CHARM: An Efficient Algorithm for Closed Association Rule Mining

Mohammed J. Zaki and Ching-Jui Hsiao
Computer Science Department
Rensselaer Polytechnic Institute, Troy NY 12180
{zaki,hsiaoc}@cs.rpi.edu
http://www.cs.rpi.edu/~zaki

Abstract

The task of mining association rules consists of two main steps. The first involves finding the set of all frequent itemsets. The second step involves testing and generating all high confidence rules among itemsets. In this paper we show that it is not necessary to mine all frequent itemsets in the first step, instead it is sufficient to mine the set of *closed* frequent itemsets, which is much smaller than the set of all frequent itemsets. It is also not necessary to mine the set of all possible rules. We show that any rule between itemsets is equivalent to some rule between closed itemsets. Thus many redundant rules can be eliminated. Furthermore, we present CHARM, an efficient algorithm for mining all closed frequent itemsets. An extensive experimental evaluation on a number of real and synthetic databases shows that CHARM outperforms previous methods by an order of magnitude or more. It is also linearly scalable in the number of transactions and the number of closed itemsets found.

1 Introduction

Since its introduction, Association Rule Mining [1], has become one of the core data mining tasks, and has attracted tremendous interest among data mining researchers and practitioners. It has an elegantly simple problem statement, that is, to find the set of all subsets of items (called itemsets) that frequently occur in many database records or transactions, and to extract the rules telling us how a subset of items influences the presence of another subset.

The prototypical application of associations is in *market basket analysis*, where the items represent products and the records the point-of-sales data at large grocery or departmental stores. These kinds of database are generally sparse, i.e., the longest frequent itemsets are relatively short. However there are many real-life datasets that very dense, i.e., they contain very long frequent itemsets.

It is widely recognized that the set of association rules can rapidly grow to be unwieldy, especially as we lower the frequency requirements. The larger the set of frequent itemsets the more the number of rules presented to the user, many of which are redundant. This is true even for sparse datasets, but for dense datasets it is simply not feasible to mine all possible frequent itemsets, let alone to generate rules between itemsets. In such datasets one typically finds an exponential number of frequent itemsets. For example, finding long itemsets of length 30 or 40 is not uncommon [2].

In this paper we show that it is not necessary to mine all frequent itemsets to guarantee that all non-redundant association rules will be found. We show that it is sufficient to consider only the closed frequent itemsets (to be defined later). Further, all non-redundant rules are found by only considering rules among the closed frequent itemsets. The set of closed frequent itemsets is a lot smaller than the set of all frequent itemsets, in some cases by 3 or more orders of magnitude. Thus even in dense domains we can guarantee completeness, i.e., all non-redundant association rules can be found.

The main computation intensive step in this process is to identify the closed frequent itemsets. It is not possible to generate this set using Apriori-like [1] bottom-up search methods that examine all subsets of a frequent itemset. Neither is it possible to mine these sets using algorithms for mining maximal frequent patterns like MaxMiner [2] or Pincer-Search [9], since to find the closed itemsets all subsets of the maximal frequent itemsets would have to be examined.

We introduce CHARM, an efficient algorithm for enumerating the set of all closed frequent itemsets. CHARM is unique in that it simultaneously explores both the itemset space and transaction space, unlike all previous association mining methods which only exploit the itemset search space. Furthermore, CHARM avoids enumerating all possible subsets of a closed itemset when enumerating the closed frequent sets.

The exploration of both the itemset and transaction space allows CHARM 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. Further, CHARM uses a two-pronged pruning strategy. It prunes candidates based not only on subset infrequency (i.e., no extensions of an infrequent are tested) as do all association mining methods, but it also prunes candidates based on non-closure property, i.e., any non-closed itemset is pruned. Finally, CHARM uses no internal data structures like Hash-trees [1] or Tries [3]. The fundamental operation used is an union of two itemsets and an intersection of two transactions lists where the itemsets are contained.

An extensive set of experiments confirms that CHARM provides orders of magnitude improvement over existing methods for mining closed itemsets, even over methods like AClose [14], that are specifically designed to mine closed itemsets. It makes a lot fewer database scans than the longest closed frequent set found, and it scales linearly in the number of transactions and also is also linear in the number of closed itemsets found.

The rest of the paper is organized as follows. Section 2 describes the association mining task. Section 3 describes the benefits of mining closed itemsets and rules among them. We present CHARM in Section 4. Related work is discussed in Section 5. We present experiments in Section 6 and conclusions in Section 7.

2 Association Rules

The association mining task can be stated as follows: Let $\mathcal{I} = \{1, 2, \cdots, m\}$ be a set of items, and let $\mathcal{T} = \{1, 2, \cdots, n\}$ be a set of transaction identifiers or *tids*. The input database is a binary relation $\delta \subseteq \mathcal{I} \times \mathcal{T}$. If an item i occurs in a transaction t, we write it as $(i, t) \in \delta$, or alternately as $i\delta t$. Typically the database is arranged as a set of transaction, where each transaction contains a set of items. For example, consider the database shown in Figure 1, used as a running example throughout this paper. Here $\mathcal{I} = \{A, C, D, T, W\}$, and $\mathcal{T} = \{1, 2, 3, 4, 5, 6\}$. The second transaction can be represented as $\{C\delta 2, D\delta 2, W\delta 2\}$; all such pairs from all transactions, taken together form the binary relation δ.

A set $X \subseteq \mathcal{I}$ is also called an *itemset*, and a set $Y \subseteq \mathcal{T}$ is called a *tidset*. 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 (minsup)* value, i.e., if $\sigma(X) \geq$ *minsup*.

An *association rule* is an expression $X_1 \xrightarrow{p} X_2$, where X_1 and X_2 are itemsets, and $X_1 \cap X_2 = \emptyset$. The *support* of the rule is given as $\sigma(X_1 \cup X_2)$ (i.e., the joint probability of a transaction containing both X_1 and X_2), and the *confidence* as $p = \sigma(X_1 \cup X_2)/\sigma(X_1)$ (i.e., the conditional probability that a transaction contains X_2, given that it contains X_1). A rule is frequent if the itemset $X_1 \cup X_2$ is frequent. A rule is *confident* if its confidence is greater than or equal to a user-specified *minimum confidence (minconf)* value, i.e, $p \geq$ *minconf*.

The association rule mining task consists of two steps [1]: 1) Find all frequent itemsets, and 2) Generate high confidence rules.

Finding frequent itemsets This step is computationally and I/O intensive. Consider Figure 1, which shows a bookstore database with six customers who buy books by different authors. It shows all the frequent itemsets with $minsup = 50\%$ (i.e., 3 transactions). $ACTW$ and CDW are the maximal-by-inclusion frequent itemsets (i.e., they are not a subset of any other frequent itemset).

Let $|\mathcal{I}| = m$ be the number of items. The search space for enumeration of all frequent itemsets is 2^m, which is exponential in m. One can prove that the problem of finding a frequent set of a certain size is NP-Complete, by reducing it to the balanced bipartite clique problem, which is known to be NP-Complete [8, 18]. However, if we assume that there is a bound on the transaction length, the task of finding all frequent itemsets is essentially linear in the database size, since the overall complexity in this case is given as $O(r \cdot n \cdot 2^l)$, where $|\mathcal{T}| = n$ is the number of transactions, l is the length of the longest frequent itemset, and r is the number of maximal frequent itemsets.

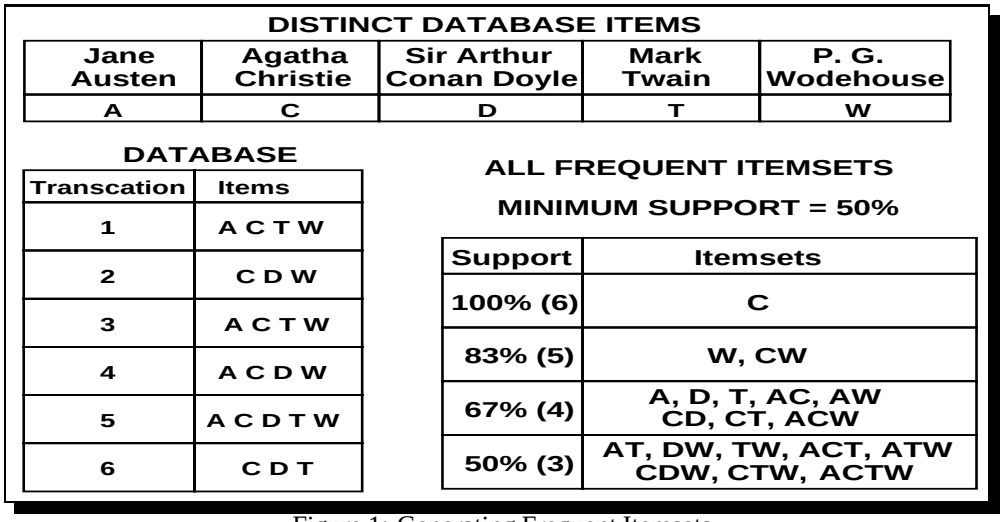

Figure 1: Generating Frequent Itemsets

Generating confident rules This step is relatively straightforward; rules of the form $X' \xrightarrow{p} X - X'$, are generated for all frequent itemsets X (where $X' \subset X$, and $X' \neq \emptyset$), provided $p \geq$ *minconf*. For an itemset of size k there are $2^k - 2$ potentially confident rules that can be generated. This follows from the fact that we must consider each subset of the itemset as an antecedent, except for the empty and the full itemset. The complexity of the rule generation step is thus $O(s \cdot 2^l)$, where s is the number of frequent itemsets, and l is the longest frequent itemset (note that s can be $O(r \cdot 2^l)$, where r is the number of maximal frequent itemsets). For example, from the frequent itemset ACW we can generate 6 possible rules (all of them have support of 4): $A \xrightarrow{1.0} CW, C \xrightarrow{0.67} AW, W \xrightarrow{0.8} AC, AC \xrightarrow{1.0} W, AW \xrightarrow{1.0} C$, and $CW \xrightarrow{0.8} A$.

3 Closed Frequent Itemsets

In this section we develop the concept of closed frequent itemsets, and show that this set is necessary and sufficient to capture all the information about frequent itemsets, and has smaller cardinality than the set of all frequent itemsets.

3.1 Partial Order and Lattices

We first introduce some lattice theory concepts (see [4] for a good introduction).

Let P be a set. A *partial order* on P is a binary relation $\leq$, such that for all $x, y, z \in P$, the relation is: 1) Reflexive: $x \leq x$. 2) Anti-Symmetric: $x \leq y$ and $y \leq x$, implies $x = y$. 3) Transitive: $x \leq y$ and $y \leq z$, implies $x \leq z$. The set P with the relation $\leq$ is called an *ordered set*, and it is denoted as a pair $(P, \leq)$. We write $x < y$ if $x \leq y$ and $x \neq y$.

Let $(P, \leq)$ be an ordered set, and let S be a subset of P. An element $u \in P$ is an *upper bound* of S if $s \leq u$ for all $s \in S$. An element $l \in P$ is a *lower bound* of S if $s \geq l$ for all $s \in S$. The least upper bound is called the **join** of S, and is denoted as $\bigvee S$, and the greatest lower bound is called the **meet** of S, and is denoted as $\bigwedge S$. If $S = \{x, y\}$, we also write $x \vee y$ for the join, and $x \wedge y$ for the meet.

An ordered set $(L, \leq)$ is a *lattice*, if for any two elements x and y in L, the join $x \vee y$ and meet $x \wedge y$ always exist. L is a *complete lattice* if $\bigvee S$ and $\bigwedge S$ exist for all $S \subseteq L$. Any finite lattice is complete. L is called a *join semilattice* if only the join exists. L is called a *meet semilattice* if only the meet exists.

Let $\mathcal{P}$ denote the power set of S (i.e., the set of all subsets of S). The ordered set $(\mathcal{P}(S), \subseteq)$ is a complete lattice, where the meet is given by set intersection, and the join is given by set union. For example the partial orders $(\mathcal{P}(\mathcal{I}), \subseteq)$, the set of all possible itemsets, and $(\mathcal{P}(\mathcal{T}), \subseteq)$, the set of all possible tidsets are both complete lattices.

The set of all frequent itemsets, on the other hand, is only a meet-semilattice. For example, consider Figure 2, which shows the semilattice of all frequent itemsets we found in our example database (from Figure 1). For any two itemsets, only their meet is guaranteed to be frequent, while their join may or may not be frequent. This follows from

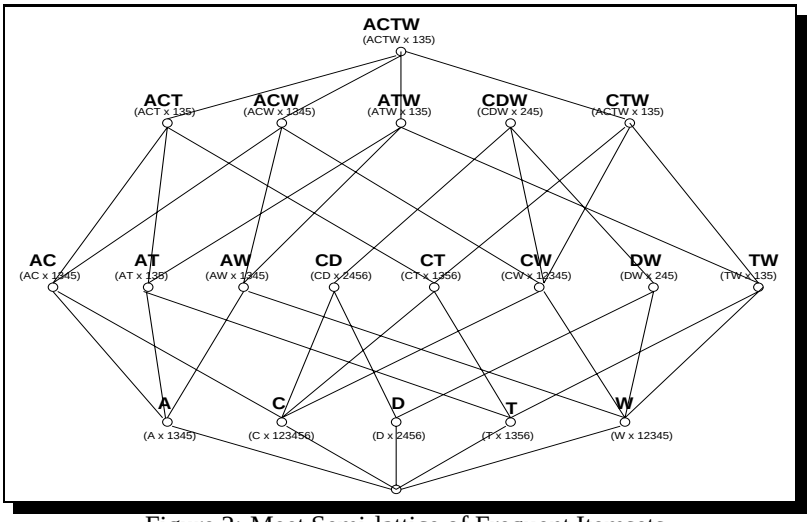

Figure 2: Meet Semi-lattice of Frequent Itemsets

the well known principle in association mining that, if an itemset is frequent, then all its subsets are also frequent. For example, $AC \wedge AT = AC \cap AT = A$ is frequent. For the join, while $AC \vee AT = AC \cup AT = ACT$ is frequent, $AC \cup DW = ACDW$ is not frequent.

3.2 Closed Itemsets

Let the binary relation $\delta \subseteq \mathcal{I} \times \mathcal{T}$ be the input database for association mining. Let $X \subseteq \mathcal{I}$, and $Y \subseteq \mathcal{T}$. Then the mappings

$$t : \mathcal{I} \mapsto \mathcal{T}, \quad t(X) = \{y \in \mathcal{T} \mid \ \forall x \in X, \ x\delta y\}$$
$$i : \mathcal{T} \mapsto \mathcal{I}, \quad i(Y) = \{x \in \mathcal{I} \mid \ \forall y \in Y, \ x\delta y\}$$

define a *Galois connection* between the partial orders $(\mathcal{P}(\mathcal{I}), \subseteq)$ and $(\mathcal{P}(\mathcal{T}), \subseteq)$, the power sets of $\mathcal{I}$ and $\mathcal{T}$, respectively. We denote a $(X, t(X))$ pair as $X \times t(X)$, and a $(i(Y), Y)$ pair as $i(Y) \times Y$. Figure 3 illustrates the two mappings. The mapping $t(X)$ is the set of all transactions (tidset) which contain the itemset X, similarly $i(Y)$ is the itemset that is contained in all the transactions in Y. For example, $t(ACW) = 1345$, and $i(245) = CDW$. In terms of individual elements $t(X) = \bigcap_{x \in X} t(x)$, and $i(Y) = \bigcap_{y \in Y} i(y)$. For example $t(ACW) = t(A) \cap t(C) \cap t(W) = 1345 \cap 123456 \cap 12345 = 1345$. Also $i(245) = i(2) \cap i(4) \cap i(5) = CDW \cap ACDW \cap ACDTW = CDW$.

The Galois connection satisfies the following properties (where $X, X_1, X_2 \in \mathcal{P}(\mathcal{I})$ and $Y, Y_1, Y_2 \in \mathcal{P}(\mathcal{T})$):
1) $X_1 \subseteq X_2 \Rightarrow t(X_1) \supseteq t(X_2)$. For $ACW \subseteq ACTW$, we have $t(ACW) = 1345 \supseteq 135 = t(ACTW)$.
2) $Y_1 \subseteq Y_2 \Rightarrow i(Y_1) \supseteq i(Y_2)$. For example, for $245 \subseteq 2456$, we have $i(245) = CDW \supseteq CD = i(2456)$.
3) $X \subseteq i(t(X))$ and $Y \subseteq t(i(Y))$. For example, $AC \subseteq i(t(AC)) = i(1345) = ACW$.

Let S be a set. A function $c : \mathcal{P}(S) \mapsto \mathcal{P}(S)$ is a *closure operator* on S if, for all $X, Y \subseteq S$, c satisfies the following properties: 1) Extension: $X \subseteq c(X)$. 2) Monotonicity: if $X \subseteq Y$, then $c(X) \subseteq c(Y)$. 3) Idempotency: $c(c(X)) = c(X)$. A subset X of S is called *closed* if $c(X) = X$.

Lemma 1 *Let $X \subseteq \mathcal{I}$ and $Y \subseteq \mathcal{T}$. Let $c_{it}(X)$ denote the composition of the two mappings $i \circ t(X) = i(t(X))$. Dually, let $c_{ti}(Y) = t \circ i(Y) = t(i(Y))$. Then $c_{it} : \mathcal{P}(\mathcal{I}) \mapsto \mathcal{P}(\mathcal{I})$ and $c_{ti} : \mathcal{P}(\mathcal{T}) \mapsto \mathcal{P}(\mathcal{T})$ are both closure operators on itemsets and tidsets respectively.*

We define a *closed itemset* as an itemset X that is the same as its closure, i.e., $X = c_{it}(X)$. For example the itemset ACW is closed. A *closed tidset* is a tidset $Y = c_{ti}(Y)$. For example, the tidset 1345 is closed.

The mappings c_{it} and c_{ti}, being closure operators, satisfy the three properties of extension, monotonicity, and idempotency. We also call the application of $i \circ t$ or $t \circ i$ a *round-trip*. Figure 4 illustrates this round-trip starting with an itemset X. For example, let $X = AC$, then the extension property says that X is a subset of its closure, since $c_{it}(AC) = i(t(AC)) = i(1345) = ACW$. Since $AC \neq c_{it}(AC) = ACW$, we conclude that AC is not closed. On the other hand, the idempotency property say that once we map an itemset to the tidset that contains

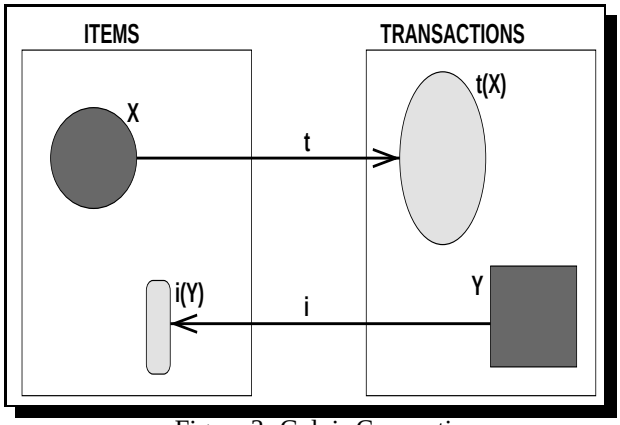

Figure 3: Galois Connection

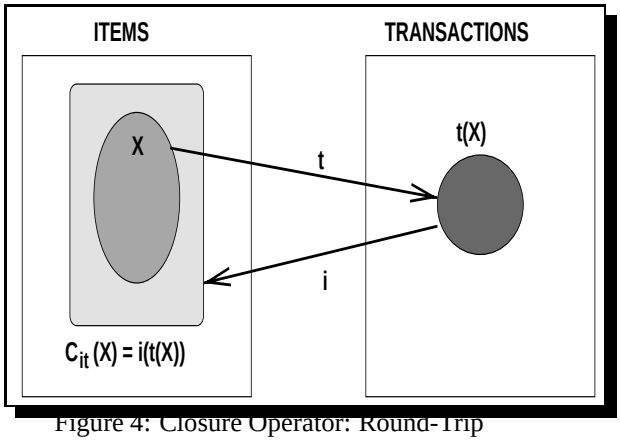

Figure 4: Closure Operator: Round-Trip

it, and then map that tidset back to the set of items common to all tids in the tidset, we obtain a closed itemset. After this no matter how many such round-trips we make we cannot extend a closed itemset. For example, after one round-trip for AC we obtain the closed itemset ACW. If we perform another round-trip on ACW, we get $c_{it}(ACW) = i(t(ACW)) = i(1345) = ACW$.

For any closed itemset X, there exists a closed tidset given by Y, with the property that $Y = t(X)$ and $X = i(Y)$ (conversely, for any closed tidset there exists a closed itemset). We can see that X is closed by the fact that $X = i(Y)$, then plugging $Y = t(X)$, we get $X = i(Y) = i(t(X)) = c_{it}(X)$, thus X is closed. Dually, Y is closed. For example, we have seen above that for the closed itemset ACW the associated closed tidset is 1345. Such a closed itemset and closed tidset pair $X \times Y$ is called a *concept*.

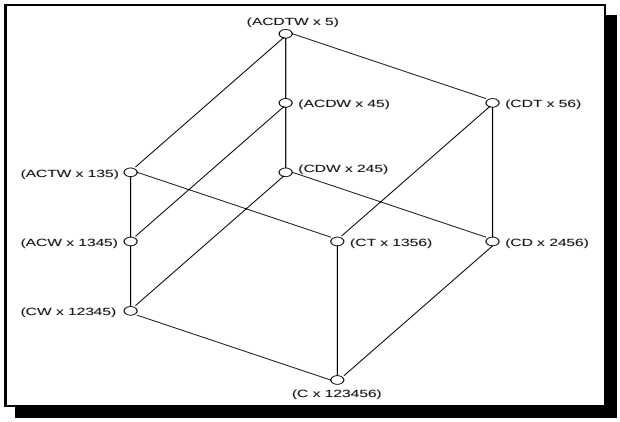

Figure 5: Galois Lattice of Concepts

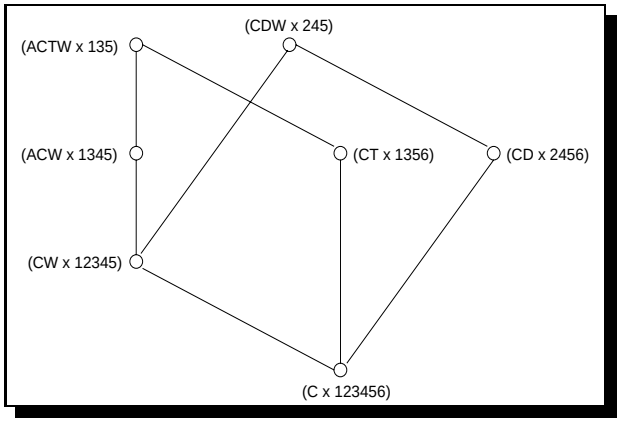

Figure 6: Frequent Concepts

A concept $X_1 \times Y_1$ is a *subconcept* of $X_2 \times Y_2$, denoted as $X_1 \times Y_1 \leq X_2 \times Y_2$, iff $X_1 \subseteq X_2$ (iff $Y_2 \subseteq Y_1$). Let $\mathcal{B}(\delta)$ denote the set of all possible concepts in the database, then the ordered set $(\mathcal{B}(\delta), \leq)$ is a complete lattice, called the *Galois* lattice. For example, Figure 5 shows the Galois lattice for our example database, which has a total of 10 concepts. The least element is the concept $C \times 123456$ and the greatest element is the concept $ACDTW \times 5$. Notice that the mappings between the closed pairs of itemsets and tidsets are anti-isomorphic, i.e., concepts with large cardinality itemsets have small tidsets, and vice versa.

3.3 Closed Frequent Itemsets vs. All Frequent Itemsets

We begin this section by defining the join and meet operation on the concept lattice (see [5] for the formal proof): The set of all concepts in the database relation δ, given by $(\mathcal{B}(\delta), \leq)$ is a (complete) lattice with join and meet given by

$$\textbf{join: } (X_1 \times Y_1) \ \vee \ (X_2 \times Y_2) = c_{it}(X_1 \cup X_2) \times (Y_1 \cap Y_2)$$

meet: $(X_1 \times Y_1) \ \wedge \ (X_2 \times Y_2) = (X_1 \cap X_2) \times c_{ti}(Y_1 \cup Y_2)$

For the join and meet of multiple concepts, we simply take the unions and joins over all of them. For example, consider the join of two concepts, $(ACDW \times 45) \vee (CDT \times 56) = c_{it}(ACDW \cup CDT) \times (45 \cap 56) = ACDTW \times 5$. On the other hand their meet is given as, $(ACDW \times 45) \wedge (CDT \times 56) = (ACDW \cap CDT) \times c_{ti}(45 \cup 56) = CD \times c_{ti}(456) = CD \times 2456$. Similarly, we can perform multiple concept joins or meets; for example, $(CT \times 1356) \vee (CD \times 2456) \vee (CDW \times 245) = c_{it}(CT \cup CD \cup CDW) \times (1356 \cap 2456 \cap 245) = c_{it}(CDTW) \times 5 = ACDTW \times 5$.

We define the support of a closed itemset X or a concept $X \times Y$ as the cardinality of the closed tidset $Y = t(X)$, i.e, $\sigma(X) = |Y| = |t(X)|$. A closed itemset or a concept is *frequent* if its support is at least *minsup*. Figure 6 shows all the frequent concepts with *minsup* = 50% (i.e., with tidset cardinality at least 3). The frequent concepts, like the frequent itemsets, form a meet-semilattice, where the meet is guaranteed to exist, while the join may not.

Theorem 1 *For any itemset X, its support is equal to the support of its closure, i.e., $\sigma(X) = \sigma(c_{it}(X))$.*

PROOF: The support of an itemset X is the number of transactions where it appears, which is exactly the cardinality of the tidset $t(X)$, i.e., $\sigma(X) = |t(X)|$. Since $\sigma(c_{it}(X)) = |t(c_{it}(X))|$, to prove the lemma, we have to show that $t(X) = t(c_{it}(X))$.

Since c_{ti} is closure operator, it satisfies the extension property, i.e., $t(X) \subseteq c_{ti}(t(X)) = t(i(t(X))) = t(c_{it}(X))$. Thus $t(X) \subseteq t(c_{it}(X))$. On the other hand since c_{it} is also a closure operator, $X \subseteq c_{it}(X)$, which in turn implies that $t(X) \supseteq t(c_{it}(X))$, due to property 1) of Galois connections. Thus $t(X) = t(c_{it}(X))$. ■

This lemma states that all frequent itemsets are uniquely determined by the frequent closed itemsets (or frequent concepts). Furthermore, the set of frequent closed itemsets is bounded above by the set of frequent itemsets, and is typically much smaller, especially for dense datasets (where there can be orders of magnitude differences). To illustrate the benefits of closed itemset mining, contrast Figure 2, showing the set of all frequent itemsets, with Figure 6, showing the set of all closed frequent itemsets (or concepts). We see that while there are only 7 closed frequent itemsets, there are 19 frequent itemsets. This example clearly illustrates the benefits of mining the closed frequent itemsets.

3.4 Rule Generation

Recall that an association rule is of the form $X_1 \xrightarrow{p} X_2$, where $X_1, X_2 \subseteq \mathcal{I}$. Its support equals $|t(X_1 \cup X_2)|$, and its confidence is given as $p = |t(X_1 \cup X_2)|/|t(X_1)|$. We are interested in finding all high support (at least *minsup*) and high confidence rules (at least *minconf*).

It is widely recognized that the set of such association rules can rapidly grow to be unwieldy. The larger the set of frequent itemsets the more the number of rules presented to the user. However, we show below that it is not necessary to mine rules from all frequent itemsets, since most of these rules turn out to be redundant. In fact, it is sufficient to consider only the rules among closed frequent itemsets (or concepts), as stated in the theorem below.

Theorem 2 *The rule $X_1 \xrightarrow{p} X_2$ is equivalent to the rule $c_{it}(X_1) \xrightarrow{q} c_{it}(X_2)$, where $q = p$.*

PROOF: It follows immediately from the fact that the support of an itemset X is equal to the support of its closure $c_{it}(X)$, i.e., $t(X) = t(c_{it}(X))$. Using this fact we can show that

$$q = \frac{|t(c_{it}(X_1) \cup c_{it}(X_2))|}{|t(c_{it}(X_1))|} = \frac{|t(c_{it}(X_1)) \cap t(c_{it}(X_2))|}{|t(X_1)|} = \frac{|t(X_1) \cap t(X_2)|}{|t(X_1)|} = \frac{|t(X_1 \cup X_2)|}{|t(X_1)|} = p \ \blacksquare$$

There are typically many (in the worst case, an exponential number of) frequent itemsets that map to the same closed frequent itemset. Let's assume that there are n itemsets, given by the set S_1, whose closure is C_1 and m itemsets, given by the set S_2, whose closure is C_2, then we say that all $n \cdot m - 1$ rules between two non-closed itemsets directed from S_1 to S_2 are *redundant*. They are all equivalent to the rule $C_1 \xrightarrow{p} C_2$. Further the $m \cdot n - 1$ rules directed from S_2 to S_1 are also redundant, and equivalent to the rule $C_2 \xrightarrow{q} C_1$. For example, looking at Figure 2 we find that the itemsets D and CD map to the closed itemset CD, and the itemsets W and CW map to the closed itemset CW. Considering rules from the former to latter set we find that the rules $D \xrightarrow{3/4} W$, $D \xrightarrow{3/4} CW$, and $CD \xrightarrow{3/4} W$ are all equivalent to the rule between closed itemsets $CD \xrightarrow{3/4} CW$. On the other hand, if we consider the rules from the latter set to the former, we find that $W \xrightarrow{3/5} D$, $W \xrightarrow{3/5} CD$, $CW \xrightarrow{3/5} D$ are all equivalent to the rule $CW \xrightarrow{5/6} CD$.

We should present to the user the most general rules (other rules are more specific; they contain one or more additional items in the antecedent or consequent) for each direction, i.e., the rules $D \xrightarrow{3/4} W$ and $W \xrightarrow{3/5} D$ (provided *minconf* $= 0.6$). Thus using the closed frequent itemsets we would generate only 2 rules instead of 8 rules normally generated between the two sets. To get an idea of the number of redundant rules mined in traditional association mining, for one dataset (mushroom), at 10% minimum support, we found 574513 frequent itemsets, out of which only 4897 were closed, a reduction of more than 100 times!

4 CHARM: Algorithm Design and Implementation

Having developed the main ideas behind closed association rule mining, 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. This separation of design and implementation aids comprehension, and allows the possibility of multiple implementations.

CHARM is unique in that it simultaneously explores both the itemset space and tidset space, unlike all previous association mining methods which only exploit the itemset space. Furthermore, CHARM avoids enumerating all possible subsets of a closed itemset when enumerating the closed frequent sets, which rules out a pure bottom-up search. This property is important in mining dense domains with long frequent itemsets, where bottom-up approaches are not practical (for example if the longest frequent itemset is l, then bottom-up search enumerates all 2^l frequent subsets).

The exploration of both the itemset and tidset space allows CHARM 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. Further, CHARM uses a two-pronged pruning strategy. It prunes candidates based not only on subset infrequency (i.e., no extensions of an infrequent itemset are tested) as do all association mining methods, but it also prunes branches based on non-closure property, i.e., any non-closed itemset is pruned. Finally, CHARM uses no internal data structures like Hash-trees [1] or Tries [3]. The fundamental operation used is an union of two itemsets and an intersection of their tidsets.

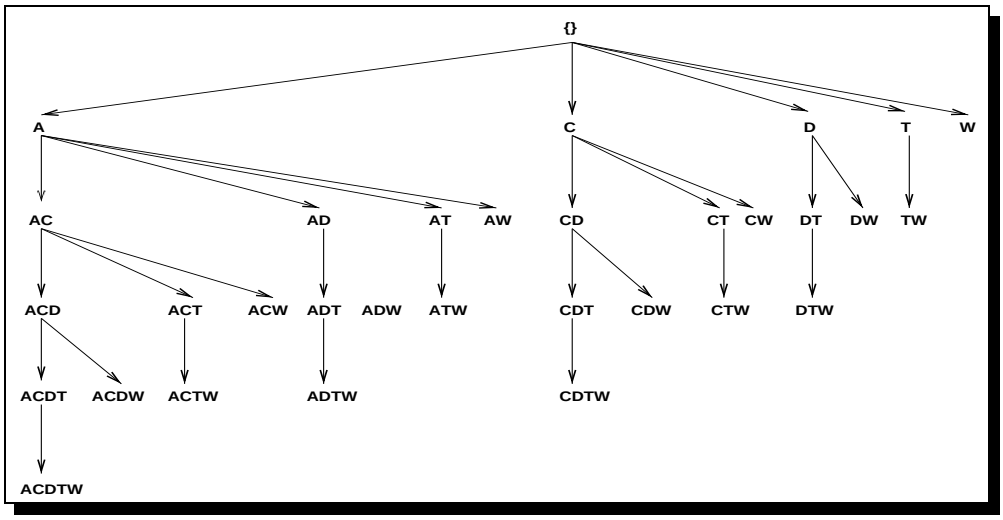

Figure 7: Complete Subset Lattice

Consider Figure 7 which shows the complete subset lattice (only the main parent link has been shown to reduce clutter) over the five items in our example database (see Figure 1). The idea in CHARM is to process each lattice node to test if its children are frequent. All infrequent, as well as non-closed branches are pruned. Notice that the children of each node are formed by combining the node by each of its siblings that come after it in the branch ordering. For example, A has to be combined with its siblings C, D, T and W to produce the children AC, AD, AT and AW.

A sibling need not be considered if it has already been pruned because of infrequency or non-closure. While a lexical ordering of branches is shown in the figure, we will see later how a different branch ordering (based on

support) can improve the performance of CHARM (a similar observation was made in MaxMiner [2]). While many search schemes are possible (e.g., breadth-first, depth-first, best-first, or other hybrid search), CHARM performs a depth-first search of the subset lattice.

4.1 CHARM: Algorithm Design

In this section we assume that for any itemset X, we have access to its tidset $t(X)$, and for any tidset Y we have access to its itemset $i(Y)$. How to practically generate $t(X)$ or $i(Y)$ will be discussed in the implementation section.

CHARM actually enumerates all the frequent concepts in the input database. Recall that a concept is given as a pair $X \times Y$, where $X = i(Y)$ is a closed itemset, and $Y = t(X)$ is a closed tidset. We can start the search for concepts over the tidset space or the itemset space. However, typically the number of items is a lot smaller than the number of transactions, and since we are ultimately interested in the closed itemsets, we start the search with the single items, and their associated tidsets.

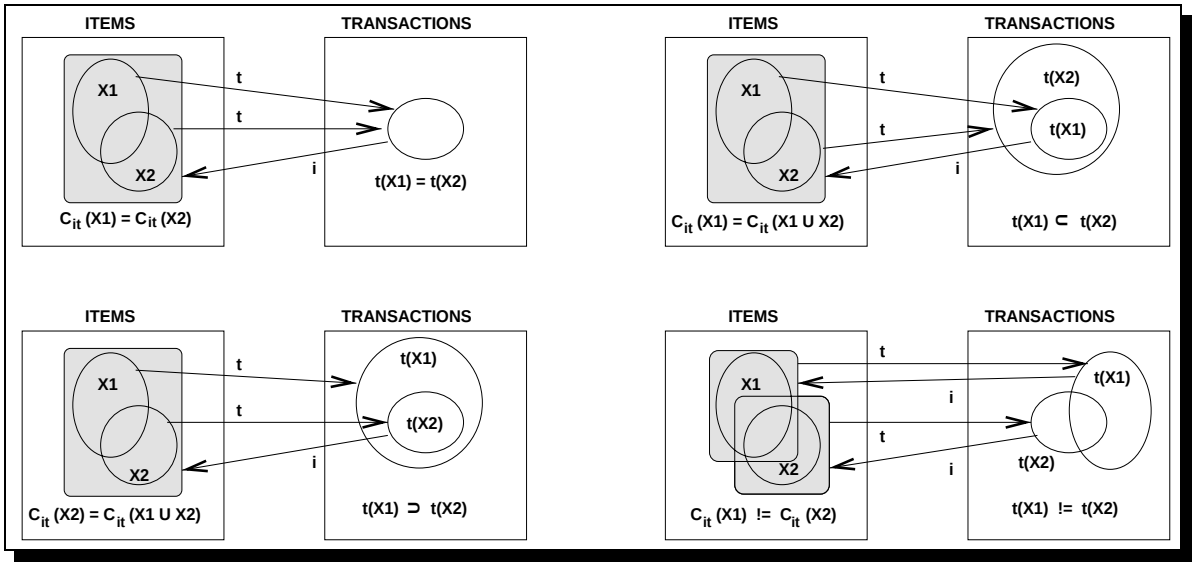

Figure 8: Basic Properties of Itemsets and Tidsets

4.1.1 Basic Properties of Itemset-Tidset Pairs

Let $f : \mathcal{P}(\mathcal{I}) \mapsto N$ be a one-to-one mapping from itemsets to integers. For any two itemsets X_1 and X_2, we say $X_1 \leq X_2$ iff $f(X_1) \leq f(X_2)$. f defines a total order over the set of all itemsets. For example, if f denotes the lexicographic ordering, then itemset $AC < AD$. As another example, if f sorts itemsets in increasing order of their support, then $AD < AC$ if support of AD is less than the support of AC.

Let's assume that we are processing the branch $X_1 \times t(X_1)$, and we want to combine it with its sibling $X_2 \times t(X_2)$. That is $X_1 \leq X_2$ (under a suitable total order f). The main computation in CHARM relies on the following properties.

1. If $t(X_1) = t(X_2)$, then $t(X_1 \cup X_2) = t(X_1) \cap t(X_2) = t(X_1) = t(X_2)$. Thus we can simply replace every occurrence of X_1 with $X_1 \cup X_2$, and remove X_2 from further consideration, since its closure is identical to the closure of $X_1 \cup X_2$. In other words, we treat $X_1 \cup X_2$ as a composite itemset.

2. If $t(X_1) \subset t(X_2)$, then $t(X_1 \cup X_2) = t(X_1) \cap t(X_2) = t(X_1) \neq t(X_2)$. Here we can replace every occurrence of X_1 with $X_1 \cup X_2$, since if X_1 occurs in any transaction, then X_2 always occurs there too. But since $t(X_1) \neq t(X_2)$ we cannot remove X_2 from consideration; it generates a different closure.

3. If $t(X_1) \supset t(X_2)$, then $t(X_1 \cup X_2) = t(X_1) \cap t(X_2) = t(X_2) \neq t(X_1)$. In this we replace every occurrence of X_2 with $X_1 \cup X_2$, since wherever X_2 occurs X_1 always occurs. X_1 however, produces a different closure, and it must be retained.

4. If $t(X_1) \neq t(X_2)$, then $t(X_1 \cup X_2) = t(X_1) \cap t(X_2) \neq t(X_2) \neq t(X_1)$. In this case, nothing can be eliminated; both X_1 and X_2 lead to different closures.

Figure 8 pictorially depicts the four cases. We see that only closed tidsets are retained after we combine two itemset-tidset pairs. For example, if the two tidsets are equal, one of them is pruned (Property 1). If one tidset is a subset of another, then the resulting tidset is equal to the smaller tidset from the parent and we eliminate that parent (Properties 2 and 3). Finally if the tidsets are unequal, then those two and their intersection are all closed.

Example Before formally presenting the algorithm, we show how the four basic properties of itemset-tidset pairs are exploited in CHARM to mine the closed frequent itemsets.

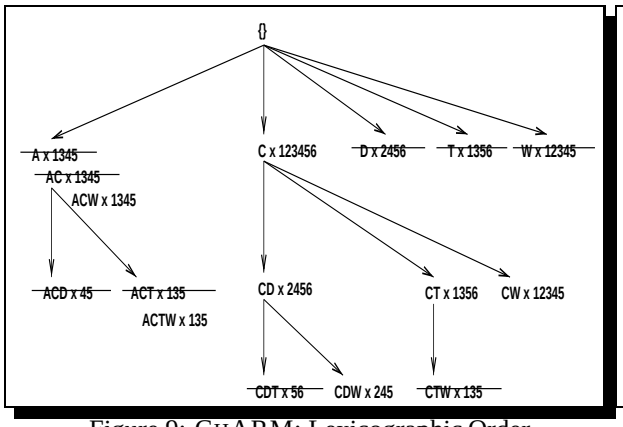

Figure 9: CHARM: Lexicographic Order

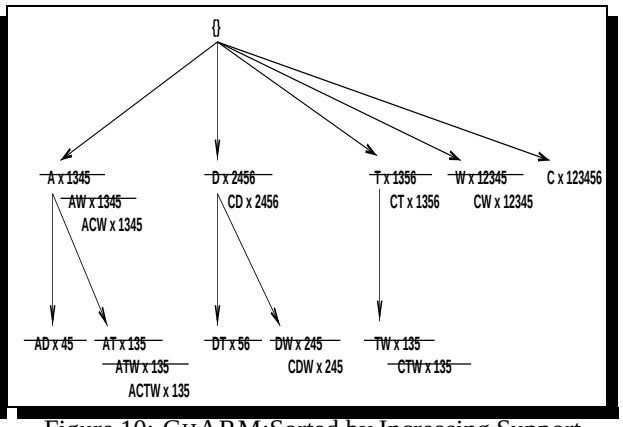

Figure 10: CHARM:Sorted by Increasing Support

Consider Figure 9. Initially we have five branches, corresponding to the five items and their tidsets from our example database (recall that we used *minsup* = 3). To generate the children of item A (or the pair $A \times 1345$) we need to combine it with all siblings that come after it. When we combine two pairs $X_1 \times t(X_1)$ and $X_2 \times t(X_2)$, the resulting pair is given as $(X_1 \cup X_2) \times (t(X_1) \cap t(X_2))$. In other words we need to perform the intersection of corresponding tidsets whenever we combine two or more itemsets.

When we try to extend A with C, we find that property 2 is true, i.e., $t(A) = 1345 \subseteq 123456 = t(C)$. We can thus remove A and replace it with AC. Combining A with D produces an infrequent set ACD, which is pruned. Combination with T produces the pair $ACT \times 135$; property 4 holds here, so nothing can be pruned. When we try to combine A with W we find that $t(A) \subseteq t(W)$. According to property 2, we replace all unpruned occurrences of A with AW. Thus AC becomes ACW and ACT becomes $ACTW$. At this point there is nothing further to be processed from the A branch of the root.

We now start processing the C branch. When we combine C with D we observe that property 3 holds, i.e., $t(C) \supset t(D)$. This means that wherever D occurs C always occurs. Thus D can be removed from further consideration, and the entire D branch is pruned; the child CD replaces D. Exactly the same scenario occurs with T and W. Both the branches are pruned and are replaced by CT and CW as children of C. Continuing in a depth-first manner, we next process the node CD. Combining it with CT produces an infrequent itemset CDT, which is pruned. Combination with CW produces CDW and since property 4 holds, nothing can be removed. Similarly the combination of CT and CW produces CTW. At this point all branches have been processed.

Finally, we remove $CTW \times 135$ since it is contained in $ACTW \times 135$. As we can see, in just 10 steps we have identified all 7 closed frequent itemsets.

4.1.2 CHARM: Pseudo-Code Description

Having illustrated the workings of CHARM on our example database, we now present the pseudo-code for the algorithm itself.

The algorithm starts by initializing the set of nodes to be examined to the frequent single items and their tidsets in Line 1. The main computation is performed in CHARM-EXTEND which returns the set of closed frequent itemsets $\mathcal{C}$.

CHARM-EXTEND is responsible for testing each branch for viability. It extracts each itemset-tidset pair in the current node set $Nodes$ ($X_i \times t(X_i)$, Line 3), and combines it with the other pairs that come after it ($X_j \times t(X_j)$, Line 5) according to the total order f (we have already seen an example of lexical ordering in Figure 9; we will look at support based ordering below). The combination of the two itemset-tidset pairs is computed in Line 6. The routine CHARM-PROPERTY tests the resulting set for required support and also applies the four properties discussed above. Note that this routine may modify the current node set by deleting itemset-tidset pairs that are already contained in other pairs. It also inserts the newly generated children frequent pairs in the set of new nodes $NewN$. If this set is non-empty we recursively process it in depth-first manner (Line 8). We then insert the possibly extended itemset $\mathbf{X}$, of X_i, in the set of closed itemsets, since it cannot be processed further; at this stage any closed itemset containing X_i has already been generated. We then return to Line 3 to process the next (unpruned) branch.

The routine CHARM-PROPERTY simply tests if a new pair is frequent, discarding it if it is not. It then tests each of the four basic properties of itemset-tidset pairs, extending existing itemsets, removing some subsumed branches from the current set of nodes, or inserting new pairs in the node set for the next (depth-first) step.

```
CHARM (δ ⊆ I × T, minsup):
1.  Nodes = {I_j × t(I_j) : I_j ∈ I ∧ |t(I_j)| ≥ minsup}
2.  CHARM-EXTEND (Nodes, C)

CHARM-EXTEND (Nodes, C):
3.  for each X_i × t(X_i) in Nodes
4.      NewN = ∅ and X = X_i
5.      for each X_j × t(X_j) in Nodes, with f(j) > f(i)
6.          X = X ∪ X_j and Y = t(X_i) ∩ t(X_j)
7.          CHARM-PROPERTY(Nodes, NewN)
8.      if NewN ≠ ∅ then CHARM-EXTEND (NewN)
9.      C = C ∪ X //if X is not subsumed

CHARM-PROPERTY (Nodes, NewN):
10. if (|Y| ≥ minsup) then
11.     if t(X_i) = t(X_j) then //Property 1
12.         Remove X_j from Nodes
13.         Replace all X_i with X
14.     else if t(X_i) ⊂ t(X_j) then //Property 2
15.         Replace all X_i with X
16.     else if t(X_i) ⊃ t(X_j) then //Property 3
17.         Remove X_j from Nodes
18.         Add X × Y to NewN
19.     else if t(X_i) ≠ t(X_j) then //Property 4
20.         Add X × Y to NewN
```

Figure 11: The CHARM Algorithm

4.1.3 Branch Reordering

We purposely let the itemset-tidset pair ordering function in Line 5 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 non-closure based pruning of itemsets. A quick look at Properties 1 and 2 tells us that these two situations are preferred over the other two cases. 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 nodes! Thus the more the occurrence of case 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. Note that the reordering is applied for each new node set, starting with the initial branches.

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

Example Figure 10 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. We initialize $Nodes = \{A \times 1345, D \times 2456, T \times 1356, W \times 12345, C \times 123456\}$ in Line 1.

At Line 3 we first process the branch $A \times 1345$ (we set $\mathbf{X} = A$ in Line 4); it will be combined with the remaining siblings in Line 5. AD is not frequent and is pruned. We next look at A and T; since $t(A) \neq t(T)$, we simply insert AT in $NewN$. We next find that $t(A) \subset t(W)$. Thus we replace all occurrences of A with AW (thus $\mathbf{X} = AW$), which means that we also change AT in $NewN$ to ATW. Looking at A and C, we find that $t(A) \subset t(C)$. Thus

AW becomes ACW ($\mathbf{X} = ACW$), and ATW in $NewN$ becomes $ACTW$. At this point CHARM-EXTEND is invoked with the non-empty $NewN$ (Line 8). But since there is only one element, we immediately exit after adding $ACTW \times 135$ to the set of closed frequent itemsets $\mathcal{C}$ (Line 9).

When we return, the A branch has been completely processed, and we add $\mathbf{X} = ACW$ to $\mathcal{C}$. The other branches are examined in turn, and the final $\mathcal{C}$ is produced as shown in Figure 10. One final note; the pair $CTW \times 135$ produced from the T branch is not closed, since it is subsumed by $ACTW \times 135$, and it is eliminated in Line 9.

4.2 CHARM: Implementation Details

We now describe the implementation details of CHARM and how it departs from the pseudo-code in some instances for performance reasons.

Data Format Given that we are manipulating itemset-tidset pairs, and that the fundamental operation is that of intersecting two tidsets, CHARM uses a *vertical* data format, where we maintain a disk-based list for each item, listing the tids where that item occurs. In other words, the data is organized so that we have available on disk the tidset for each item. In contrast most of the current association algorithms [1, 2, 3] assume a *horizontal* database layout, consisting of a list of transactions, where each transaction has an identifier followed by a list of items in that transaction.

The vertical format has been shown to be successful for association mining. It has been used in Partition [16], in (Max)Eclat and (Max)Clique [19], and shown to lead to very good performance. In fact, the *Vertical* algorithm [15] was shown to be the best approach (better than horizontal) when tightly integrating association mining with database systems. The benefits of using the vertical format have further been demonstrated in Monet [12], a new high-performance database system for query-intensive applications like OLAP and data mining.

Intersections and Subset Testing Given the availability of vertical tidsets for each itemset, the computation of the tidset intersection for a new combination is straightforward. All it takes is a linear scan through the two tidsets, storing matching tids in a new tidset. For example, we have $t(A) = 1345$ and $t(D) = 2456$. Then $t(AD) = 1345 \cap 2456 = 45$.

The main question is how to efficiently compute the subset information required while applying the four properties. At first this might appear like an expensive operation, but in fact in the vertical format, it comes for free.

When intersecting two tidsets we keep track of the number of mismatches in both the lists, i.e., the cases when a tid occurs in one list but not in the other. Let $m(X_1)$ and $m(X_2)$ denote the number of mismatches in the tidsets for itemsets X_1 and X_2. There are four cases to consider:

$$\begin{array}{ll} m(X_1) = 0 \text{ and } m(X_2) = 0, & \text{then } t(X_1) = t(X_2) \text{ — Property 1} \\ m(X_1) = 0 \text{ and } m(X_2) \neq 0, & \text{then } t(X_1) \subset t(X_2) \text{ — Property 2} \\ m(X_1) \neq 0 \text{ and } m(X_2) = 0, & \text{then } t(X_1) \supset t(X_2) \text{ — Property 3} \\ m(X_1) \neq 0 \text{ and } m(X_2) \neq 0, & \text{then } t(X_1) \neq t(X_2) \text{ — Property 4} \end{array}$$

For $t(A)$ and $t(D)$ from above, $m(A) = 2$ and $m(D) = 2$, and as we can see, $t(A) \neq t(D)$. Next consider $t(A) = 1345$ and $t(W) = 12345$. We find $m(A) = 0$, but $m(W) = 1$, which shows that $t(A) \subset t(W)$. Thus CHARM performs support, subset, equality, and inequality testing simultaneously while computing the intersection itself.

Eliminating Non-Closed Itemsets Here we describe a fast method to avoid adding non-closed itemsets to the set of closed frequent itemsets $\mathcal{C}$ in Line 9. If we are adding a set $\mathbf{X}$, we have to make sure that there doesn't exist a set $C \in \mathcal{C}$ such that $\mathbf{X} \subset C$ and both have the same support (MaxMiner [2] faces a similar problem while eliminating non-maximal itemsets).

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. The 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 subsets may have the same support.

CHARM uses the sum of the tids in the tidset as the hash function, i.e., $h(\mathbf{X}) = \sum_{T \in t(\mathbf{X})} T$. This reduces the chances of unrelated itemsets being in the same cell. Each hash table cell is a linked list sorted by support as primary

key and the itemset as the secondary key (i.e., lexical). Before adding $\mathbf{X}$ to $\mathcal{C}$, we hash to the cell, and check if $\mathbf{X}$ is a subset of only those itemsets with the same support as $\mathbf{X}$. We found experimentally that this approach adds only a few seconds of additional processing time to the total execution time.

Optimized Initialization There is only one significant departure from the pseudo-code in Figure 11. Note that if we initialize the $Nodes$ set in Line 1 with all frequent items, and invoke CHARM-EXTEND then, in the worst case, we might perform $n(n-1)/2$ tidset intersections, where n is the number of frequent items. If l is the average tidset size in bytes, the amount of data read is $l \cdot n \cdot (n-1)/2$ bytes. Contrast this with the horizontal approach that reads only $l \cdot n$ bytes.

It is well known that many itemsets of length 2 turn out to be infrequent, thus it is clearly wasteful to perform $O(n^2)$ intersection. 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 5, so that we combine two items I_i and I_j only if $I_i \cup I_j$ is known to be frequent. The number of intersections performed after this check is equal to the number of frequent pairs, which is in practice closer to $O(n)$ rather than $O(n^2)$. Further this check only has to be done initially only for single items, and not in later stages.

We now describe how we compute the frequent itemsets of length 2 using the vertical format. As noted above we clearly cannot perform all intersections between pairs of frequent items.

The solution is to perform a vertical to horizontal transformation on-the-fly. For each item I, we scan its tidset into memory. We insert item I in an array indexed by tid for each $T \in t(I)$. For example, consider the tidset for item A, given as $t(A) = 1345$. We read the first tid $T = 1$, and then insert A in the array at index 1. We also insert A at indices $3, 4$ and 5. We repeat this process for all other items and their tidsets. Figure 12 shows how the inversion process works after the addition of each item and the complete horizontal database recovered from the vertical tidsets for each item. Given the recovered horizontal database it is straightforward to update the count of pairs of items using an upper triangular 2D array.

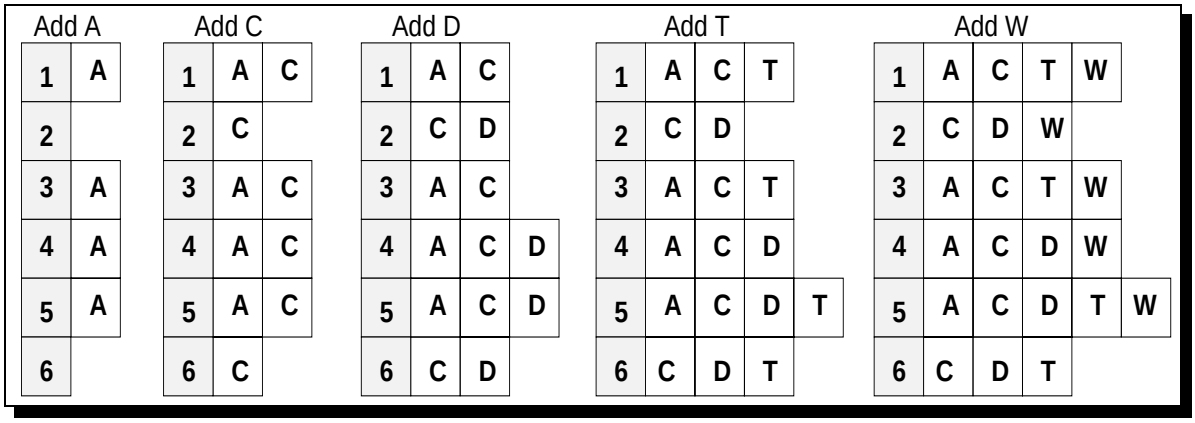

Figure 12: Vertical-to-Horizontal Database Recovery

Memory Management For initialization CHARM scans the database once to compute the frequent pairs of items (note that finding the frequent items is virtually free in the vertical format; we can calculate the support directly from an index array that stores the tidset offsets for each item. If this index is not available, computing the frequent items will take an additional scan). Then, while processing each initial branch in the search lattice it needs to scan single item tidsets from disk for each unpruned sibling. CHARM is fully scalable for large-scale database mining. It implements appropriate memory management in all phases as described next.

For example, while recovering the horizontal database, the entire database will clearly not fit in memory. CHARM handles this by only recovering a block of transactions at one time that will fit in memory. Support of item pairs is updated by incrementally processing each recovered block. Note that regardless of the number of blocks, this process requires exactly one database scan over the vertical format (imagine k pointers for each of k tidsets; the pointer only moves forward if the tid is points to belongs to the current block).

When the number of closed itemsets itself becomes very large, we cannot hope to keep the set of all closed itemsets $\mathcal{C}$ in memory. In this case, the elimination of some non-closed itemsets is done off-line in a post-processing step. Instead of inserting $\mathbf{X}$ in $\mathcal{C}$ in Line 9, we simply write it to disk along with its support and hash value. In the post-processing step, we read all close itemsets and apply the same hash table searching approach described above to eliminate non-closed itemsets.

Since CHARM processes each branch in the search in a depth-first fashion, its memory requirements are not substantial. It has to retain all the itemset-tidsets pairs on the levels of the current left-most branches in the search space. Consider 7 for example. Initially is has to retain the tidsets for $\{AC, AD, AT, AW\}$, $\{ACD, ACT, ACW\}$, $\{ACDT, ACDW\}$, and $\{ACDTW\}$. Once AC has been processed, the memory requirement shrinks to $\{AD, AT,$-$AW\}$, $\{ADT, ADT\}$, and $\{ADTW\}$. In any case this is the worst possible situation. In practice the applications of subset infrequency and non-closure properties 1, 2, and 3, prune many branches in the search lattice.

For cases where even the memory requirement of depth-first search exceed available memory, it is straightforward to modify CHARM to write temporary tidsets to disk. For example, while processing the AC branch, we might have to write out the tidsets for $\{AD, AT, AW\}$ to disk. Another option is to simply re-compute the intersections if writing temporary results is too expensive.

4.3 Correctness and Efficiency

Theorem 3 (correctness) *The* CHARM *algorithm enumerates all closed frequent itemsets.*

PROOF: CHARM correctly identifies all and only the closed frequent itemsets, since its search is based on a complete subset lattice search. The only branches that are pruned as those that either do not have sufficient support, or those that result in non-closure based on the properties of itemset-tidset pairs as outlined at the beginning of this section. Finally CHARM eliminates the few cases of non-closed itemsets that might be generated by performing subsumption checking before inserting anything in the set of all closed frequent itemsets $\mathcal{C}$. ■

Theorem 4 (computational cost) *The running time of* CHARM *is $O(l \cdot |\mathcal{C}|)$, where l is the average tidset length, and $\mathcal{C}$ is the set of all closed frequent itemsets.*

PROOF: Note that starting with the single items and their associated tidsets, as we process a branch the following cases might occur. Let X_c denote the current branch and X_s the sibling we are trying to combine it with. We prune the X_s branch if $t(X_c) = t(X_s)$ (Property 1). We extend X_c to become $X_c \cup X_s$ if $t(X_c) \subset t(X_s)$ (Property 2). We prune X_s if $t(X_c) \supset t(X_s)$, and we extend X_s to become $t(X_c) \subset t(X_s)$ (Property 3). Finally a new node is only generated if we get a new possibly closed set due to properties 3 and 4. Also note that each new node in fact represents a closed tidset, and thus indirectly represents a closed itemset, since there exists a unique closed itemset for each closed tidset. Thus CHARM performs on the order of $O(|\mathcal{C}|)$ intersections (we confirm this via experiments in Section 6; the only extra intersections performed are due to case where CHARM may produce non-closed itemsets like $CTW \times 135$, which are eliminated in Line 9). If each tidset is on average of length l, an intersection costs at most $2 \cdot l$. The total running time of CHARM is thus $2 \cdot l \cdot |\mathcal{C}|$ or $O(l \cdot |\mathcal{C}|)$. ■

Theorem 5 (I/O cost) *The number of database scans made by* CHARM *is given as $O(\frac{|\mathcal{C}|}{\alpha \cdot |\mathcal{I}|})$, where $\mathcal{C}$ is the set of all closed frequent itemsets, $\mathcal{I}$ is the set of items, and α is the fraction of database that fits in memory.*

PROOF: The number of database scans required is given as the total memory consumption of the algorithm divided by the fraction of database that will fit in memory. Since CHARM computes on the order of $O(|\mathcal{C}|)$ intersections, the total memory requirement of CHARM is $O(l \cdot |\mathcal{C}|)$, where l is the average length of a tidset. Note that as we perform intersections the size of longer itemsets' tidsets shrinks rapidly, but we ignore such effects in our analysis (it is thus a pessimistic bound). The total database size is $l \cdot |\mathcal{I}|$, and the fraction that fits in memory is given as $\alpha \cdot l \cdot |\mathcal{I}|$. The number of data scans is then given as $(l \cdot |\mathcal{C}|)/(\alpha \cdot l \cdot |\mathcal{I}|) = |\mathcal{C}|/(\alpha \cdot |\mathcal{I}|)$. ■

Note that in the worst case $|\mathcal{C}|$ can be exponential in $|\mathcal{I}|$, but this is rarely the case in practice. We will show in the experiments section that CHARM makes very few database scans when compared to the longest closed frequent itemset found.

5 Related Work

A number of algorithms for mining frequent itemsets [1, 2, 3, 9, 10, 13, 16, 19] have been proposed in the past. Apriori [1] was the first efficient and scalable method for mining associations. It starts by counting frequent items, and during each subsequent pass it extends the current set of frequent itemsets by one more item, until no more frequent itemsets are found. Since it uses a pure bottom-up search over the subset lattice (see Figure 7), it generates all 2^l subsets of a frequent itemset of length l. Other methods including DHP [13], Partition [16], AS-CPA [10], and DIC [3], propose enhancements over Apriori in terms of the number of candidates counted or the number of data scans. But they still have to generate all subsets of a frequent itemset. This is simply not feasible (except for very high support) for the kinds of dense datasets we examine in this paper. We use Apriori as a representative of this class of methods in our experiments.

Methods for finding the maximal frequent itemsets include *All-MFS* [8], which is a randomized algorithm, and as such not guaranteed to be complete. *Pincer-Search* [9] not only constructs the candidates in a bottom-up manner like *Apriori*, but also starts a top-down search at the same time. Our previous algorithms (Max)Eclat and Max(Clique) [19, 17] range from those that generate all frequent itemsets to those that generate a few long frequent itemsets and other subsets. *MaxMiner* [2] is another algorithm for finding the maximal elements. It uses novel superset frequency pruning and support lower-bounding techniques to quickly narrow the search space. Since these methods mine only the maximal frequent itemsets, they cannot be used to generate all possible association rules, which requires the support of all subsets in the traditional approach. If we try to compute the support of all subsets of the maximal frequent itemsets, we again run into the problem of generating all 2^l subsets for an itemset of length l. For dense datasets this is impractical. Using MaxMiner as a representative of this class of algorithms we show that modifying it to compute closed itemsets renders it infeasible for all except very high supports.

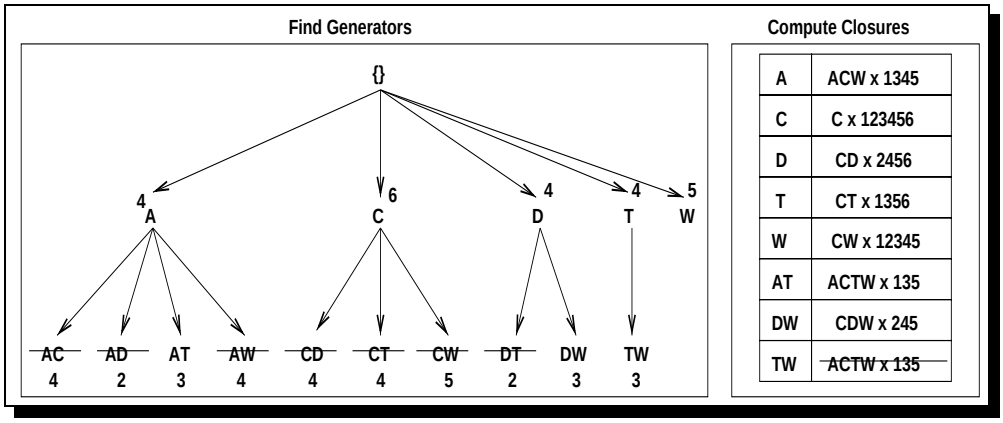

Figure 13: AClose Algorithm: Example

AClose [14] is an Apriori-like algorithm that directly mines closed frequent itemsets. There are two main steps in AClose. The first is to use a bottom-up search to identify *generators*, the smallest frequent itemsets that determines a closed itemset via the closure operator c_{it}. For example, in our example database, both $c_{it}(A) = ACW$ and $c_{it}(AC) = ACW$, but only A is a generator for ACW. All generators are found using a simple modification of Apriori. Each time a new candidate set is generated, AClose computes their support, pruning all infrequent ones. For the remaining sets, it compares the support of each frequent itemset with each of 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. This process is repeated until no more generators can be produced.

The second step in AClose 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, i.e., the closure of an itemset X is given as $c_{it}(X) = \bigcap_{X \subseteq i(T)} i(T)$, where T is a tid. 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.

Figure 13 shows the working of AClose on our example database. After generating candidate pairs of items, it is determined that AD and DT are not frequent, so they are pruned. The remaining frequent pairs are pruned if their

support matches the support of any of their subsets. AC, AW are pruned, since their support is equal to the support of A. CD is pruned because of D, CT because of T, and CW because of W. After this pruning, we find that no more candidates can be generated, marking the end of the first step. In the second step, AClose computes the closure of all unpruned itemsets. Finally some duplicate closures are removed (e.g., both AT and TW produce the same closure). We will show that while AClose is much better than Apriori, it is uncompetitive with CHARM.

A number of previous algorithms have been proposed for generating the Galois lattice of concepts [5, 6]. These algorithms will have to be adapted to enumerate only the frequent concepts. Further, they have only been studied on very small datasets. Finally the problem of generating a basis (a minimal non-redundant rule set) for association rules was discussed in [18] (but no algorithms were given), which in turn is based on the theory developed in [7, 5, 11].

6 Experimental Evaluation

We chose several real and synthetic datasets for testing the performance of CHARM. The real datasets are the same as those used in MaxMiner [2]. 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. These datasets are publicly available from IBM Almaden (www.almaden.ibm.com/cs/quest/demos.html).

We also chose a few synthetic datasets (also available from IBM Almaden), 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 sparse when compared to the real sets, but we modified the generator to produce longer frequent itemsets.

Database	# Items	Avg. Record Length	# Records	Scaleup DB Size
chess	76	37	3,196	31,960
connect	130	43	67,557	675,570
mushroom	120	23	8,124	81,240
pumsb*	7117	50	49,046	490,460
pumsb	7117	74	49,046	490,460
T20I12D100K	1000	20	100,000	1,600,000
T30I8D100K	1000	30	100,000	1,600,000
T40I8D400K	1000	40	100,000	1,600,000

Table 1: Database Characteristics

Table 1 also 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. It also shows the number of records used for the scaleup experiments below. As one can see the average transaction size for these databases is much longer than conventionally used in previous literature.

All experiments described below were performed on a 400MHz Pentium PC with 256MB of memory, running RedHat Linux 6.0. Algorithms were coded in C++.

6.1 Effect of Branch Ordering

Figure 14 shows the effect on running time if we use various kinds of branch orderings in CHARM. We compare three ordering methods — lexicographical order, increasing by support, and decreasing by support. We observe that decreasing order is the worst. On the other hand processing branch itemsets in increasing order is the best; it is about a factor of 1.5 times better than lexicographic order and about 2 times better than decreasing order. Similar results were obtained for synthetic datasets. All results for CHARM reported below use the increasing branch ordering, since it is the best.

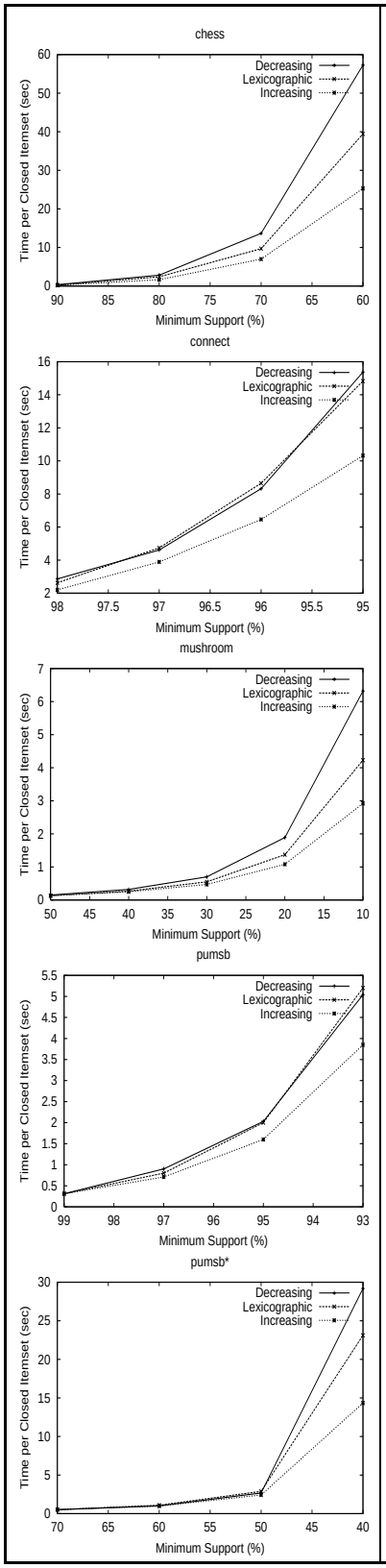

Figure 14: Branch Ordering

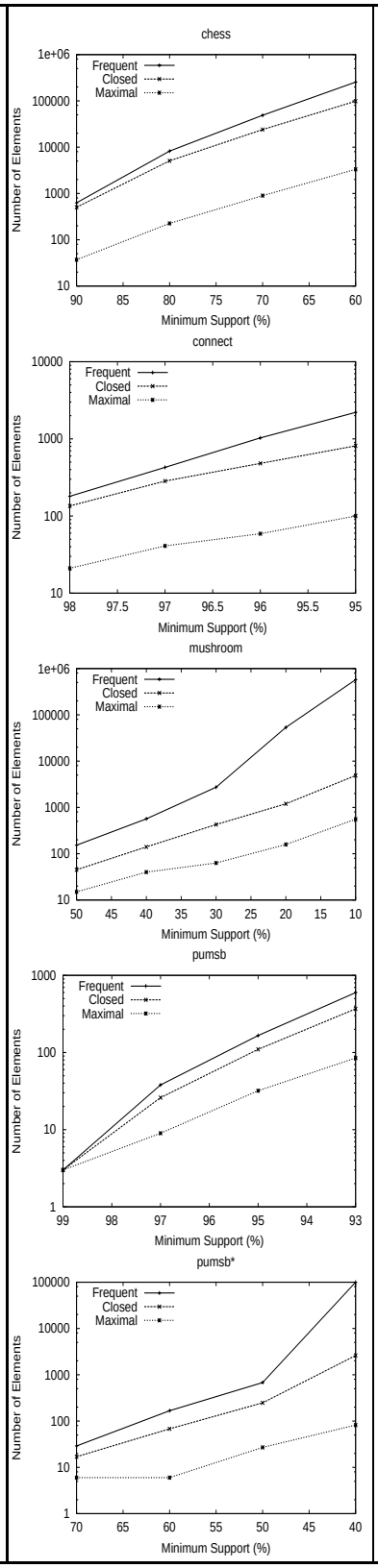

Figure 15: Set Cardinality

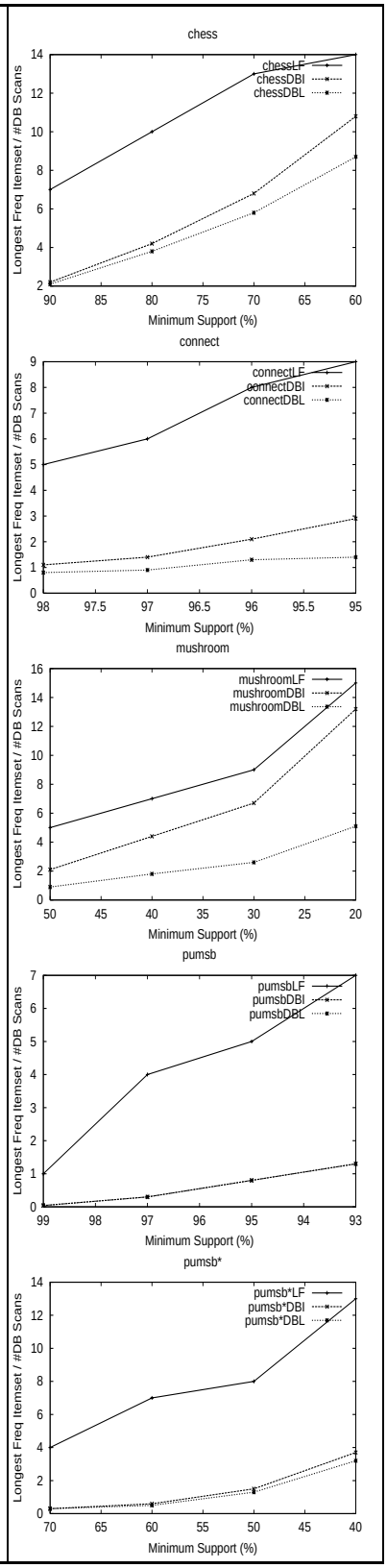

Figure 16: Longest Frequent Itemset (LF) vs. Database Scans (DBI=Increasing, DBL=Lexical Order)

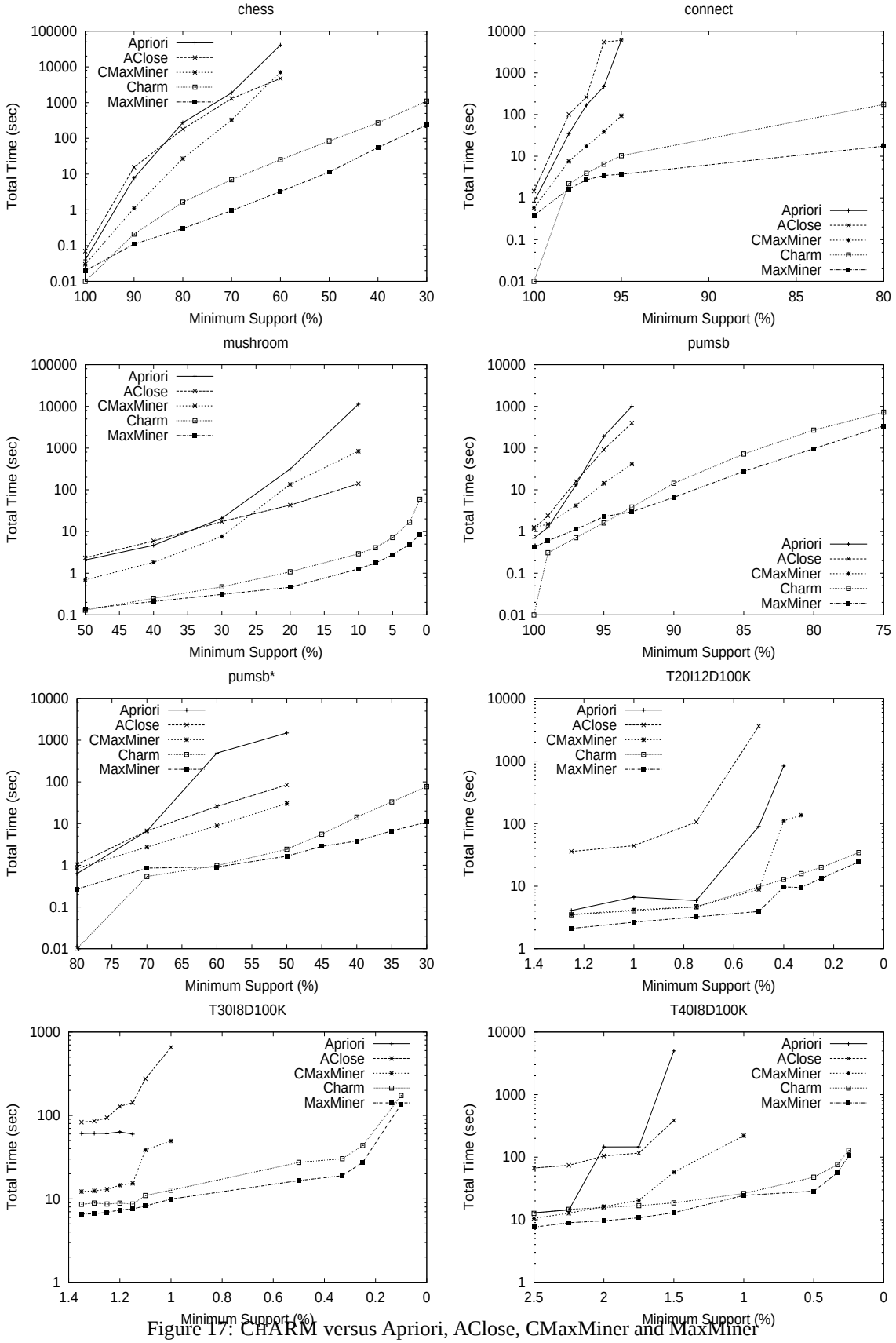

Figure 17: CHARM versus Apriori, AClose, CMaxMiner and MaxMiner

6.2 Number of Frequent, Closed, and Maximal Itemsets

Figure 15 shows the total number of frequent, closed and maximal itemsets found for various support values. It should be noted that the maximal frequent itemsets are a subset of the closed frequent itemsets (the maximal frequent itemsets must be closed, since by definition they cannot be extended by another item to yield a frequent itemset). The closed frequent itemsets are, of course, a subset of all frequent itemsets. Depending on the support value used the set of maximal itemsets is about an order of magnitude smaller than the set of closed itemsets, which in turn is an order of magnitude smaller than the set of all frequent itemsets. Even for very low support values we find that the difference between maximal and closed remains around a factor of 10. However the gap between closed and all frequent itemsets grows more rapidly. For example, for mushroom at 10% support, the gap was a factor of 100; there are 558 maximal, 4897 closed and 574513 frequent itemsets.

6.3 CHARM versus MaxMiner, AClose, and Apriori

Here we compare the performance of CHARM against previous algorithms. MaxMiner only mines maximal frequent itemsets, thus we augmented it by adding a post-processing routine that uses the maximal frequent itemsets to generate all closed frequent itemsets. In essence we generate all subsets of the maximal itemsets, eliminating an itemset if its support equals any of its subsets. The augmented algorithm is called CMaxMiner. The AClose method is the only extant method that directly mines closed frequent itemsets. Finally Apriori mines only the frequent itemsets. It would require a post-processing step to compute the closed itemsets, but we do *not* add this cost to its running time.

Figure 17 shows how CHARM compares to the previous methods on all the real and synthetic databases. We find that Apriori cannot be run except for very high values of support. Even in these cases CHARM is 2 or 3 orders of magnitude better. Generating all subsets of frequent itemsets clearly takes too much time.

AClose can perform an order of magnitude better than Apriori for low support values, but for high support values it can in fact be worse than Apriori. This is because for high support the number of frequent itemsets is not too much, and the closure computing step of AClose dominates computation time. Like Apriori, AClose couldn't be run for very low values of support. The generator finding step finds too many generators to be kept in memory.

CMaxMiner, the augmented version of MaxMiner, suffers a similar fate. Generating all subsets and testing them for closure is not a feasible strategy. CMaxMiner cannot be run for low supports, and for the cases where it can be run, it is 1 to 2 orders of magnitude slower than CHARM.

Only MaxMiner was able to run for all the values of support that CHARM can handle. Except for high support values, where CHARM is better, MaxMiner can be up to an order of magnitude faster than CHARM, and is typically a factor of 5 or 6 times better. The difference is attributable to the fact that the set of maximal frequent itemsets is typically an order of magnitude smaller than the set of closed frequent itemsets. But it should be noted that, since MaxMiner only mines maximal itemsets, it cannot be used to produce association rules. In fact, any attempt to calculate subset frequency adds a lot of overhead, as we saw in the case of CMaxMiner.

These experiments demonstrate that CHARM is extremely effective in efficiently mining all the closed frequent itemsets, and is able to gracefully handle very low support values, even in dense datasets.

6.4 Scaling Properties of CHARM

Figure 18 shows the time taken by CHARM per closed frequent itemset found. The support values are the same as the ones used while comparing CHARM with other methods above. As we lower the support more closed itemsets are found, but the time spent per element decreases, indicating that the efficiency of CHARM increases with decreasing support.

Figure 19 shows the number of tidset intersections performed per closed frequent itemset generated. The ideal case in the graph corresponds to the case where we perform exactly the same number of intersections as there are closed frequent itemsets, i.e., a ratio of one. We find that for both connect and chess the number of intersections performed by CHARM are close to ideal. CHARM is within a factor of 1.06 (for chess) to 2.6 (for mushroom) times the ideal. This confirms the computational efficiency claims we made before. CHARM indeed performs $O(|\mathcal{C}|)$ intersections.

Figure 16 shows the number of database scans made by CHARM compared to the length of the longest closed frequent itemset found for the real datasets. The number of database scans for CHARM was calculated by taking the sum of the lengths of all tidsets scanned from disks, and then dividing the sum by the tidset lengths for all items in the database. The number reported is pessimistic in the sense that we incremented the sum even though we may

have space in memory or we may have scanned the tidset before (and it has not been evicted from memory). This effect is particularly felt for the case where we reorder the itemsets according to increasing support. In this case, the most frequent itemset ends up contributing to the sum multiple times, even though its tidset may already be cached (in memory). For this reason, we also show the number of database scans for the lexical ordering, which are much lower than those for the sorted case. Even with these pessimistic estimates, we find that CHARM makes a lot fewer database scans than the longest frequent itemset. Using lexical ordering, we find, for example on pumsb*, that the longest closed itemset is of length 13, but CHARM makes only 3 database scans.

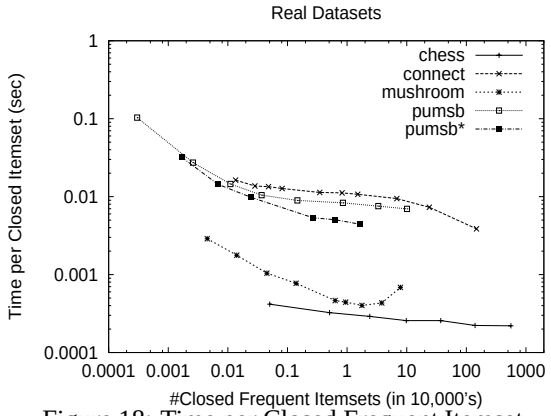

Figure 18: Time per Closed Frequent Itemset

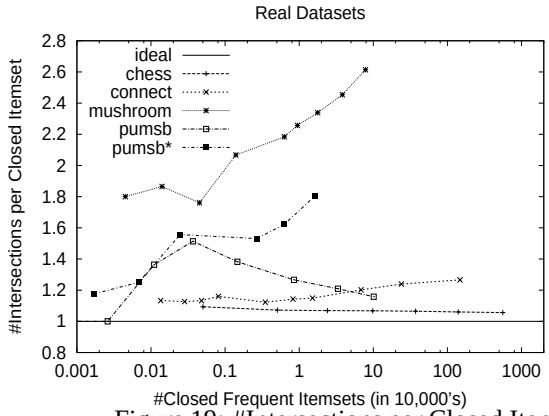

Figure 19: #Intersections per Closed Itemset

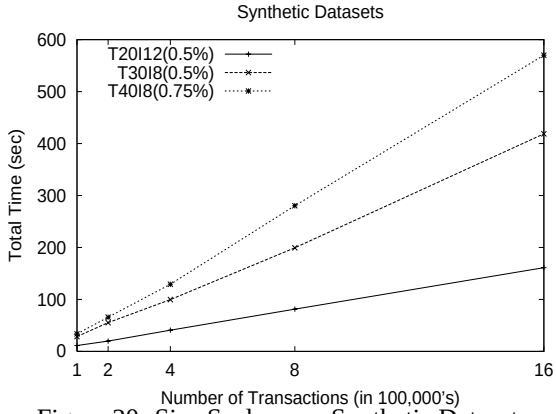

Figure 20: Size Scaleup on Synthetic Datasets

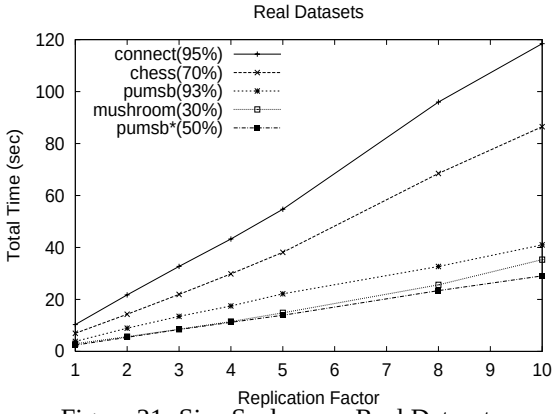

Figure 21: Size Scaleup on Real Datasets

Finally in Figures 20 and 21 we show how CHARM scales with increasing number of transactions. For the synthetic datasets we kept all database parameters constant, and increased the number of transactions from $100K$ to $1600K$. We find a linear increase in time. For the real datasets we replicated the transactions from 2 to 10 times. We again find a linear increase in running time with increasing number of transactions.

7 Conclusions

In this paper we presented and evaluated CHARM, an efficient algorithm for mining closed frequent itemsets in large dense databases. CHARM is unique in that it simultaneously explores both the itemset space and tidset space, unlike all previous association mining methods which only exploit the itemset space. The exploration of both the itemset and tidset space allows CHARM 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.

An extensive set of experiments confirms that CHARM provides orders of magnitude improvement over existing methods for mining closed itemsets. It makes a lot fewer database scans than the longest closed frequent set found, and it scales linearly in the number of transactions and also is also linear in the number of closed itemsets found.

Acknowledgement

We would like to thank Roberto Bayardo for providing us the MaxMiner algorithm, as well as the real datasets used in this paper.

References

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

[3] 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.

[4] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

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

[6] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept formation algorithms based on galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1991.

[7] J. L. Guigues and V. Duquenne. Familles minimales d'implications informatives resultant d'un tableau de donnees binaires. *Math. Sci. hum.*, 24(95):5–18, 1986.

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

[9] 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.

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

[11] M. Luxenburger. Implications partielles dans un contexte. *Math. Inf. Sci. hum.*, 29(113):35–55, 1991.

[12] S. Manegold P. A. Boncz and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *25nd Intl. Conf. Very Large Databases*, September 1999.

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

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

[17] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, to appear, 2000.

[18] M. J. Zaki and M. Ogihara. Theoretical foundations of association rules. In *3rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, June 1998.

[19] 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.