on the other hand requires complex hash trees. 2) There is automatic pruning of irrelevant information as the intersections proceed; only tids relevant for frequency determination remain after each intersection. 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 [8]. Further, vertical bitmaps offer scope for compression [17].

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 utilize a vertical data representation called *diffsets*, that we recently proposed [20]. Diffsets keep track of differences in the tids of a candidate pattern from its parent frequent pattern. These differences are propagated all the way from one node to its children starting from the root. We showed in [20] that diffsets drastically cut down (by orders of magnitude) the size of memory required to store intermediate results. 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 very efficiently.

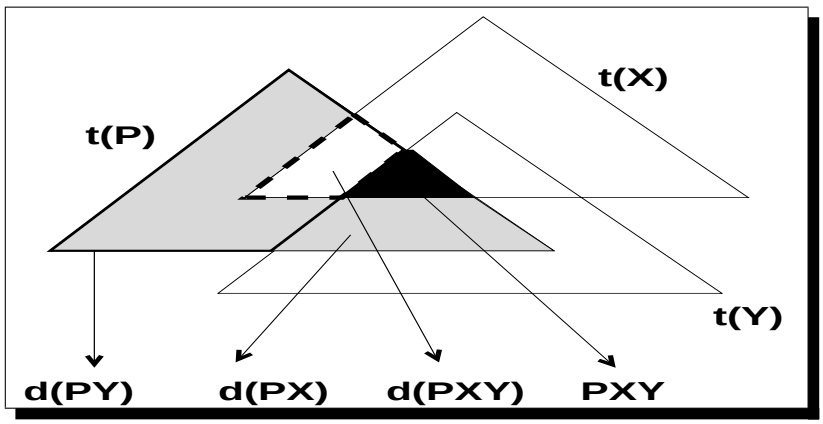

Figure 8. *Diffsets: Prefix P and class members X and Y*

More formally, consider a class with prefix P. Let $d(X)$ denote 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 $d(XY)$ as a difference of tidsets $t(PX) - t(PY)$, we compute it as the difference of the diffsets $d(PY) - d(PX)$. Figure 8 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. Diffsets are typically much smaller than storing the tidsets with each child since only the essential changes are propagated from a node to its children. Diffsets also shrink as longer itemsets are found.

Diffsets and Subsumption Checking: Notice that diffsets cannot be used directly for generating a hash key as was possible with tidsets. The reason is that depending on the class prefix, nodes in different branches will have different diffsets, even though one is subsumed by the other. The solution is to keep track of the hash key $h(PXY)$ for PXY in the same way as we store $\sigma(PXY)$. In other words, assume

that we have available $h(PX)$, then we can compute $h(PXY) = h(PX) - \sum_{T \in d(PXY)} T$. Of course, this is only possible because of our choice of hash function described in Section 4.2. Thus we associate with each member of a class its hash key, and the subsumption checking proceeds exactly as for tidsets.

Differences and Subset Testing: We assume that the initial database is stored in tidset format, but we use diffsets thereafter. Given the availability of diffsets for each itemset, the computation of the difference for a new combination is straightforward. All it takes is a linear scan through the two diffsets, storing tids in one but not the other. The main question is how to efficiently compute the subset information, while computing differences, required for applying the four IT-pair properties. At first this might appear like an expensive operation, but in fact it comes for free as an outcome of the set difference operation. While taking the difference of two sets we keep track of the number of mismatches in both the diffsets, i.e., the cases when a tid occurs in one list but not in the other. Let $m(X_i)$ and $m(X_j)$ denote the number of mismatches in the diffsets $d(X_i)$ and $d(X_j)$. There are four cases to consider:

$$\begin{aligned}
&m(X_i) = 0 \text{ and } m(X_j) = 0, \quad \text{then } d(X_i) = d(X_j) \text{ or } t(X_i) = t(X_j) \text{ — Property 1}\\
&m(X_i) > 0 \text{ and } m(X_j) = 0, \quad \text{then } d(X_i) \supset d(X_j) \text{ or } t(X_i) \subset t(X_j) \text{ — Property 2}\\
&m(X_i) = 0 \text{ and } m(X_j) > 0, \quad \text{then } d(X_i) \subset d(X_j) \text{ or } t(X_i) \supset t(X_j) \text{ — Property 3}\\
&m(X_i) > 0 \text{ and } m(X_j) > 0, \quad \text{then } d(X_i) \neq d(X_j) \text{ or } t(X_i) \neq t(X_j) \text{ — Property 4}
\end{aligned}$$

Thus CHARM performs support, subset, equality, and inequality testing simultaneously while computing the difference itself. Figure 7 shows the search for closed sets using diffsets instead of tidsets. The exploration proceeds in exactly the same way as described in Example 1. However, this time we perform difference operations on diffsets (except for the root class, which uses tidsets). Consider an IT-pair like $TAWC \times 6$. Since this indicates that $TAWC$ differs from its parent $TC \times 1356$ only in the tid 6, we can infer that the real IT-pair should be $TAWC \times 135$.

4.4 Other optimizations and Correctness

Optimized Initialization: There is only one significant departure from the pseudo-code in Figure 5. Note that if we initialize the $[P]$ set in Line 1 with all frequent items, and invoke CHARM-Extend then, in the worst case, we might perform $n(n-1)/2$ difference operations where n is the number of frequent items. It is well known that many itemsets of length 2 turn out to be infrequent, thus it is clearly wasteful to perform $O(n^2)$ operations. To solve this performance problem we first compute the set of frequent itemsets of length 2, and then we add a simple check in Line 6 (not shown for clarity; it only applies to 2-itemsets), so that we combine two items X_i and X_j only if $X_i \cup X_j$ is known to be frequent. The number of operations performed after this check is equal to the number of frequent pairs, which in practice is closer to $O(n)$ rather than $O(n^2)$. To compute the frequent itemsets of length 2 using the vertical format we perform a multi-stage vertical-to-horizontal transformation on-the-fly as described in [19], over distinct ranges of tids. Given a recovered horizontal database chunk it is straightforward to update the count of pairs of items using an upper triangular 2D array. We then process the next chunk. The horizontal chunks are thus temporarily materialized in memory and then discarded after processing [19].

Memory Management: Since CHARM processes branches in a depth-first fashion, its memory requirements are not substantial. It has to retain all the itemset-diffsets pairs on the levels of the current left-most branches in the search space. The use of diffsets also reduces drastically the memory consumption. For cases where even the memory requirement of depth-first search and diffsets exceed available memory, it is straightforward to modify CHARM to write/read temporary diffsets to/from disk, as in [17].

Theorem 2 (correctness). CHARM *enumerates all frequent closed itemsets.*
Proof*:* CHARM correctly identifies all and only the closed frequent itemsets, since its search is based on a complete IT-tree search space. The only branches that are pruned as those that either do not have sufficient support, or those that are subsumed by another closed set based on the properties of itemset-tidset pairs as outlined in Theorem 1. Finally CHARM eliminates any non-closed itemset that might be generated by performing subsumption checking before inserting anything in the set of all frequent closed itemsets $\mathcal{C}$. ■

5 Experimental Evaluation

Experiments were performed on a 400MHz Pentium PC with 256MB of memory, running RedHat Linux 6.0. Algorithms were coded in C++. For performance comparison we used the original source or object code for Close [14], Pascal [3], Closet [15] and Mafia [6], all provided to us by their authors. The original Closet code had a subtle bug, which affected the performance, but not the correctness. Our comparison below uses the new bug-free, optimized version of Closet obtained from its authors. Mafia has an option to mine only closed sets instead of maximal sets. We refer to this version of Mafia below. We also include a comparison with the base Apriori algorithm [1] for mining all itemsets. Timings in the figures below are based on total wall-clock time, and include all preprocessing costs (such as vertical database creation in CHARM and Mafia).

Benchmark Datasets We chose several real and synthetic database benchmarks [1, 4], publicly available from IBM Almaden (www.almaden.ibm.com/cs/quest/demos.html), for the performance tests. The PUMS datasets (pumsb and pumsb*) 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. The connect and chess datasets are derived from their respective game steps. The latter three datasets were originally taken from the UC Irvine Machine Learning Database Repository. The synthetic datasets (T10 and T40), using the IBM generator, mimic the transactions in a retailing environment.

The gazelle dataset comes from click-stream data from a small dot-com company called Gazelle.com, a legware and legcare retailer, which no longer exists. A portion of this dataset was used in the KDD-Cup 2000 competition. This dataset was recently made publicly available by Blue Martini Software (download it from www.ecn.purdue.edu/KDDCUP).

Typically, the real datasets are very dense, i.e., they produce many long frequent itemsets even for very high values of support. The synthetic datasets mimic the transactions in a retailing environment. Usually the synthetic datasets are sparser when compared to the real sets.

Database	# Items	Avg. Length	Std. Dev.	# Records	Max. Pattern (sup)	Levels Searched
chess	76	37	0	3,196	23 (20%)	17
connect	130	43	0	67,557	29 (10%)	12
mushroom	120	23	0	8,124	21 (0.075%)	11
pumsb*	7117	50	2	49,046	40 (5%)	16
pumsb	7117	74	0	49,046	22 (60%)	17
gazelle	498	2.5	4.9	59,601	154 (0.01%)	11
T10I4D100K	1000	10	3.7	100,000	11 (0.025%)	11
T40I10D100K	1000	40	8.5	100,000	25 (0.001%)	19

Table 1. *Database Characteristics*

Table 1 shows the characteristics of the real and synthetic datasets used in our evaluation. It shows the number of items, the average transaction length, the standard deviation of transaction lengths, and the number of records in each database. The table additionally shows the length of the longest maximal pattern (at the lowest minimum support used in our experiments) for the different datasets, as well as the maximum level of search that CHARM performed to discover the longest pattern. For example on gazelle, the longest closed pattern was of length 154 (any method that mines all frequent patterns will be impractical for such long patterns), yet the maximum recursion depth in CHARM was only 11! The number of levels skipped is also considerable for other real datasets. The synthetic dataset $T10$ is extremely sparse and no levels are skipped, but for $T40$ 6 levels were skipped. These results give an indication of the effectiveness of CHARM in mining closed patterns, and are mainly due to repeated applications of Properties 1 and 2 in Theorem 1.

Before we discuss the performance results of different algorithms it is instructive to look at the total number of frequent closed itemsets and distribution of closed patterns by length for the various datasets, as shown in Figure 9. We have grouped the datasets according to the type of distribution. chess, pumsb*, pumsb, and connect all display an almost symmetric distribution of the closed frequent patterns with different means. $T40$ and mushroom display an interesting bi-modal distribution of closed sets. $T40$, like $T10$, has a many short patterns of length 2, but it also has another peak at length 6. mushroom has considerably longer patterns; its second peak occurs at 19. Finally gazelle and T10 have a right-skewed distribution. gazelle

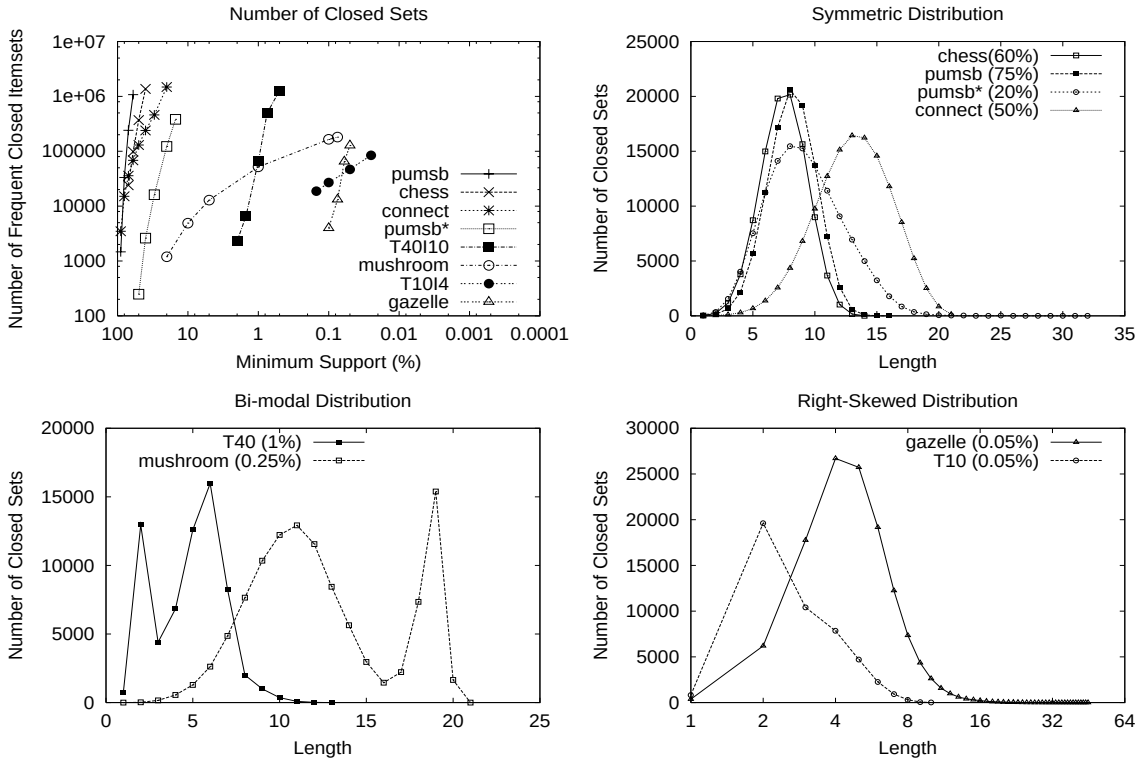

Figure 9. *Number of Frequent Closed Itemsets and Distribution by Length*

tends to have many small patterns, with a very long right tail. $T10$ exhibits a similar distribution, with the majority of the closed patterns begin of length 2! The type of distribution tends to influence the behavior of different algorithms as we will see below.

5.1 Performance Testing

We compare the performance of CHARM against Apriori, Close, Pascal, Mafia and Closet in Figure 10. Since Closet was provided as a Windows executable by its authors, we compared it separately on a 900 MHz Pentium III processor with 256MB memory, running Windows 98. In [21] we tested a modified version of a maximal pattern finding algorithm (MaxMiner [4]) to discover closed sets in a post-processing step, and found it to be too slow for all except short patterns.

Symmetric Datasets Let us first compare how the methods perform on datasets which exhibit a symmetric distribution of closed itemsets, namely chess, pumsb, connect and pumsb*. We observe that Apriori, Close and Pascal work only for very high values of support on these datasets. The best among the three is Pascal which can be twice as fast as Close, and up to 4 times better than Apriori. On the other hand, CHARM is several orders of magnitude better than Pascal, and it can be run on very low support values, where none of the former three methods can be run. Comparing with Mafia, we find that both CHARM and Mafia have similar performance for higher support values. However, as we lower the minimum support, the performance gap between CHARM and Mafia widens. For example at the lowest support value plotted, CHARM is about 30 times faster than Mafia on Chess, about 3 times faster on connect and pumsb, and 4 times faster on pumsb*. CHARM outperforms Closet by an order of magnitude or more, especially as support is lowered. On chess and pumsb* it is about 10 times faster than Closet, and about 40 times faster on pumsb. On connect Closet performs better at high supports, but CHARM does better at lower supports. The reason is that connect has transactions with lot of overlap among items, leading to a compact FP-tree and to faster performance. However, as support is lowered FP-tree starts to grow, and Closet loses its edge.

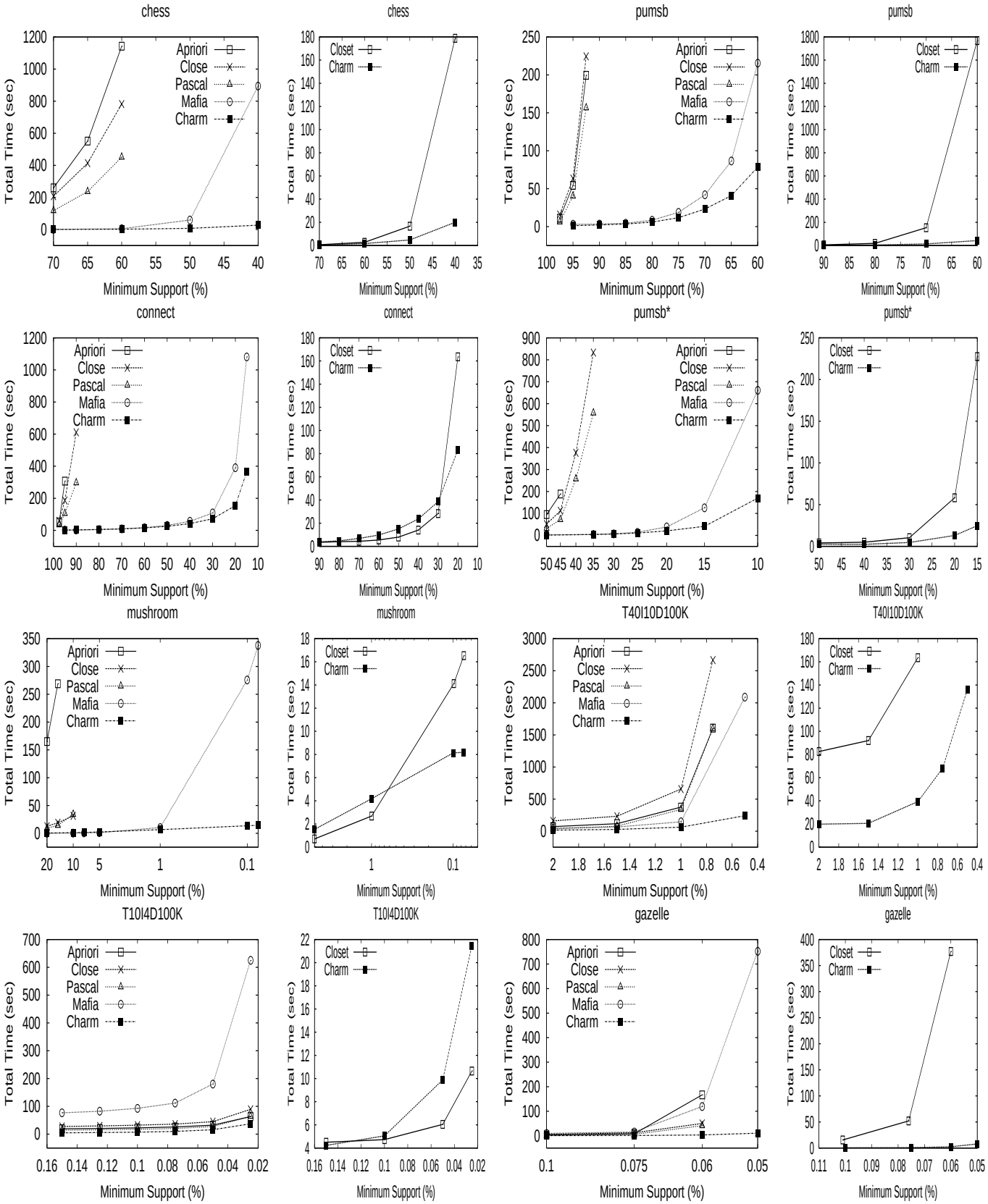

Figure 10. CHARM *versus Apriori, Close, Pascal, Mafia, and Closet*

Bi-modal Datasets On the two datasets with a bi-modal distribution of frequent closed patterns, namely mushroom and $T40$, we find that Pascal fares better than for symmetric distributions. For higher values of support the maximum closed pattern length is relatively short, and the distribution is dominated by the first mode. Apriori, Close and Pascal can hand this case. However, as one lowers the minimum support the second mode starts to dominate, with longer patterns. These these methods thus quickly lose steam and become uncompetitive. Between CHARM and Mafia, up to 1% minimum support there is negligible difference, however, when the support is lowered there is a huge difference in performance. CHARM is about 20 times faster on mushroom and 10 times faster on $T40$ for the lowest support shown. The gap continues to widen sharply. We find that CHARM outperforms Closet by a factor of 2 for mushroom and 4 for $T40$.

Right-Skewed Datasets On gazelle and $T10$, which have a large number of very short closed patterns, followed by a sharp drop, we find that Apriori, Close and Pascal remain competitive even for relatively low supports. The reason is that $T10$ had a maximum pattern length of 11 at the lowest support shown. Also gazelle at 0.06% support also had a maximum pattern length of 11. The level-wise search of these three methods is able to easily handle such short patterns. However, for gazelle, we found that at 0.05% support the maximum pattern length suddenly jumped to 45, and none of these three methods could be run.

$T10$, though a sparse dataset, is problematic for Mafia. The reason is that $T10$ produces long sparse bitvectors for each item, and offers little scope for bit-vector compression and projection that Mafia relies on for efficiency. This causes Mafia to be uncompetitive for such datasets. Similarly Mafia fails to do well on gazelle. However, it is able to run on the lowest support value. The diffset format of CHARM is resilient to sparsity (as shown in [20]) and it continues to outperform other methods. For the lowest support, on $T10$ it is twice as fast as Pascal and 15 times better than Mafia, and it is about 70 times faster than Mafia on gazelle. CHARM is about 2 times slower than Closet on $T10$. The reason is that the majority of closed sets are of length 2, and the tidset/diffsets operations in CHARM are relatively expensive compared to the compact FP-tree for short patterns (max length is only 11). However, for gazelle, which has much longer closed patterns, CHARM outperforms Closet by a factor of 160!

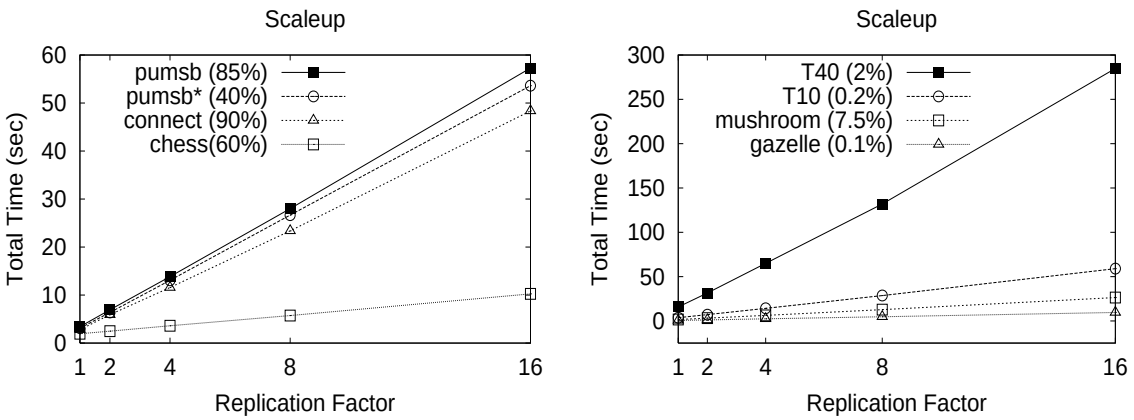

Figure 11. *Size Scaleup on Different Datasets*

Scaling Properties of CHARM Figure 11 shows how CHARM scales with an increasing number of transactions. For this study, we kept all database parameters constant, and replicated the transactions from 2 to 16 times. Thus, for example, for $T40$, which has 100K transactions initially, at a replication factor of 16, it will have 1.6 million transactions. At a given level of support, we find a linear increase in the running time with increasing number of transactions.

Effect of Branch Ordering Figure 12 compares three ordering methods — lexicographic order, increasing by support, and decreasing by support. Decreasing order is the worst, while processing class elements in increasing order is the best. Similar results were obtained for the other datasets. All results for CHARM reported above used increasing branch ordering.

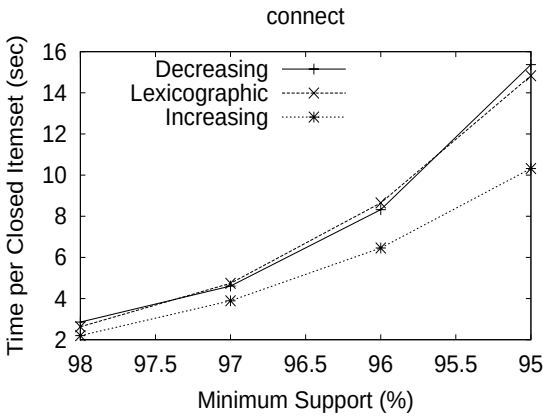

Figure 12. *Branch Ordering*

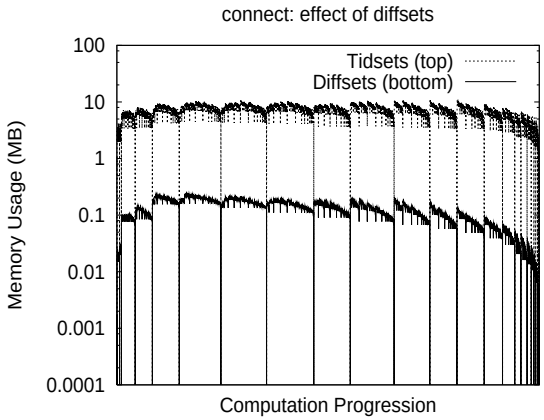

Figure 13. *Memory Usage (80% minsup)*

DB	50%	20%	DB	0.1%	0.05%	DB	1%	0.5%
connect	0.68MB	1.17MB	gazelle	0.13MB	1.24MB	T40I10D100K	0.39MB	0.52MB

Table 2. *Maximum Memory Usage (Using Diffsets)*

Memory Usage Figure 13 shows how the memory usage, for storing the tidsets and diffsets, changes as computation progresses. The total usage for tidsets is generally under 10MB, but for diffsets it is under 0.2MB, a reduction by a factor of 50! The sharp drop to 0 (the vertical lines) indicates the beginning of a new prefix class. Table 2 shows the maximum memory usage for three datasets for different values of support. Here also we see that the memory footprint using diffsets is extremely small even for low values of support. These results confirm that for many datasets the intermediate diffsets can easily fit in memory.

6 Conclusions

We presented and evaluated CHARM, an efficient algorithm for mining closed frequent itemsets. CHARM simultaneously explores both the itemset space and tidset space using the new IT-tree framework, which allows it to use a novel search method that skips many levels to quickly identify the closed frequent itemsets, instead of having to enumerate many non-closed subsets. We utilized a new vertical format based on diffsets, i.e., storing the differences in the tids as the computation progresses. An extensive set of experiments confirms that CHARM can provide orders of magnitude improvement over existing methods for mining closed itemsets. It also scales linearly in the number of transactions.

It has been shown in recent studies that closed itemsets can help in generating non-redundant rules sets, which are typically a lot smaller than the set of all association rules [18]. An interesting direction of future work is to develop methods to mine closed patterns for other mining problems like sequences, episodes, multi-dimensional patterns, etc., and to study how much reduction in their respective rule sets is possible. It also seems worthwhile to explore if the concept of "closure" extends to metrics other than support. For example, for confidence, correlation, etc.

Acknowledgments

We would like to thank Lotfi Lakhal and Yves Bastide for providing us the source code for Close and Pascal, Jiawei Han, Jian Pei, and Jianyong Wang for sending us the executable for Closet, and Johannes Gehrke for the Mafia algorithm. We thanks Roberto Bayardo for providing us the IBM real datasets, and Ronny Kohavi and Zijian Zheng of Blue Martini Software for giving us access to the Gazelle dataset.

Bibliography

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 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] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *SIGKDD Explorations*, 2(2), December 2000.

[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] D. Cristofor, L. Cristofor, and D. Simovici. Galois connection and data mining. *Journal of Universal Computer Science*, 6(1):60–73, 2000.

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

[9] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.

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

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

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

[14] 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.

[15] 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.

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

[17] P. Shenoy, J.R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *ACM SIGMOD Intl. Conf. Management of Data*, May 2000.

[18] M. J. Zaki. Generating non-redundant association rules. In *6th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, August 2000.

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

[20] M. J. Zaki and K. Gouda. Fast vertical mining using Diffsets. Technical Report 01-1, Computer Science Dept., Rensselaer Polytechnic Institute, March 2001.

[21] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed association rule mining. Technical Report 99-10, Computer Science Dept., Rensselaer Polytechnic Institute, October 1999.