Itemset Trees for Targeted Association Querying

Miroslav Kubat
Department of Electrical and Computer Engineering
University of Miami
1251 Memorial Drive, Coral Gables, FL 33146
mkubat@chronos.ece.miami.edu

Alaaeldin Hafez
Department of Computer Sciences, Faculty of Engineering
Alexandria University
Alexandria, Egypt
ahafez2001@yahoo.com

Vijay V. Raghavan
Center for Advanced Computer Studies
University of Louisiana at Lafayette
Lafayette, LA 70504-4330, U.S.A.
raghavan@cacs.louisiana.edu

Jayakrishna R. Lekkala
Qualcomm, Inc.
5775 Morehouse Drive, L 230B
San Diego, CA 92121
jlekkala@qualcomm.com

Wei Kian Chen
Center for Advanced Computer Studies
University of Louisiana at Lafayette
Lafayette, LA 70504-4330, U.S.A.
weikian@cacs.louisiana.edu

Abstract

Association mining techniques search for groups of frequently co-occurring items in market-basket type of data and turn these groups into business-oriented rules. Previous research has focused predominantly on how to obtain *exhaustive* lists of such associations. However, users often prefer a quick response to *targeted queries*. For instance, they may want to learn about the buying habits of customers that frequently purchase cereals and fruits. To expedite the processing of such queries, we propose an approach that converts the market-basket database into an *itemset tree*. Experiments indicate that the targeted queries are answered in time that is roughly linear in the number of market baskets, N. Also the construction of the itemset tree has $O(N)$ space and time requirements. Some useful theoretical properties are proved.

Keywords. Data mining, association mining, market baskets

1 Introduction

The young field of *association mining* offers exceptional opportunities for the development of efficient algorithms to analyze *market baskets*. A market basket is understood as a list of *items* a customer purchases (as registered at the checkout desk) [9]. A supermarket may deal in thousands of different items and its database is likely to consist of millions of baskets, each comprising, say, 10–20 different items on average. Groups of co-occurring items are called *itemsets*. The *support* of an itemset specifies how many baskets contain this itemset. If 600 market baskets contain [`cheese,butter,milk`], then this itemset's support is 600. The *size* of this itemset is 3.

To know which itemsets are *frequent* (i.e. have supports above a user-selected threshold) is most beneficial. Supermarkets prefer to place associated items on neighboring shelves, advertise them in the same catalogue, and avoid discounts on more than one member of the same itemset. Moreover, itemsets can be turned into business-oriented rules such as "50% of the customers buying cheese and butter will also buy milk." In this case, the rule has the form [`cheese`, `butter`] $\Rightarrow$ [`milk`], and the *confidence* in the rule is 50%. The paradigm extends to a broader range of applications, such as the Web, where a "basket" can represent the set of hyperlinks pointing to a web page, or a digital library where the basket can comprise all references citing a given document. Frequent itemsets then signal relations among web pages or scientific papers.

Classical association-mining techniques focus on *batch* processing that produces a list of itemsets that have sufficiently high supports. The seminal paper by Agrawal et al. [3] introduces the algorithm Apriori (see also [4, 6]) that recursively creates itemsets of size k (containing k items) from those of size $k-1$. The process starts with itemsets of size 1, from which Apriori builds itemsets of size 2, then size 3, and so on, always retaining for further expansion only itemsets whose supports exceed

a given threshold. To counter the exponential nature of the involved computations, several recent papers suggested improvements. Among them, [11] describes a simple mechanism to reduce the number of database scans, [13, 24] accelerate the search by the use of incremental techniques, [8, 21] prune the discovered associations, [5] investigates parallel search, [26] suggests sampling, and [12] addresses issues pertaining to distributed databases.

In this paper, we are interested in applications where the users wish to examine a transaction database by submitting specialized queries [17, 20]. We assume that the user is not interested in a complete list of itemsets, but prefers to constrain the search to specific items. A query of this kind may seek all frequent itemsets containing `bread` and `milk`. Another query may seek all rules of the form $[\texttt{bread}, \texttt{milk}] \Rightarrow s$ (where s is an itemset), with a confidence of at least 30%. In both cases, repeated search for *all* itemsets would waste precious computational resources.

The observation that long response times tend to discourage interactive users has inspired studies of methods that attempt to capitalize on preprocessing. For instance, [22] suggests to *cache* previously discovered itemsets so that future queries can retrieve relevant associations from the cache instead of having to analyze the entire database. The authors of [23] speed up the retrieval by the use of hash tables, and [10] proposes to organize the itemsets in a lattice. To reduce the prohibitive space requirements of this last approach, we introduced a more compact data structure called an *itemset tree* [16]. In this paper, we investigate algorithms that use itemset trees to answer targeted queries, establish some novel properties, and report some illustrative experiments.

Broadly speaking, the technique takes inspiration from the ideas behind *Concept Formation,* a field that received attention in the 1980s [15]. The basic scenario assumes that the system accepts a series of objects and creates from them a hierarchy of "concepts." Each object, as well as each concept, is characterized by a set of *features.* In the context of this work, objects correspond to market baskets and fea-

tures correspond to items. Although these concept-formation techniques primarily seek to model human cognition, their ability to group interrelated items makes them (upon some modification) attractive also to association mining.

Section 2 develops the notion of an itemset tree and introduces an algorithm that generates this data structure from a set of market baskets. Section 3 explains how to use it to process three different query types. Section 4 then reports experiments illustrating the technique's performance in terms of the amount of computation needed to answer a representative query. The same section briefly investigates the costs of building an itemset tree from data. Section 5 establishes the existence of a one-to-one mapping between the space of itemset trees and the space of market-basket databases. This mapping guarantees that the resulting itemset tree does not depend on the order in which the market baskets have been presented. Section 6 offers some concluding remarks and ideas for future research.

2 Building Itemset Trees

Let D denote a set of itemsets and let M be the number of distinct items encountered in D. Each item is identified with an integer from $[1, M]$, so that the items in an itemset, $s = [a_1, a_2, \ldots, a_p]$, can be ordered: $a_i < a_j$ for $i < j$, where a_i and a_j are integers identifying i^{th} and j^{th} items respectively.

Definition 2.1 (ancestor, largest common ancestor, child) *Let the symbols s, c, and l denote itemsets.*

1. *We say that s is an* ancestor *of c, and write $s \sqsubseteq c$, iff $s = [a_1, a_2, \ldots, a_m]$, $c = [a_1, a_2, \ldots, a_n]$, and $m \leq n$.*
2. *We say that l is the* largest common ancestor *of s and c, and write $l = s \sqcap c$, iff $l \sqsubseteq s$, $l \sqsubseteq c$, and there is no l' such that $l' \sqsubseteq s, l' \sqsubseteq c$, and $l \sqsubseteq l'$, $l \neq l'$.*

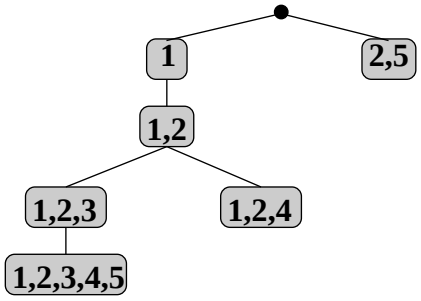

Figure 1: Illustration of the relations in a group of itemsets

3. We say that c is a child of s *iff $s \sqsubseteq c$ (s is ancestor of c) and there is no l, different from s and c, such that $s \sqsubseteq l \sqsubseteq c$.* □

Example.

Consider the database $D = \{[1], [1,2], [1,2,3], [1,2,4], [1,2,3,4,5], [2,5]\}$ depicted in Figure 1. Here, $[1,2]$ is an ancestor of $[1,2,3,4,5]$. The largest common ancestor of $[1,2,3]$ and $[1,2,4]$ is $[1,2]$. Note that $[1]$ is *not* the largest common ancestor of these two itemsets. Further on, $[1,2,4]$ is a child of $[1,2]$ but $[1,2,3,4,5]$ is *not* a child of $[1,2,4]$ because $[1,2,4]$ is not an ancestor of $[1,2,3,4,5]$ (since the item "3" is not in $[1,2,4]$).

Note that an ancestor of c can be viewed as an uninterrupted sequence of the smallest items of c. For instance, baskets $[1,2]$ and $[2,5]$ cannot have a common ancestor according to Definition 2.1 because any ancestor of $[1,2]$, when listed from the smallest to the largest number, will begin with "1," which is not contained in $[2,5]$. This case is dealt with by having a root node whose itemset is empty. By definition, an empty set is a (trivial) ancestor of any non-empty itemset. Note that if s is an ancestor of c, then s is also the largest common ancestor of s and c.

Lemma 2.1 (relations between two itemsets) *Let s and c be itemsets. Then one and only one out of the following conditions holds:*

1. $s \sqcap c = \emptyset$.
2. $s = c$.
3. $(s \neq c) \wedge (s \sqsubseteq c)$.
4. $(s \neq c) \wedge (c \sqsubseteq s)$.
5. $(s \neq c) \wedge (s \sqcap c = l)$ *such that* $l \neq s, l \neq c$.

Proof.
Condition 1 represents the case when two itemsets have only the empty set as the common ancestor. If they have some other common ancestor, then they are either equal to each other (condition 2) or different from each other. If they are different from each other, then one can be an ancestor of the other (conditions 3 and 4) or they have a common ancestor different from either of them (condition 5). □

With the terminology and basic relations among itemsets being established, we can formally specify the notion of an itemset tree.

Definition 2.2 (itemset tree) *An* itemset tree, *T, consists of a* root *and a (possibly empty) set, $\{T_1, T_2, \ldots, T_k\}$, each element of which is an itemset tree. The root is a pair $[s, f(s)]$ where s is an itemset and $f(s)$ is a frequency. If s_i denotes the itemset associated with the root of the i-th subtree, then $s \sqsubseteq s_i, s \neq s_i$, must be satisfied for all i.*

Note that any itemset tree that satisfies Definition 2.2 is a partially ordered set of the pairs [itemset, frequency]. This means that the concrete layout of a root's children is irrelevant as long as they all occupy the same level in the tree. Sometimes, we will draw a difference between an *internal node* and a *leaf node*. By a leaf node, we understand a tree with an empty set of subtrees. All nodes that are not leaf

nodes are internal nodes. The itemset associated with the root may be empty, while the itemsets associated with all other internal and leaf nodes are nonempty (note that this additional condition does not violate Definition 2.2). Table 1 describes an algorithm that builds such an itemset tree incrementally, through a series of market basket insertions. This means that the itemset tree is constructed during a single pass through the given set of market baskets.

The principle of the algorithm can be summarized by the following procedure. Let $R = [s_R, f(s_R)]$ denote the root, let s denote the next market basket to be inserted, and let c_i denote the children of R. Without loss of generality, assume that s_R is an ancestor of s. If $s \neq s_R$, then the following possibilities are considered. In the simplest case, s_R is the only common ancestor of s with any c_i or the set of R's children is empty (case 1). In this event, a new child, $[s, f(s) = 1]$, is appended to R. If s is identical to some c_i (case 2), then the frequency of c_i is incremented by 1. A more instructive situation occurs when s *does* have a common ancestor with some c_i. Then, one and only one of the conditions 3–5 of Lemma 2.1 is satisfied. If s is an ancestor of some c_i such that $s \neq c_i$ (case 3), then s is inserted between the root and c_i, its frequency being set to $f(s) = f(c_i) + 1$. If some c_i, such that $s \neq c_i$, is an ancestor of s (case 4), then the algorithm is recursively called for the subtree rooted at c_i. Finally (case 5), if l is the largest common ancestor of s and c_i such that $l \neq s$ and $l \neq c_i$, then a node $[l, f(l) = f(c_i) + 1]$ is inserted as a new child of R, and s and c_i become the children of l. Whatever the action, the root's frequency, $f(s_R)$, is always incremented by 1.

It is easy to see that the algorithm inserting a market basket in an itemset tree is finite. The only time when the recursion is called is when the conditions of case 4 are satisfied. The last time this can happen is when c_i (in the command *Construct(s,T(*c_i*))* is in a leaf node. Since a leaf node does not have any children, the only choice Algorithm 1 has is case 1. This prevents the recursion from going any deeper.

Table 1: Algorithm 1, incorporating a market basket, s, in an itemset tree, T.

Construct(s, T).

Let $R = [s_R, f(s_R)]$ denote the root of T and let $[c_i, f(c_i)]$ be R's children.

if $s = s_R$ **then** set $f(s_R) := f(s_R) + 1$

else if s is an ancestor of s_R **then**

a. create a new root $[s, f(s) := f(s_R) + 1]$;

b. make $[s_R, f(s_R)]$ the child of the new root.

else if l is the common ancestor of s and s_R (such that $l \neq s, l \neq s_R$) **then**

a. create a new root, $[l, f(l) := f(s_R) + 1]$;

b. append to l two children: $[s, f(s) := 1]$ and $[s_R, f(s_R)]$.

else

a. set $f(s_R) := f(s_R) + 1$

b. select an action from the following list:

1) **if** s_R is the only common ancestor of s with any child of R, or if the set of R's children is empty, **then** create a new leaf, $[s, f(s) := 1]$, and make it a child of R.
2) **if** $c_i = s$, **then** $f(c_i) := f(c_i) + 1$.
3) **if** s is an ancestor of c_i, **then** insert $[s, f(s) := f(c_i) + 1]$ between R and c_i.
4) **if** c_i is an ancestor of s, **then** let $T(c_i)$ be the subtree rooted at $[c_i, f(c_i)]$; call *Construct*$(s, T(c_i))$.
5) **if** l is the largest common ancestor of c_i and s (such that $l \neq s, l \neq c_i$) and the itemset associated with R is an ancestor of l, **then**:
   i. disconnect $[c_i, f(c_i)]$ from R;
   ii. create a new node, $[l, f(l) := f(c_i) + 1]$ as a child of R;
   iii. append to l two children: $[s, f(s) := 1]$ and $[c_i, f(c_i)]$.

Example.

Figure 2 shows what happens if Algorithm 1 processes the market baskets from database $D = \{[1,4],[2,5],[1,2,3,4,5],[1,2,4],[2,5],[2,4]\}$. The following steps are carried out.

1. The first basket is turned into the root, $R = [[1,4],1]$.
2. The second basket, $[2,5]$, and $s_R = [1,4]$ are not equal, but have the empty set as a trivial ancestor. Here, as indicated in Figure 2, a new root, l is created, where $l = [\emptyset, f(l)]$, and $f(l) = f(s_R) + 1 = 2$. The current root, R, and a new node $[[2,5],1]$ are made the children of l.
3. The largest common ancestor of the basket $[1,2,3,4,5]$ and the itemset in the node $[[1,4],1]$ is $l = [1]$. Therefore, two new nodes are created (case 5).
4. $c_i = [1]$ is an ancestor of the fourth basket, $[1,2,4]$ (case 4). This basket is therefore propagated down the subtree rooted at $[1]$. The recursive call reveals that $[1,2,4]$ has a common ancestor with $[1,2,3,4,5]$ and that the largest such ancestor is $[1,2]$. A new node is created according to case 5.
5. The fifth basket, $[2,5]$, is identical with one of the root's children (case 2). The algorithm's only action is to update this node's frequency.
6. The sixth basket, $[2,4]$, turns out to have a common ancestor with $[2,5]$ (case 5). The largest such ancestor, $[2]$, becomes the child of R.

□

Note that the algorithm guarantees that one and only one of the five actions from Table 1 is taken. Moreover, it can never happen that two of the root's children satisfy one of the conditions 2–5 from Lemma 2.1 at the same time.

The "size" of an itemset tree is given by the number of nodes (including the root) and by the number of layers (root being the first layer). Simple analysis of Algorithm 1 reveals that no new node is created in case 2; in the event of case 1 or case 3, one new node is created; case 5 creates two new nodes. In case 4, the decision is postponed but, still, processing a single basket will never give rise to more than

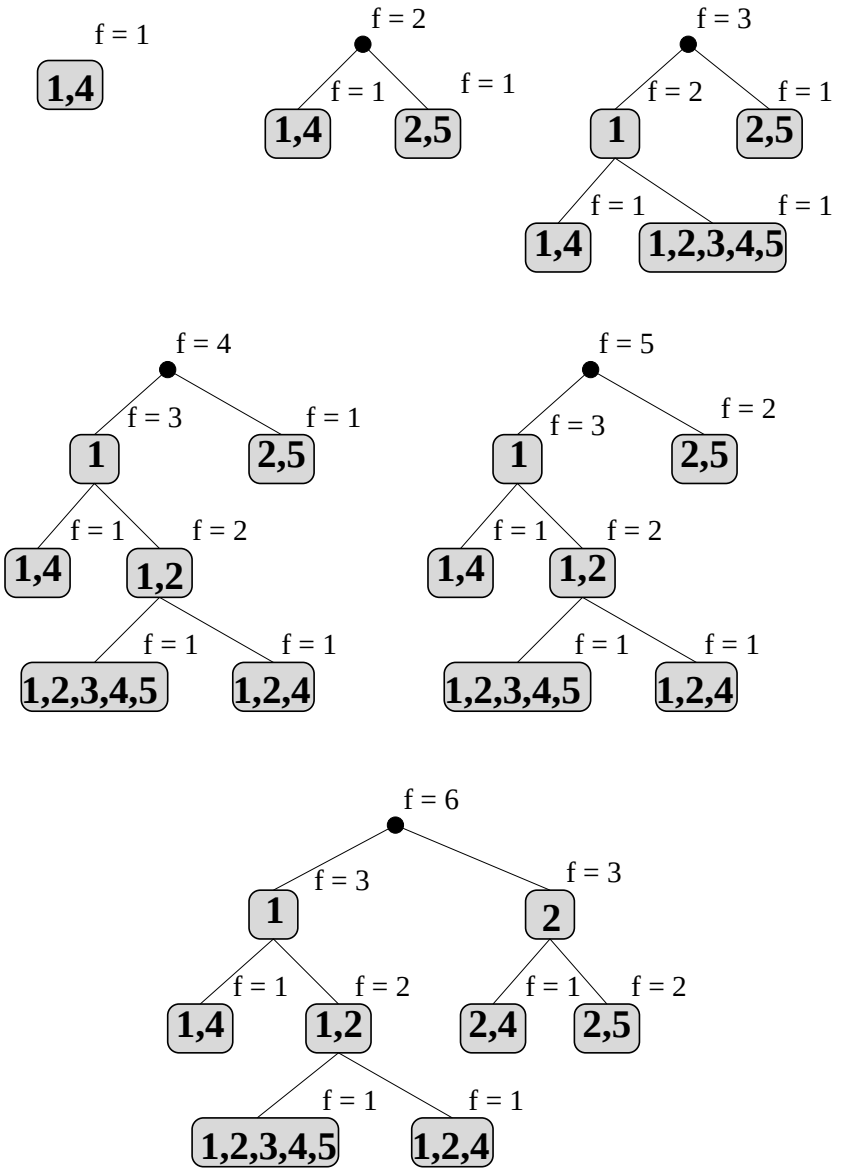

Figure 2: Itemset-tree construction from the following sequence of market baskets: $[1, 4], [2, 5], [1, 2, 3, 4, 5], [1, 2, 4], [2, 5], [2, 4]$.

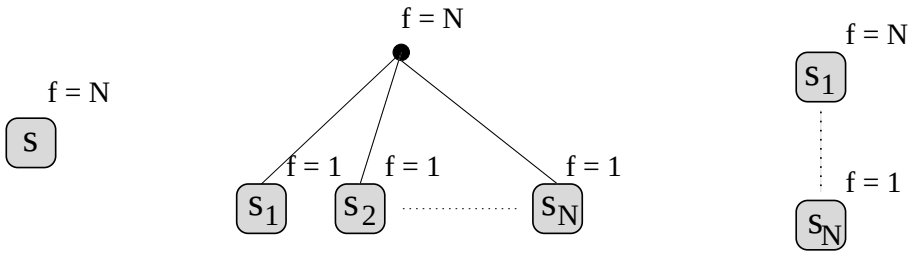

Figure 3: Three extreme cases of itemset trees: a single-child tree (left), a single-layer tree (middle), and a linear tree (right).

two new nodes, and thus cannot increase the depth of the tree by more than one layer. This proves the following lemma.

Lemma 2.2 (itemset-tree size) *Let T denote the itemset tree generated from a database of N distinct market baskets using Algorithm 1.*

1. *The number of nodes in T is upper bounded by $2N + 1$.*
2. *The number of layers in T is upper bounded by $N + 1$.*

We conclude that the *memory requirements* of the resulting itemset tree are comparable with the size of the original database, and thus do not represent a major obstacle to employing the technique in real-world domains.

Note that, depending on the contents of the database D, three extreme cases, depicted in Figure 3, are possible. The first of them happens when D contains N identical market baskets. In this event, the generated itemset tree consists of a single node, $[s, f(s) = N]$. The second extreme case happens when no two baskets in D have a common ancestor. Then, the tree consists of the root and a single layer with N children. Finally, if the baskets can be ordered in a sequence $s_1, \ldots, s_N$, such that s_i is an ancestor of s_{i+1} for any $i \in [1, N-1]$, then each layer will contain a single node.

Our discussions up to this point do not provide an interpretation of the frequencies stored in the itemset tree. The next lemma is introduced to lead into such an interpretation. The actual use of the frequencies in computing itemset supports will be considered in section 4.

Lemma 2.3 (frequency) *The frequency of a node in an itemset tree equals the number of market baskets that have passed through this node during the construction of the itemset tree. The frequency of a child is smaller than the frequency of the parent.*

Proof.
The proof follows immediately from Algorithm 1. Each time a market basket is inserted in the itemset tree, the frequency of each node such that the itemset associated with it is an ancestor of the incoming basket is incremented by 1. The parent's frequency is incremented before a child's frequency is incremented. □

3 Exploiting Itemset Trees

Having described how to create itemset trees from market-basket, we will now show how to exploit this data structure for simple association mining tasks. We will introduce three techniques. First of them calculates the support of an itemset; the second finds all frequent itemsets that contain a user-specified list of items; and the third generates all rules that exceed a user-specified minimum confidence and have a user-specified itemset as antecedent.

- *Task 1: establishing the support of an itemset.*
  Input: an itemset.
  Output: this itemset's support.

Table 2: Algorithm 2 that uses itemset tree T to find the support of itemset s.

Let R denote the root of T and let $[c_i, f(c_i)]$ be R's children.
Let T_i denote the subtree whose root is $[c_i, f(c_i)]$.
To invoke *Count,* use: $f(s) := Count(s, T, 0)$.

Count(s,T,f(s)).

For each c_i such that $first_item(c_i) \leq first_item(s)$:

1. **If** $s \subseteq c_i$ **then return** $(f(s) + f(c_i))$.
2. **If** $s \not\subseteq c_i$ and $last_item(c_i) > last_item(s)$,
   then return $(f(s) + Count(s, T_i, f(s)))$.

The algorithm from Table 2 sums up the frequencies of all nodes that *subsume* s (we say that c_i subsumes s, and write $s \subseteq c$, if any item contained in s is also contained in c). At the beginning, the procedure sets $f(s) = 0$, and then examines the itemsets, c_i, associated with the root's children. If the first item of some c_i is greater than the first item in s, then the itemset-tree building algorithm guarantees that none of the nodes of the subtree rooted at c_i subsumes s. This means that the subtree does not affect the support $f(s)$ and can be ignored.

Three different situations can occur in the event of $first_item(c_i) \leq first_item(s)$. First of them is marked by $s \subseteq c_i$, which means that each basket that has passed through this node contained the itemset s. Therefore, it is enough to update the support, $f(s) = f(s) + f(c_i)$, where $f(c_i)$ is the frequency of the child containing c_i. Second, if $s \not\subseteq c_i$ but $c_i \sqsubseteq s$, the algorithm is applied recursively to the subtree rooted at c_i. Third, if neither condition is satisfied, then none of the nodes in the subtree rooted at c_i can subsume s. In this case, $f(s)$ is not updated.

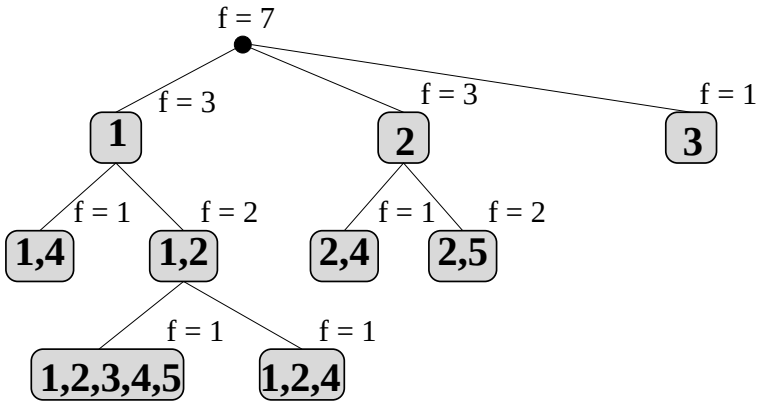

Figure 4: An example itemset tree.

By way of illustration, suppose that the itemset tree from Figure 4 is queried, and that the user wants to know the support of [2,5]. The condition in *Count*'s loop will exclude the rightmost branch because $\mathit{first_item}([3]) \not\leq \mathit{first_item}([2,5])$. Condition 1 will apply to the left node at the lowest level of the tree ([1,2,3,4,5]) and to the rightmost node at the second level ([2,5]). Investigating the substree below the node [1], condition 2 is satisfied for all children, so the recursive call is made for all of them.

To see the motivation for condition 2, suppose that the algorithm reaches (in some hypothetical tree) a node that contains [2,3,4,7] and the user wants to know the frequency of [2,5]. Then, it is true that $[2,5] \not\subseteq [2,3,4,7]$ but it still does not make sense to investigate the subtree associated with this node because each of this node's children will contain [2,3,4,7] as the leading part of its itemset. As a result, none of these children can contain item 5.

The reader can see that the way the itemsets are organized makes it very likely that to answer an average query, the system will have to visit only a subset of the nodes. Many nodes will be excluded from further consideration by the conditions

Table 3: Algorithm 3 that processes the itemset tree, T, and returns the list of all itemsets that contain user-specified items (user-specified itemset is denoted by s).

Let R denote the root of T and let $[c_i, f(c_i)]$ be R's children.
Let T_i denote the subtree whose root is $[c_i, f(c_i)]$.
To invoke *Selective_mining* use: $L := Selective_mining(s, T, \{\})$.

$Selective_mining(s, T, L)$.

For each c_i such that $first_item(c_i) \leq first_item(s)$:

1. **If** $s \subseteq c_i$ **then**
   i. Let L' be the set of all itemsets, l, such that $s \subseteq l \subseteq c_i$.
   ii. **return** $(L \cup L')$.
2. **If** $s \not\subseteq c_i$ and $last_item(c_i) > last_item(s)$,
   then return $(L \cup Selective_mining(s, T_i, L))$.

$first_item(c_i) \leq first_item(s)$ and $c_i \sqsubseteq s$. □

- *Task 2: selective search for itemsets.*
  Input: an itemset, s, and a threshold θ.
  Output: all itemsets that subsume s and have support at least θ.

The algorithm in Table 3 returns the list of all itemsets that subsume the user-specified itemset s and have support at least some θ. Note that the algorithm is similar to the one described in Table 2. Both of them constrain their searches by comparisons of the first items in s and c_i. Further on, they both test the subsumption and ancestor relations between s and c_i.

However, there are also differences. In the event of $s \subseteq c_i$, *all* itemsets that are subsumed by c_i, and themselves subsume s, have to be appended to the list,

Table 4: Algorithm 4 takes results of Algorithm 3 and extracts all distinct itemsets that subsume a user given itemset and have supports above a user-set threshold, θ.

Let L denote the set of itemsets returned by algorithm 4.
To invoke *All_frequent_sets* use: $All_frequent_sets(L, \theta)$

$All_frequent_sets(L, \theta)$

1. Merge all duplicates found in L (and recalculate the respective supports)
2. Remove from L all itemsets whose supports are less than θ.

L, of itemsets subsuming the user-specified items. Moreover, once the search has been completed, two additional steps have to be accomplished. First, some itemsets might be contained in L more than once, and it is thus necessary to merge these duplicates into a single occurrence (adding up their supports). Then, itemsets with supports below θ are removed. These steps are listed in the algorithm in Table 4.

Again, note that to answer a query of this kind, the system does not have to search through the entire database. This means that the complexity of this algorithm is upper bounded by $O(N)$ where N is the number of market baskets. □

- *Task 3: selective generation of rules.*
  Input: an itemset, s, and a confidence threshold, θ_C.
  Output: all rules, $s \Rightarrow l$ that have confidence above θ_C.

Itemsets discovered by association mining algorithms are often used to generate rules of the form $s \Rightarrow l$, where s and l are itemsets. An interpretation of a rule like this can be: "60% of the customers buying `milk` and `butter` will also buy `cheese`." The percentage (in this case, 60%) is sometimes called a *confidence* in the rule. Naturally, we expect the technique of itemset trees to be able to support also this

Table 5: Algorithm 5 that processes the itemset tree, T, and returns the list of all rules that have a user-specified itemset, s, as antecedent and have confidence about a user-specified threshold θ_C. Care is taken that only rules satisfying user-specified minimum support, θ_S, are returned.

To invoke *Rules* let $L_R := \emptyset$ and call $Rules(s, T, L_R, \theta_C, \theta_S)$

$Rules(s, T, L_R, \theta_C, \theta_S)$.

1. Use Algorithm 4 and Algorithm 5 to obtain a list, L, of itemsets containing s.
2. **For each** itemset, c, from the list L
   i. Create itemset l such that $s \cap l = \emptyset$ and $s \cup l = c$.
   ii. Using Algorithm 3, obtain the supports of c and l, denoted by $g(c)$ and $g(l)$, respectively.
   iii. Calculate the confidence of the rule $s \Rightarrow l$ as $C = g(l)/g(c)$.
   iv. **If** $C \geq \theta_C$, **then** add the rule to L_R.

---

type of query.

The algorithm accomplishing this goal is summarized in Table 5. In the first step, all itemsets containing s and satisfying the user-specified threshold, θ_S (for minimum support), are found by Algorithm 3 and Algorithm 4 described above. Suppose an itemset c has been discovered and denote by l all items from c that are *not* contained in s. These itemsets form the basis of the rule $s \Rightarrow l$. If the supports of the itemsets c and l are denoted by $g(c)$ and $g(l)$, respectively, then the confidence of this rule is calculated as the ratio $g(l)/g(c)$. All rules satisfying the requirement that $g(l)/g(c) \geq \theta_C$ are returned to the user. □

Example.

Suppose the user submitted itemset $s = \{2, 3, 6\}$ and that one of the itemsets containing all these three items is $c = \{2, 3, 4, 6, 8, 10\}$. This gives rise to the rule $\{2, 3, 6\} \Rightarrow \{4, 8, 10\}$. To obtain the confidence in this rule, the supports $g(s)$ and $g(l)$ of the itemsets $s = \{2, 3, 6\}$ and $l = \{4, 8, 10\}$ are determined by Algorithm 3. If $g(l)/g(c) \geq \theta_C$, then the rule is accepted. Otherwise, it is removed from the list of candidates.

4 Experiments

The experiments are intended to provide some insight into the computational costs associated with the techniques described above. First of all, we want to learn whether the query anwering is sufficiently fast. If yes, then the question is whether the costs of building the itemset tree are affordable. It turns out that the query processing is very fast and that itemset trees have $O(N)$ requirements in space and time, where N is the number of market basket in the database.

In the sequel, we first describe how synthetic testbeds were generated. Then, we present experimental results separately for query processing and for itemset tree construction.

4.1 Experimental Data

As is common in the association-mining literature, we experimented with synthetic data whose characteristics are easy to control. To this end, we implemented a data generator that accepts as parameters the requested number of market baskets, the average size of a market basket, and the number of distinct items encountered in the database.

To create experimental market baskets, we implemented a simple generator. In

the first step, the generator chooses the size of the market basket, using a source of random numbers (uniform distribution) that takes as input a user-set mean value and range (minimum and maximum value). Then, each item to be placed in this market basket is obtained from the normal distribution defined by a user-provided mean and standard deviation (the output number is rounded to the nearest integer). Yet another parameter, M, specifies the number of different items—all values outside the interval $[1, M]$ are discarded.

We consider two testbeds. In the first, the number of different items is $M = 500$ so as to model domains where an average item is found in many market baskets. In domains of this kind, two randomly drawn market baskets have a high chance of having non-empty intersection. More specifically, the item generator used the mean value of 125 and the standard deviation of 250 (and the range [1,500]). The other testbed assumes that the number of distinct items is $M = 10,000$. The intention is to model a domain where the contents of market baskets do not overlap so frequently. In this case, the mean value is 2,500 and the standard deviation is 5,000 (and the range is from 1 to 10,000).

For each of the two domains, we generated 100,000 market baskets with average sizes either 10 or 100. The intention is to measure the response times for itemset trees that have been created from databases of different parameters.

4.2 The Costs of Query Answering

To estimate the average computational costs of query answering, we measured the number of nodes (in the itemset tree) that Algorithm 3 had to visit when trying to obtain a complete list of itemsets to be returned to the user. In this way, we eliminate the influence of programming subtleties and focus on the real contribution of the proposed data structure.

To start with, we used Algorithm 1 to create itemset trees from databases synthe-

sized by the data generator. From the three different types of queries introduced in Section 3, we chose the second type as the most typical representative—it subsumes the first type, and the last type only adds insignificant additional computation. We therefore assume that the user submits to the system a list of items and receives, in return, a complete list of itemsets that contain all items from this list. For instance, if the user submits [2,5], the system returns all itemsets containing items 2 and 5. In the experiments, we considered two different situations. In the first, the user submits 2 distinct items that have to be contained in each of the returned itemsets; in the second, the user submits a list of 8 distinct items. To obtain statistically reliable results, we always generated $1,000$ random queries and then calculated the average number of nodes visited per query.

The graphs in Figure 5 summarize the results by plotting the number of visited nodes against the size of the original database (from $10,000$ through $100,000$ market baskets). Two domains are considered, one with 500 distinct items (left), and the other with $10,000$ distinct items (right).[1] The average size of a market basket was 10 in the upper graphs and 100 in the bottom graphs. Each graph contains two curves: one for queries containing 2 items, and one for queries containing 8 items.

The reader can see that, to answer a query, the system had to traverse only a relatively small portion of the itemset tree. In fact, the number of visited nodes grows less then linearly in the size of the original database. The higher the pairwise overlaps of the individual markets, the bigger the part of the itemset tree that has to be traversed to answer the query. This is because a smaller portion of the nodes are eliminated by the conditions from Algorithm 3.

Figure 6 expresses the same results in percentages. The reader can see that

[1] The size of a tree generated from $10,000$ different items did not differ much from the size of a tree generated from 500 different items as long as the tree was created from the same number of market baskets (the difference never exceeded 10%). The number of nodes in the generated itemset tree never exceeded the number of market baskets by more than 30%.

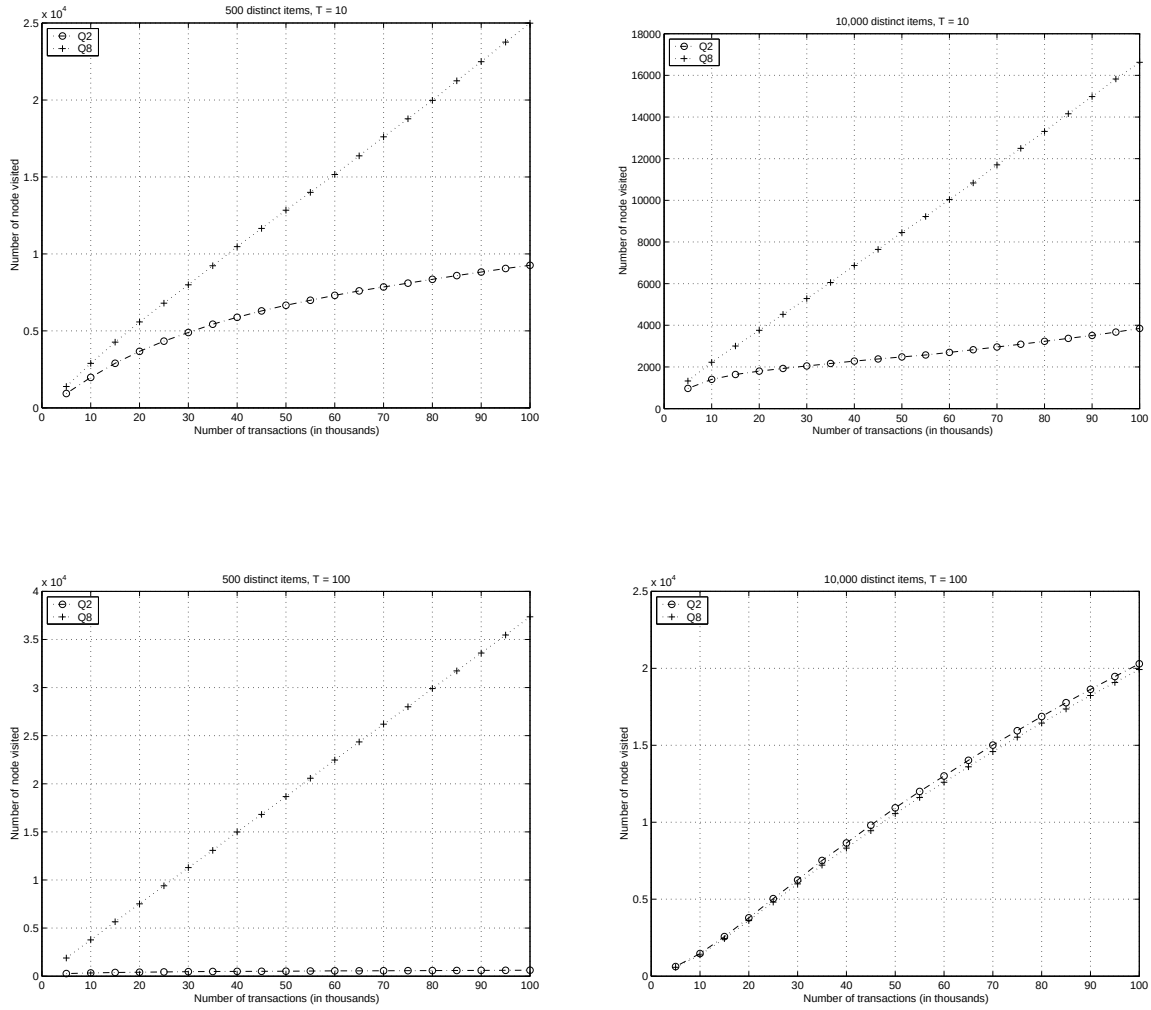

Figure 5: Average number of itemset-tree nodes visited per query. Along the horizontal axis, the size of the original database is plotted. Along the vertical axis, the number of visited nodes is plotted. T is the average number of items in a market basket. $Q2$ and $Q8$ refer to queries consisting of 2 and 8 items, respectively.

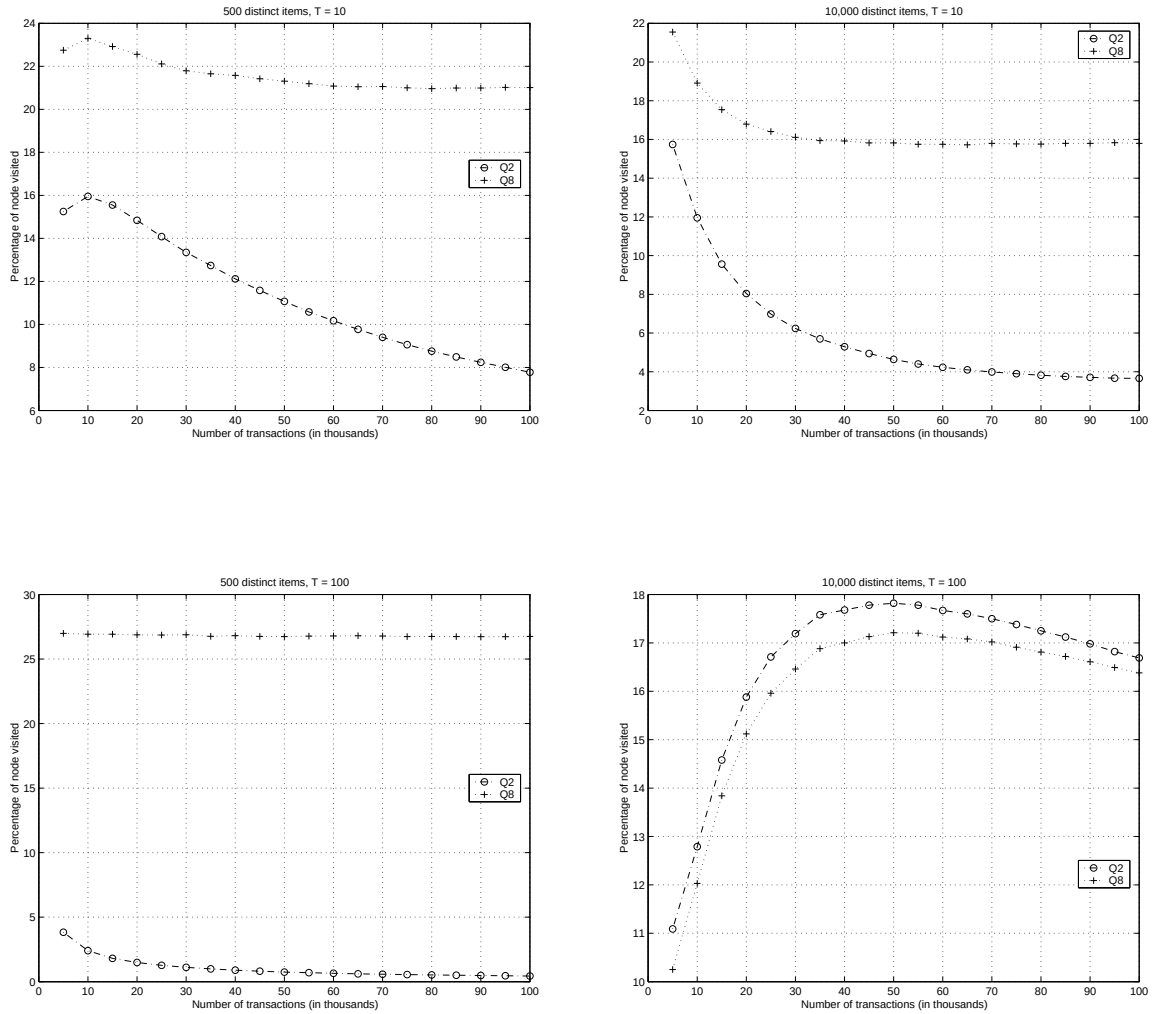

Figure 6: Average number of itemset-tree nodes visited per query. Along the horizontal axis, the size of the original database is plotted. The vertical axis plots the percetage of nodes that were visited per average query. T is the average number of items in a market basket. $Q2$ and $Q8$ refer to queries consisting of 2 and 8 items, respectively.

the percentage of nodes visited per query always drops with the size of the original database. This demonstrates the scalability of the techniques to large, real-world databases.[2] In none of these experiments did the query answering modul have to visit more than a quarter of the nodes. In most of the cases, the percentage was actually much smaller than that.

At the same time, the figure shows how difficult it is to come up with a heuristic that would predict the response time based on the characteristics of the underlying data. Obviously, a complex interplay of the two conditions from Algorithm 3 affects the size of the traversed portion of the itemset tree. For instance, in queries consisting of 8 items, the average "distance" between the first and the last integer is bigger than in queries consisting of 2 items. Therefore, the chance that $s \subseteq c_1$ (the first condition) is also smaller. At the same time, the second condition ($last_item(c_i) > last_item(s)$) is also affected. A detailed study of the related phenomena would perhaps deserve a separate paper. Here, our objective was only to show the relatively low costs of query answering and the scalability of the technique. For a database of 10^5 market baskets, the query answering module typically had to visit some $10,000$–$20,000$ nodes, carrying out relatively inexpensive operations on each of them. This amounts to response time of no more than a few seconds.

4.3 Building an Itemset Tree

Having found that itemset trees facilitate very fast query answering, we wanted to make sure that building the trees from market basket databases is affordable. Lemma 2.2 established the upper bound on the number of nodes created from N distinct market baskets as $2N + 1$. To insert a single market basket to an existing itemset tree, Algorithm 1 must visit less then the total number of existing nodes. This means that the costs of inserting N market baskets cannot exceed $O(N^2)$. In

[2]Scalability is a major problems of many data mining techniques.

reality, the computation complexity can be expected to be much lower because the number of nodes visited per one insertion will constitute only a fraction of N.

The theoretical results published earlier [16] indicate that the expected costs, under certain assumptions, can be roughly approximated by a linear function. Our next series of experiments was designed to verify this result. We measured the CPU time (SUN Sparc10, program written in C) that Algorithm 1 needed to convert a market-basket database into an itemset tree. The goal was to develop some insight into how this time depends on the number of market baskets.

Again, two sets of experiments were conducted to determine whether the overall results are affected by the extent to which the contents of a pair of randomly chosen market baskets overlap. In the first set, the number of distinct items was 500; in the other, this number was $10,000$. The independent variable is the size of the database. To diminish the effect of statistical fluctuations, we synthesized five different databases for each database size, and then calculated the average times the itemset tree construction took.

Figure 7 shows the results. The reader can see that the curve is roughly linear, indicating reasonable computation costs for both of the experiments. In the right-hand plot (for $10,000$ distinct items), the computation costs were higher, but they still grew more or less linearly in the size of the database. The standard deviations were typically around 10%.

We also wanted to know whether the computation costs somehow depend on the average size of a market basket. To this end, we ran a series of experiments, where the number of market baskets was fixed at $2,000, 4,000$, and $10,000$, while the size of an average market basket varied from 10 to 100. We assume that higher values do not make much sense in supermarket type of data. Again, we measured the consumed CPU time.

The results, shown in Figure 8, indicate that, in the case of 500 distinct items, the costs of itemset-tree generation do not appear to be affected by the market-basket

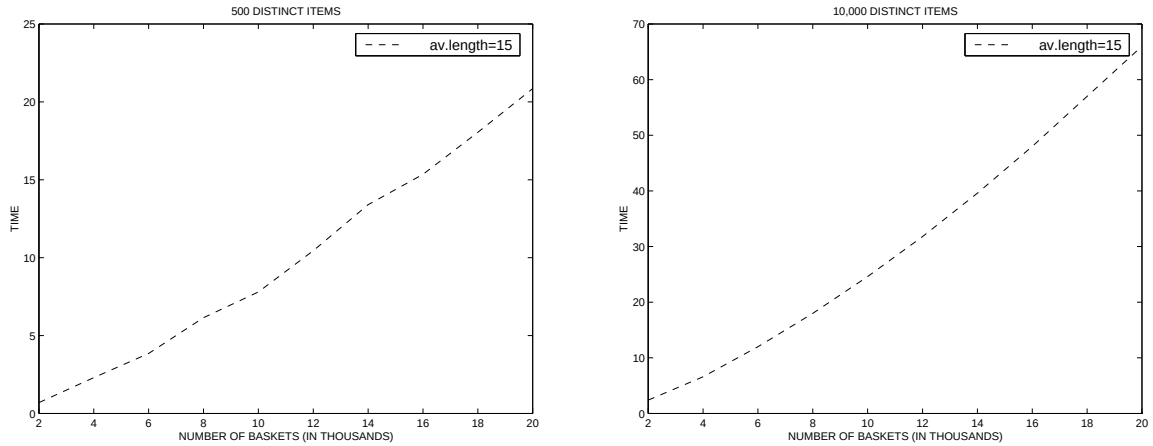

Figure 7: CPU costs of itemset-tree construction versus the number of market baskets for 500 distinct items (left) and 10,000 distinct items (right).

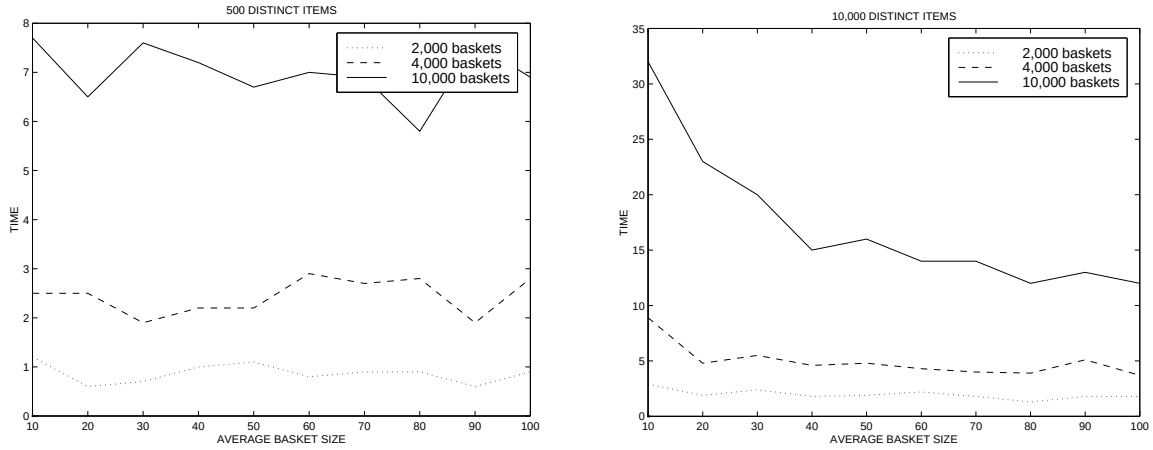

Figure 8: CPU costs of itemset-tree construction versus average size of a market basket for 500 distinct items (left) and 10,000 distinct items (right).

size. In the case of $10,000$ distinct items, these costs decrease with growing size of market baskets, but only for the case where database was large. These tendencies are encouraging. Again, the plots depict averages from 5 runs. The standard deviations were typically around 10–15%.

4.4 Summary of the Experimental Results

The experiments indicate that the time needed to answer an average query is approximately linear in the number of market baskets contained in the original database (and thus, essentially, also in the size of the itemset tree). In reality, only a fraction of the tree's nodes (typically some 5–20%) have to be visited which, for a database containing $100,000$ market baskets could be accomplished in a few seconds on SUN Sparc10. An itemset tree representing hundreds of millions of market baskets may entail responses that exceed the usual expectations for the on-line mode of operation. Still, this limitation may not be prohibitive. Trying to answer the query using the original data would certainly be much more expensive, and the same applies to the use of Apriori-based algorithms, even when considering all the improvements enumerated in the Introduction. Moreover, we believe that methods of querying the itemset tree can be further improved and the same applies to their efficient implementation. With the growing performance of hardware, we believe that even very large market-basket databases can soon be queried on-line.

This efficiency can be achieved fairly easily because the construction of an itemset tree incurs computational costs that grow roughly linearly in the size of the original database. The CPU time does not appear to be affected by the size of an average market basket, at least not when this size reflects circumstances typical of supermarket data (10–100 items in each market basket). In view of the fact that the itemset tree has to be constructed only once (which can be accomplished off-line), these costs do not represent major obstacle.

Importantly, the itemset tree is very easy to update: whenever a new set of market baskets become available, they can be incorporated in the tree one by one, using Algorithm 1. The Uniqueness Theorem proved in the next section guarantees that the resulting itemset tree is independent of the order in which the market baskets have been presented. In our opinion, this incremental nature is one of the main assets of our approach.

5 The Uniqueness Theorem

Before showing how to exploit the data structure for association mining, let us first ask whether the itemset tree generated by Algorithm 1 depends on the sequence of basket presentation. In this context, recall that Definition 2.2 does not assume any ordering of the children of a root. Although the children do seem to be ordered when the tree is graphically displayed (e.g., in Figure 2), this order does not have any functional interpretation. In other words, two itemset trees are equal if they only differ in the order in which the children of any of its nodes are displayed. We present a lemma needed to prove the main result of this section.

Lemma 5.1 *Let S_1 be a sequence of baskets and let S_2 be another sequence obtained by swapping a pair of neighboring baskets in S_1. If the itemset tree T_1 is built from S_1 and the itemset tree T_2 is built from S_2, then $T_1 = T_2$.*

Proof (sketch).
Let A and B denote the two swapped market baskets (S_1 and S_2 differ only in the order of A and B). Suppose an itemset tree has been built from the sequence preceding A and B. Let c_i denote the child of the root of this tree (each child will have a different index, i). If A and B each affect a different child, then the order in which they are presented is obviously irrelevant. We can also ignore as trivial the

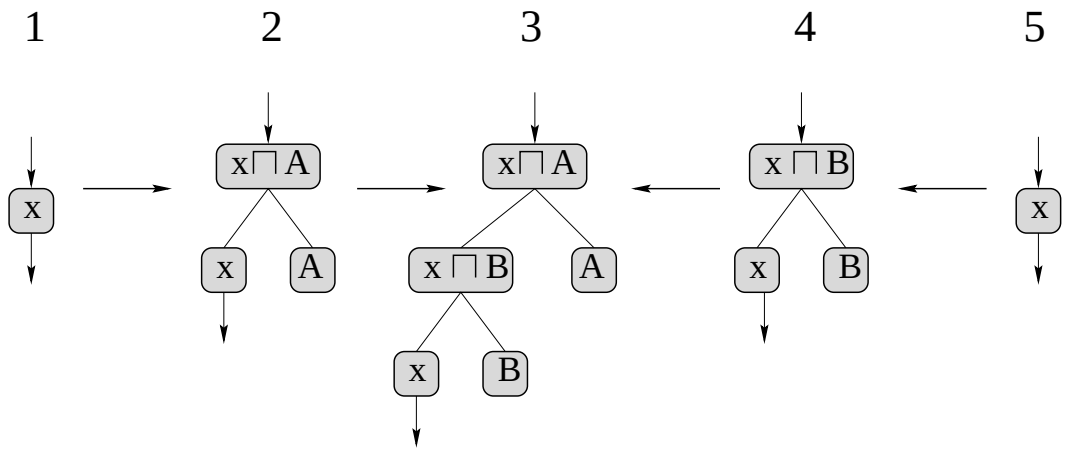

Figure 9: Two market baskets, A and B, are incorporated in an itemset tree.

case where no c_i has a common ancestor with either A or B. Finally, we can ignore the case where $A \sqsubseteq c_i \sqsubseteq B$ (which subsumes the case $A = c_i$ such that $c_i \sqsubseteq B$).

Suppose there is some c_i such that $A \sqcap c_i \neq \emptyset$ and $B \sqcap c_i \neq \emptyset$. Algorithm 1 is called recursively for the subtree rooted at c_i and the recursive calls continue until a node, $[x, f(x)]$, is reached such that $x \sqcap A \neq x \sqcap B$.[3] The nature of Algorithm 1 guarantees that either $x \sqcap A \sqsubseteq x \sqcap B$ or $x \sqcap B \sqsubseteq x \sqcap A$. Without loss of generality, assume that $x \sqcap A \sqsubseteq x \sqcap B$. The task is to show that processing A before B leads to the same itemset tree as processing B before A. This means that, in Figure 9, state 3 is reached from state 2 by processing B or from state 4 by processing A.

1. Suppose that A is processed first (state 2). The assumption $x \sqcap A \sqsubseteq x \sqcap B$ implies that $x \sqcap A \sqsubseteq B$ because $x \sqcap B \sqsubseteq B$. Therefore, B will become the child of $x \sqcap A$. Suppose Algorithm 1 performs action 5 (for the other actions, the proof is analogous), creating a new node, $[x \sqcap B, f(x \sqcap B)]$, and appending $[x, f(x)]$ and $[B, f(B)]$ to this node as its children. The result is state 3 in Figure 9.

[3] If no such node is reached then, again, the case is trivial.

2. Suppose that B was processed first (state 4). The assumption $x \sqcap A \sqsubseteq x \sqcap B$ implies that the common ancestor of A and B is $x \sqcap B \sqcap A = x \sqcap A$. This node then has two children: one with itemset A and the other with itemset $x \sqcap B$. Again, the result is state 3 in Figure 9. □

Theorem 5.1 (order independence) *Regardless of the order in which the market baskets are processed, Algorithm 1 always creates the same itemset tree.*

Proof.
Lemma 5.1 established that $T_1 = T_2$ when a single pair of neighboring baskets is swapped. Since any permutation of the set of market baskets can be obtained after a finite number of such swaps, we conclude that Algorithm 1 generates the same itemset tree no matter how the market baskets are ordered. □

Another question is whether the original set of market baskets can be re-created from the obtained itemset tree. This property is important because, together with Theorem 5.1, it implies the existence of a one-to-one mapping between the space of market baskets and the space of itemset trees.

Theorem 5.2 *If the itemset tree has been built using Algorithm 1, then the original database can be re-created from this tree.*

Proof (sketch).
It is easy to see, based on Lemma 2.3, that Algorithm 1 guarantees that the frequency associated with an itemset-tree node is equal to the sum of the frequencies associated with this node's children plus the number of market baskets identical with the itemset in this node. The original database is re-created in two steps. First, for any leaf, $[c, f(c)]$, the frequency $f(c)$ says how many times c appeared in the database. Second, for any internal node, $[c, f(c)]$, such that $f(c) > f_{SUB}$, where f_{SUB} is the

Table 6: Algorithm 6 re-creates the original database, D, from an itemset tree, T, created by Algorithm 1

Recreate(D, T). At the beginning, let $D := \emptyset$.

For each node in T:

1. **If** the node, $[c, f(c)]$, is a leaf, **then** create $n = f(c)$ identical market baskets consisting of the itemset c, and add them to D.
2. **If** the node, $[c, f(c)]$, is an internal node, **then** denote by f_{SUB} the sum of the frequencies of all its children. **If** $f(c) > f_{SUB}$, **then** create $n = f(c) - f_{SUB}$ market baskets containing c and add them to D.

---

sum of the frequencies of c's children, the number $f(c) - f_{SUB}$ says how many times c appeared in the database. These considerations underly Algorithm 6 that re-creates the database from the itemset tree (see Table 6). □

Theorem 5.3 (uniqueness) *There is a one-to-one mapping between the space of market baskets and the space of itemset trees constructed by Algorithm 1.*

Proof.
The proof follows immediately from Theorem 5.1 and Theorem 5.2. □

6 Discussion and Ideas for Future Research

The paper suggested a mechanism for on-line association mining in market-basket type of databases. The idea is to convert the database into an *itemset tree* that organizes the data in a manner that facilitates access to the stored information.

Faster detection of itemsets and association rules then comes at the price of modest additional memory requirements. However, these requirements are garanteed to be less than twice the size of the original database.

Our algorithm builds the itemset tree in the course of a single pass through the database. The approach rests on the premise that each of the items encountered in the database is identified with a distinct integer and that the items in a market basket (or in an itemset) can thus be ordered. The exact way this numbering is carried out might affect the speed of the tree-generating algorithm as well as the time needed to process a query. A detailed study of the related performance issues is a topic for future research.

The paper describes algorithms capable of answering three essential queries. In all of them, the user submits a set of items. In the first query, the system returns the support of this itemset. In the second query, the system returns all itemsets that contain the submitted items and have support at least equal some user-set minimum. In the third query, the system returns all rules that have the submitted items an antecedent and satisfy the requirement of a user-set minimum confidence.

This research draws the attention of the research community to a direction we believe is fruitful. Indeed, association-mining methods taking advantage of tree data structures have been suggested also by some other authors [1, 2, 19]. A detailed study comparing these techniques with our itemset trees cannot be made here due to page limit and will be addressed in a follow-up paper. As for other cost-efficient approaches, we believe our approach offers some advantages. For instance, the dynamic datamining suggested by [10] does not seem to guarantee the completeness of the discovered itemsets and [29, 7], while demonstrating the ability of cost-efficient on-line association mining do not focus on targeted queries.

Interestingly, our technique can be understood as an example of unsupervised machine learning [15]. A popular goal in unsupervised machine learning is to develop a data structure that would maximize the predictability of some features (items, in

our case) based on the presence of other features. In comparison to previous work, an important advantage of itemset trees is that our algorithm creates an itemset tree that is *order independent:* the same tree is created irrespective of the order in which the market baskets are presented.

Some extensions are necessary if the method is to be applied to realistic domains. For one thing, real-world items tend to be naturally grouped in higher-level abstraction categories (for instance, `Sprite` and `Coke` are subsumed by `carbonated soft drinks` that are, in turn, subsumed by `beverages`). Association-mining algorithms from the Apriori family have been adapted to reflect such different levels of abstraction [25, 18] and similar mechanisms should therefore be introduced also in the itemset-tree framework. Another strand of research should focus on the fact that many realistic domains vary in time—for instance, each season (spring, summer, fall, and winter) can be marked by somewhat different buying patterns. The itemset-tree framework should perhaps be extended so as to make possible the discovery of time-dependent itemsets. One possibility is to employ the FLORA paradigm suggested a few years ago in the machine learning community [27, 28].

Acknowledgments

An early part of the research was carried out when M. Kubat, A. Hafez, and J. Lekkala were with the University of Louisiana. The work was supported by the grant NSF/LEQSF-SI-JFAP-06 from the Louisiana Board of Regents and NSF; by the grant LEQSF(1999-02)-RD-A-53 from the Louisiana Board of Regents; and by DE-FG02-97ER1220 from the U.S. Department of Energy.

References

[1] Agarwal, R., Aggarwal, C., and Prassad, V.V.V. (1999). Depth-first Generation of Large Itemsets of Association Rules. IBM Tech. Report RC21538, July 1999

[2] Agarwal, R., Aggarwal, C., and Prassad, V.V.V. (2000). A Tree Projection Agorithm for Generation of Frequent Itemsets, *Journal of Parallel and Distributed Computing*

[3] Agrawal, R., Imielinski, T., and Swami, A. (1993). Mining Association Rules between Sets of Items in Large Databases. *Proceedings of ACM SIGMOD*, pp. 207–216

[4] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A.I. (1996). Fast Discovery of Association Rules. In Fayyad et al., *Advances in Knowledge Discovery and Data Mining*, pp. 307–328

[5] Agrawal, R. and Shafer, J.C. (1996). Parallel Mining of Association Rules. *IEEE Transactions on Knowledge and Data Engineering, 8,* December, pp. 962–969

[6] Agrawal, R. and Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Databases. *Proceedings of the 20th International Conference on Very Large Databases*, pp. 478–499

[7] Aggarwal, C.C. and Yu, P.S. (2001). A New Approach to Online Generation of Association Rules. *IEEE Transactions on Knowledge and Data Engineering, 13,* pp 527–540

[8] Bayardo, R.J. and Agrawal, R. (1999). Mining the Most Interesting Rules. *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 145–154), San Diego, California

[9] Brin, S., Motwani, R., Ullman, J.D, and Tsur, S. (1997). Dynamic Itemset Counting and Implication Rules for Market Basket Data. *SIGMOD Record*, 26(2), p. 255

[10] Brin, S. and Page, L. (1999). Dynamic Data Mining: Exploring Rule Spaces by Sampling. Manuscript available from `http://www-db.stanford.edu/~sergey`

[11] Chen, M.-S., Park, J.S., and Yu, P.S. (1996). Data Mining for Path Traversal Patterns in Web Environments. *Proceedings of the 16th International Conference on Distributed Computing Systems* pp. 385–392

[12] Cheung, D.W., Ng, V.T., Fu, A.W., and Fu, W. (1996). Efficient Mining of Association Rules in Distributed Databases. *IEEE Transactions on Knowledge and Data Engineering, 8,* 911–922

[13] Cheung, D.W., Han, J., Ng, V., and Wong, C.Y. (1996). Maintenance of Discovered Association Rules in Large Databases. *Proceedings of the International Conference on Data Engineering*, New Orleans, LA

[14] Fisher, D.H., Pazzani, M., and Langley, P. (1991). *Concept Formation: Knowledge and Experience in Unsupervised Learning*, Morgan Kaufman

[15] Gennari, J. Langley, P. and Fisher, D. (1989). Models of Incremental Concept Formation. *Artificial Intelligence,* 40, 11–62

[16] Hafez A., Deogun, J. S., and Raghavan, V. V. (1999). Item-set tree: A data structure for data mining. *Proceedings of the Conference on Data Warehousing and KDD (DaWaK-99)*, Florence, Italy, pages 183–192, Springer

[17] Han, J. (1998). Towards On-Line Analytical Mining in Large Databases. *SIGMOD Record, 27(1)*

[18] Han, J. and Fu, Y. (1999). Mining Multiple-Level Association Rules in Large Databases. *IEEE Transactions on Knowledge Engineering,* 11, 798–805

[19] Han, J., Pei, J., and Yin, Y. (2000). Mining Frequent Patterns without Candidate Generation, *SIGMOD'00*

[20] Hidber, C. (1999). Online Association Rule Mining. *SIGMOD Conference*

[21] Liu, B., Hsu, W., and Ma, Y. (1999). Pruning and Summarizing the Discovered Associations. *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 125–134), San Diego, California

[22] Nag, B., Deshpande, P.M., and DeWitt, D. (1999). Using a Knowledge Cache for Interactive Discovery of Association Rules. *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 244–253), San Diego, California

[23] Park, J.-S., Chen, M.S., and Yu, P.S. (1995). An Effective Hash Based Algorithm for Mining Association Rules. *Proceedings of the ACM SIGMOD*, pp. 175–186

[24] Raghavan, V.V. and Hafez, A. (2000). Dynamic Data Mining. *Proceedings of the 13th International Conference on Industrial and Engineering Applications of Artificial Intelligence* (pp. 220–229), New Orleans, LA

[25] Srikant, R. and Agrawal, R. (1995). Mining Generalized Association Rules. *Proceedings of the 21th International Conference on Very Large Databases*, pp. 407–419

[26] Toivonen, H. (1996). Sampling Large Databases for Association Mining. *Proceedings of the 22nd International Conference on Very Large Databases* (pp. 134–145), Mumbay, India

[27] Widmer, G. and Kubat, M. (1996). Learning in the Presence of Concept Drift and Hidden Contexts. *Machine Learning 23*, 69–101.

[28] Widmer, G. and Kubat, M., eds. (1998) *Machine Learning, 32*, No.2., Special Issue on Concept Drift and Learning in Context-Sensitive Domains.

[29] Zaki, M. (2000). Generating Non-Redundant Association rules. *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 34–43), Boston, MA