

A Framework for Incremental Generation of Frequent Closed Itemsets

Petko Valtchev, Rokia Missaoui, Robert Godin

Département d'Informatique, UQAM, C.P. 8888, succ. "Centre Ville",
Montréal, Québec, Canada, H3C 3P8

Abstract

Concept lattices provide a theoretical framework for the efficient resolution of the association rule problem. The paper describes an extension to the underlying approach as a contribution to the issue of incremental data mining. In particular, we propose an incremental algorithm for mining frequent closed itemsets (*FCIs*) that is based on our most recent work on lattice construction. Unlike other *FCI*-based techniques, our method avoids the whole reconstruction of the *FCI* set whenever new transactions are added to the database and/or when the minimal support is changed. The paper presents the basic algorithm and an implementation by means of a trie data structure as well as a set of results from a preliminary study of the method's practical performances.

Key Words: Frequent itemsets, association rules, incremental methods, formal concept analysis, Galois lattices.

1 Introduction

Association rule mining within a transaction database [2] is a classical data mining topic, whereby the most challenging problem is the detection of frequently occurring patterns in the transaction sets (*frequent itemsets*) [1, 4, 11].

A major difficulty with association rules is the prohibitive number of frequent itemsets (and hence association rules) that results from even a reasonably large dataset. The frequent *closed* itemsets (*FCIs*) research topic [21, 13, 14] constitutes a promising approach to the problem of reducing the number of the reported association rules. Yet another difficulty arises with dynamic databases where the transaction set is frequently updated. Although the necessity of processing volatile data in an incremental manner have been repeatedly emphasized in the general data mining literature (see for example [9]), a few incremental algorithms for association rule generation (and hence frequent itemset detection) have been reported so far [5, 6, 15]. The conclusions drawn from this studies poited at an additional storage demand due to the impossibility to prune some of the infrequent itemsets in run time.

Our own approach for incremental *FI* generation is based on the intuitive idea that *FCIs* may be the right answer to the problem of the storage overhead. Therefore, we studied the potential of Galois (concept) lattices and formal concept analysis as a (formal) framework for the resolution of the problem. In this paper, we present a straightforward adaptation of our recent work on incremental lattice construction to the resolution of the rule generation problem and clarify both the forces and the limitations of the approach.

The paper starts with a short recall on association rule mining problem (Section 2) followed by a brief summary of relevant results from Galois lattice theory and algorithmics (Section 3). The outline of our approach is given in Section 4 whereas an efficient implementation based on a trie structure is presented in Section 5. Section 6 discusses preliminary results about the practical performances of our method and Section 7 provides a short survey of related work.

2 Association rule mining problem

Let $I = \{i_1, i_2, \dots, i_m\}$ be a set of m distinct items. A transaction T contains a set of items in I , and has an associated unique identifier called *TID*. A subset X of I where $k = |X|$ is referred to as a k -itemset (or simply an itemset), and k is called the length of X . A transaction database (*TDB*), say \mathcal{D} , is a set of transactions. The fraction of transactions in \mathcal{D} that present an itemset X is called the support of X and is denoted $supp(X)$. For example, the support of efh in Table 1 is 33%¹. Thus, an itemset is frequent (or large) when the ratio $supp(X)$ reaches at least a user-specified minimum threshold called *minsupp*.

As a running example, let us consider Table 1 which shows a supermarket database with a sample set of transactions $\mathcal{D} = \{1, \dots, 9\}$ involving items from the set $I = \{a, \dots, h\}$. The itemsets whose support is higher than 30% of $|\mathcal{D}|$ are given on the right of Table 1.

An association rule is an implication of the form $X \Rightarrow Y$, where X and Y are subsets of I , and $X \cap Y = \emptyset$ (e.g., $e \Rightarrow h$). The support of a rule $X \Rightarrow Y$ is defined as $supp(X \cup Y)$ while its confidence is computed as the ratio $supp(X \cup Y)/supp(X)$. For example, the support and confidence of $e \Rightarrow h$ are 33% and 75% respectively.

Given a database of transactions, the problem of mining association rules consists in generating all association rules that have certain user-specified minimum support and confidence (called *minconf*). This problem can be split into two steps:

- Detecting all frequent (large) itemsets (*FIs*) (i.e., itemsets that occur in the database with a support $\geq minsupp$)

¹In the rest of the paper, we shall use the number of the transactions supporting X instead of the fraction.

Trans.	Items
1	a, b, c, d, e, f, g, h
2	a, b, c, e, f
3	c, d, f, g, h
4	e, f, g, h
5	g
6	e, f, h
7	a, b, c, d
8	b, c, d
9	d

Itemset	Supp.	Itemset	Supp.	Itemset	Supp.
a	3	b	4	c	5
d	5	e	4	f	5
g	4	h	4		
ab	3	ac	3	bc	4
bd	3	cd	4	cf	3
ef	4	eh	3	fg	3
fh	4	gh	3		
abc	3	bcd	3	efh	3
fgh	3				

Table 1: **Left:** A sample transaction database. **Right:** The itemsets of support greater than 30%.

- Generating association rules from large itemsets (i.e., rules whose confidence $\geq \text{minconf}$)

The second step is relatively straightforward. However, the first step presents a great challenge because the set of frequent itemsets may grow exponentially with $|I|$.

Since the most time consuming operation in association rule generation is the computation of frequent itemsets, some recent studies have proposed a search space pruning based on the computation of frequent *closed* itemsets only, without any loss of information. In particular, approaches inspired by *Galois lattices* [3] have been suggested to that end [21, 13]. Thus, only a subset of *FIs* is produced and stored, which is made up of the *frequent closed itemsets (FCIs)*. An itemset X is a closed itemset if adding an arbitrary item i from $I - X$ to X results in an itemset whose support is lower than the support of X (see next section for a formal definition):

$$\forall i \in I - X, \text{supp}(X \cup \{i\}) < \text{supp}(X).$$

The following table provides the set of all *CI*s, both frequent (more than 30%) and infrequent ones, relative to the TDB of the previous example (see Table 1).

Set of CI	Closed itemsets
<i>FCI</i>	c, d, g, f, bc, cd, cf, ef, fh, abc, bcd, efh, fgh
<i>CI - FCI</i>	abcd, abcef, cd fgh, e fgh, abcdefgh

The key property in the *CI* framework states that any itemset has the same support as its closure, and hence is as frequent as its closure. For example, the closure of the itemset b is bc and both sets have a support of 4.

Previous work [9, 13] has shown that *CI*s and *FCIs* may be used in the generation of all *FIs* and rules, whereby there is no need to further access the *TDB*. Another important aspect of the rule generation problem is the enormous number of rules that can be generated even for high support and confidence thresholds. Producing minimal covers, or basis, for the entire rule sets is more useful from the user point of view. Again, previous work has shown that such minimal covers can be generated directly from the set of *FCIs* [16] or the lattice of *CI*s [21] (see the next section). Furthermore, the *CI*s lattice structure provides a context for the efficient generation of rules limited to any given frequent item subset [9].

The possibility of incrementally constructing the *FI* set is a highly sought feature within a dynamic database where new transactions may be added at any time. To motivate our study of the algorithmic problems which arise with dynamic data, let us consider the following example. Assume that the *initial TDB*, \mathcal{D}^- , includes only transactions $\{1, 2, 4 \dots, 9\}$ while the *increment* is made up of transaction #3. The following table provides the sets of *CI* for both the initial *TDB* and the increment. The augmented *TDB*, \mathcal{D} , is the union of \mathcal{D}^- and the increment.

Set of CI	Closed itemsets
CI	d, g, bc, ef, abc, bcd, efh, abcd, abcef, efgh, abcdefgh
$Increment$	c, f, cd, cf, fh, fgh, cdfgh

While a batch algorithm would have to start the computation of the CI s in \mathcal{D} from scratch, an incremental method will use both the new transaction and the existing set of CI s from \mathcal{D}^- to compute the new CI s in $Increment$ and thus obtain the complete set of transactions from \mathcal{D} .

Just like in the general case of FI , there is clearly a room for incremental techniques which maintain efficiently the FCI set upon the insertion of new transactions. In the rest of the paper, we present an approach based on algorithms for Galois lattice construction, which, to the best of our knowledge, pioneers the work on the subject.

3 Background on Galois lattices

The following is a summary of the key results from the Galois lattice theory [3], which provide the basis of our approach towards incremental generation of frequent closed itemsets.

3.1 Basics

The domain focuses on the partially ordered structure², known under the names of *Galois lattice* [3] or *concept lattice* [20], which is induced by a binary relation R over a pair of sets O (*objects*) and A (*attributes*). For example, Figure 1 on the left shows the binary relation $\mathcal{K} = (O, A, R)$ (or *context*) drawn from the TDB of Table 1 where transactions are taken as objects, items as attributes, and oRa is to be read as “transaction o has an item a ”. Two functions, f and g , summarize the links between subsets of objects and subsets of attributes induced by R .

Definition 1. *The function f maps a set of objects into a set of common attributes, whereas g^3 is the dual for attribute sets:*

- $f : \mathcal{P}(O) \rightarrow \mathcal{P}(A)$, $f(X) = X' = \{a \in A \mid \forall o \in X, oRa\}$
- $g : \mathcal{P}(A) \rightarrow \mathcal{P}(O)$, $g(Y) = Y' = \{o \in O \mid \forall a \in Y, oRa\}$

For example, w.r.t. the table \mathcal{K} in Figure 1, $f(14) = fgh$ and $g(abc) = 127^4$. Furthermore, the compound operators $g \circ f(X)$ and $f \circ g(Y)$ (denoted by $''$) are *closure* operators over $\mathcal{P}(O)$ and $\mathcal{P}(A)$ respectively. This means, in particular, that $Z \subseteq Z''$ and $(Z'')'' = Z''$ for any $Z \in \mathcal{P}(A)$ or $Z \in \mathcal{P}(O)$. Thus, each of the $''$ operators induces a family of *closed* subsets, further denoted $\mathcal{C}_{\mathcal{K}}^a$ (from *attributes*) and $\mathcal{C}_{\mathcal{K}}^o$ (from *objects*) respectively. With the example of Figure 1, the attribute sets in $\mathcal{C}_{\mathcal{K}}^a$, represents the CI s in the TDB \mathcal{D}^- presented in the previous section.

A key result of the domain states that, when ordered with set inclusion, both $\mathcal{C}_{\mathcal{K}}^o$ and $\mathcal{C}_{\mathcal{K}}^a$ form complete lattices which are sub-lattices of $\mathcal{P}(O)$ and $\mathcal{P}(A)$ respectively. Moreover, f and g constitute bijective mappings between $\mathcal{C}_{\mathcal{K}}^o$ and $\mathcal{C}_{\mathcal{K}}^a$, and isomorphisms between the corresponding lattices. This allows the pairs of mutually corresponding closed subsets to be organized in a unique structure.

Definition 2. *A **closed pair** or **concept** is a pair of sets (X, Y) where $X \in \mathcal{P}(O)$, $Y \in \mathcal{P}(A)$, $X = Y'$ and $Y = X'$. X is called the **extent** and Y the **intent** of the concept.*

For example, $(134, fgh)$ is a closed pair, but $(16, efh)$ is not. Within the CI mining framework, the closed pairs are useful as they contain both a closed itemset Y and the (closed) set X of all transactions which share exactly Y , i.e., the supporting transaction set.

²An excellent introduction to partial orders and lattices may be found in [7].

³Hereafter, both f and g are expressed by $'$.

⁴We use a separator-free form for sets, e.g., 127 stands for $\{1, 2, 7\}$, and ab for $\{a, b\}$.

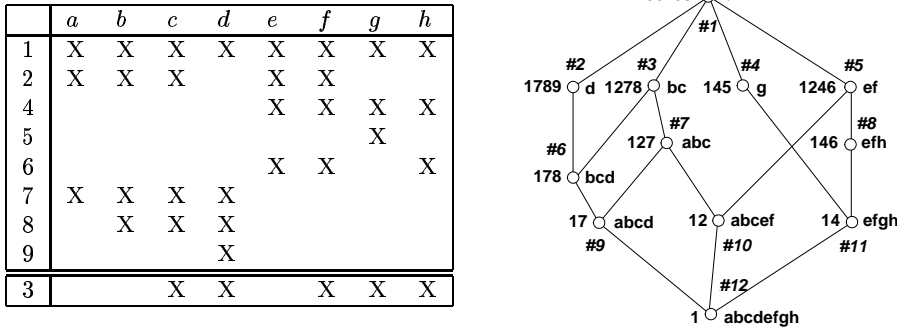


Figure 1: **Left:** Binary table $\mathcal{K}^- = (O = \{1, 2, 4, \dots, 9\}, A = \{a, b, \dots, h\}, R)$ and the object 3. **Right:** The Hasse diagram of the Galois lattice derived from \mathcal{K}^- .

Furthermore, the set $\mathcal{C}_{\mathcal{K}}$ of all closed pairs/concepts of $\mathcal{K} = (O, A, I)$ is partially ordered by intent/extent inclusion:

$$(X_1, Y_1) \leq_{\mathcal{K}} (X_2, Y_2) \Leftrightarrow X_1 \subseteq X_2, Y_2 \subseteq Y_1.$$

Theorem 1. *The partial order $\mathcal{L} = \langle \mathcal{C}_{\mathcal{K}}, \leq_{\mathcal{K}} \rangle$ is a complete lattice, called **Galois** or **concept lattice**, with joins and meets defined as follows:*

- $\bigvee_{i=1}^k (X_i, Y_i) = ((\bigcup_{i=1}^k X_i)'' , \bigcap_{i=1}^k Y_i)$,
- $\bigwedge_{i=1}^k (X_i, Y_i) = (\bigcap_{i=1}^k X_i, (\bigcup_{i=1}^k Y_i)'')$.

The Hasse diagram of the lattice \mathcal{L}^- drawn from $\mathcal{K}^- = (\{1, 2, 4, \dots, 9\}, A, R)$ is shown on the right side of Figure 1 where itemsets and transaction sets are drawn on both sides of a node representing a closed pair. For example, the join and the meet of the closed pairs $c_1 = (178, bcd)$ and $c_2 = (127, abc)$ are $(1278, bc)$ and $(17, abcd)$ respectively.

The Galois lattice provides a hierarchical organization of all closed pairs which may be used to speed-up their computation and subsequent retrieval. It is particularly useful when the set of closed pairs is to be generated incrementally, a problem we discuss in the next paragraph.

3.2 Incremental lattice update

Incremental methods construct the lattice \mathcal{L} starting from $\mathcal{L}_0 = \langle \{(\emptyset, A)\}, \emptyset \rangle$ and gradually incorporating a new object o_i into the lattice \mathcal{L}_{i-1} which corresponds to a table $\mathcal{K}_{i-1} = (\{o_1, \dots, o_{i-1}\}, A, I)$. Each incorporation involves a set of structural updates [17].

3.2.1 Principles of the incremental approach

The basic approach initially described in [10] and then improved in [18], follows a fundamental property of the *Galois connection* established by f and g on $(\mathcal{P}(O), \mathcal{P}(A))$: both families of closed subsets are themselves closed under set intersection [3]. Thus, the integration of a new object/transaction is mainly aimed at the insertion into \mathcal{L}_{i-1} of all concepts (further called *new concepts*) whose intent does not correspond to the intent of an existing concept, and is the intersection of $\{o_i\}'$ with the intent of an existing concept. Hence, three groups of concepts in \mathcal{L}_{i-1} are distinguished: *generator* concepts (denoted $\mathbf{G}(o)$) which give rise to new concepts and help compute the respective new intents and extents; *old* concepts (denoted $\mathbf{U}(o)$) which remain completely unchanged; and *modified* concepts

(labeled $\mathbf{M}(o)$) which evolve by integrating o_i into their extents while their intents remain stable. The delimitation of the three sets together with the creation of the new concepts, and their subsequent integration into the existing lattice structure constitutes the main part of the algorithm’s task.

```

1: procedure ADD-OBJECT(In:  $\mathcal{L}$  a lattice,  $o$  an object)
2:
3: SORT( $\mathcal{L}$ )      {in descending order}
4: for all  $\bar{c}$  in  $\mathcal{L}$  do
5:   if  $\text{Intent}(\bar{c}) \subseteq \{o\}'$  then
6:     ADD( $\text{Extent}(\bar{c}), o$ )     $\{(\bar{c}) \text{ is a modified concept}\}$ 
7:   else
8:      $\text{Int} \leftarrow \text{Intent}(\bar{c}) \cap \{o\}'$      $\{(\bar{c}) \text{ is an old concept}\}$ 
9:     if not  $(\text{Int}', \text{Int}) \in \mathcal{L}$  then
10:       $c \leftarrow \text{NEW-CONCEPT}(\text{Extent}(\bar{c}) \cup \{o\}, \text{Int})$      $\{(\bar{c}) \text{ is a generator}\}$ 
11:      UPDATE-ORDER( $c, \bar{c}$ ) ; ADD( $\mathcal{L}, c$ )

```

Algorithm 1: Update of a Galois (concept) lattice upon an insertion of a new object.

3.2.2 Description of the algorithm

In the sequel, we consider the subset of the algorithm described in [10] which deals with the recognition of the above three concept sets and the creation of new concepts only. Details about the lattice order updates (primitive UPDATE-ORDER) are skipped since they are irrelevant to our purposes. Thus, the concepts are first sorted in increasing order with respect to the corresponding intent sizes⁵ (line 3) and then each of them is examined in order to identify its actual category (lines 4–11). Modified concepts \bar{c} are those whose intent $\text{Intent}(\bar{c})$ is included in the description of the new object o , i.e., the set of attributes $\{o\}'$ (line 6). The remaining concepts are potentially old unless the intersection between the intent $\text{Intent}(\bar{c})$ and $\{o\}'$ represents a completely new intent in \mathcal{L} in which case \bar{c} is a generator and a new concept c is created. A property which remains implicit in the code states that a generator is the *maximum* of the set of concepts which generate a new intent by the intersection of their intent with $\{o\}'$. It is noteworthy that the extent of the new concept is just the extent of its generator, $\text{Extent}(\bar{c})$, augmented by the new object o , a fact we shall use in computing the support for *CI*s.

As an illustration, consider the insertion of object $o = 3$ into the lattice \mathcal{L} induced by the object set $\{12456789\}$ which is drawn on the right of Figure 1. Following Algorithm 1, the three categories of concepts are $\mathbf{U}(o) = \{c_{\#7}, c_{\#9}\}$, $\mathbf{M}(o) = \{c_{\#1}, c_{\#2}, c_{\#4}\}$, and $\mathbf{G}(o) = \{c_{\#3}, c_{\#5}, c_{\#6}, c_{\#8}, c_{\#10}, c_{\#11}, c_{\#12}\}$. The new concepts (identified by the *CI*s) are: $\{c, f, cd, cf, fh, fgh, cdfgh\}$; their complete integration within the Galois lattice may be observed in Figure 2 which shows the result of the whole operation once object #3 is inserted.

4 Incremental Generation of Frequent Closed Itemsets

A method for computing the *CI* family may be drawn from Algorithm 1 by focusing on relevant aspects of the concepts, as discussed in the following paragraphs. Our approach has been named *GALICIA* (for GALois Lattice-based Incremental Closed Itemset Approach).

⁵The result a (decreasing) linear extension of the lattice order.

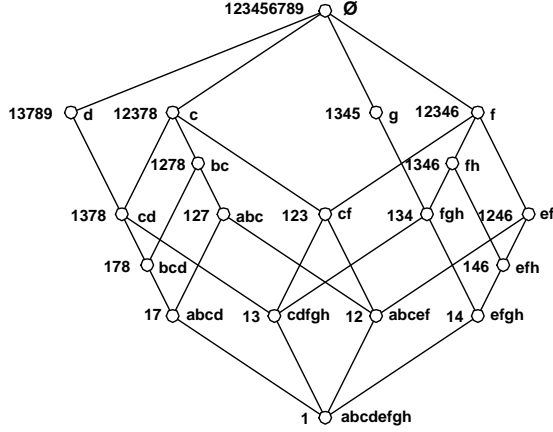


Figure 2: The Hasse diagram of the concept (Galois) lattice derived from \mathcal{K} .

4.1 Principles of the approach

Our aim is to construct $\mathcal{C}_{\mathcal{D}}^a$ only by looking at the new transaction, T_n and the *current* family of *CI*s, $\mathcal{C}_{\mathcal{D}^-}^a$ ⁶. The following observations underlie our approach.

First, for each transaction T , its itemset I_T is a *CI*. Then, since the family $\mathcal{C}_{\mathcal{D}^-}^a$ is closed under intersection, its update upon the addition of T_n amounts to computing all the intersections of existing *CI* with I_{T_n} , which are not already present in it⁷. The set of resulting *CI*s, say $\delta\mathcal{C}^a$, is split in two parts: already existing *CI*s and *new CI*s. Any intersection may be generated more than once, e.g., c is generated by both bc and abc on Figure 1. However, there is always a unique minimal⁸ *CI*, further called the *minimal generator*, which helps generate it (bc for the case of the new *CI* c). It is noteworthy that the minimal generator of an already existing closed itemset X is X itself, whereas new *CI*s clearly diverge from their minimal generators. Hence, existing intersections are compared to *modified* concepts in \mathcal{L}^- and new intersections to new concepts. Furthermore, a minimal generator *CI* corresponds to the intent of a (maximal⁹) *generator* concept.

The absolute support of *CI*s in $\mathcal{C}_{\mathcal{D}}^a$ are obtained from supports in \mathcal{D}^- in the following manner: for all *CI*s in $\delta\mathcal{C}^a$, their support in \mathcal{D} is the support of their minimal generator plus one, while the supports of the remaining *CI*s stay unchanged. This means, in particular, that all the supports of *CI*s corresponding to existing intersections have to be increased by one in $\mathcal{C}_{\mathcal{D}}^a$.

It is important to note that since the approach is incremental, there is a need to keep the whole set of *CI*s, including those which are not frequent. This is due to the fact that after one or many insertions of new transactions, some *CI*s may become frequent while some *FCI*s may become infrequent closed itemsets. Moreover, discarding some *CI*s acting as intents of *generator* concepts will lead to disregarding their corresponding new concepts. As an illustration, let's assume that Transactions 10 and 11 are added to \mathcal{D} (see Table 1) with itemsets $abcd$ and $abcde$ respectively. In that case, some *FCI*s such as cf , efh , fgh will become infrequent *CI*s (27%) while $abcd$ will turn frequent itemset (36%). If ever $abcef$ is discarded during the update process simply because it is an infrequent (22%) *CI*, then the new *CI* $abce$ will never be generated.

⁶We assume $\mathcal{D} = \mathcal{D}^- \cup \{T_n\}$.

⁷All such intersections are closed in \mathcal{D} .

⁸With respect to set-theoretic inclusion.

⁹Here maximal is taken with respect to lattice order which is the inverse of intent inclusion.

4.2 Description of the algorithm

Algorithm 2 hereafter preserves the main control structure of its lattice counterpart: each *CI* of the current collection (*FamilyCI*) is examined to establish its specific category (*modified*, *old* or *minimal generator* of a new *CI*). Like in the lattice procedure, modified *CI*s simply get their support increased (line 9) and old ones remain unchanged (line 11). Processing generators diverges slightly from the lattice version as no order is supposed in *FamilyCI*¹⁰. Actually, each new *CI* is stored together with the maximal support already reached for that *CI*. Thus, each time the *CI* is generated (lines 13–17), the support is tentatively updated. Furthermore, the storage of new *CI*s is organized separately (collection *NewCI*) so that unnecessary tests can be avoided. This computation yields the correct support at the

```

1: procedure UPDATE-CLOSED(In:  $T_n$  a transaction, FamilyCI a collection of itemsets)
2:
3: Local : NewCI a collection of itemsets
4:
5: NewCI  $\leftarrow \emptyset$  ;  $I_n \leftarrow T_n.itemset$ 
6: for all  $e$  in FamilyCI do
7:    $I_e \leftarrow e.itemset$ 
8:   if  $I_e \subseteq I_n$  then
9:      $e.support++$     {e is modified}
10:  else
11:     $Y \leftarrow I_e \cap I_n$  ;  $e_Y \leftarrow lookup(FamilyCI, Y)$     {e is old or potential generator}
12:    if  $e_Y = NULL$  then
13:       $e_Y \leftarrow lookup(NewCI, Y)$     {e is a potential generator}
14:      if  $e_Y = NULL$  then
15:         $node \leftarrow new-node(Y, e.support + 1)$  ;  $NewCI \leftarrow NewCI \cup \{node\}$ 
16:      else
17:         $e_Y.support \leftarrow \max(e.support + 1, e_Y.support)$ 
18: FamilyCI  $\leftarrow FamilyCI \cup NewCI$ 

```

Algorithm 2: Update of the closed itemset family upon a new transaction arrival.

end of the current *CI* family traversal since minimal generators are *CI*s with maximal support among all *CI*s generating a new *CI*. This fact is strongly reinforced by an implementation proposal which utilizes trie structures in order to reduce redundancy in both the storage and the update of the *CI* family.

5 Trie-based method

In what follows, we describe GALICIA-T, an improved version of GALICIA based on tries.

5.1 Trie basics

The *trie* (from *retrieval*) data structure [12] provides a good trade-off between storage requirements and manipulation cost. It is currently used to store sets of words over a finite alphabet. In its basic form, a trie is a tree whereby letters from the alphabet are assigned to edges, so that each word corresponds to a unique path in the tree (see Figure 3). Nodes carry minimal information: those corresponding to the end of a word, further called *terminal* nodes, are distinguished from the rest, called *inner* nodes. As an illustration, see the trie corresponding to the *CI*s over \mathcal{K}^- which is given

¹⁰A sorted collection could have been used instead, but this would offer only a modest reduction of the support computation overhead, whereas the main complexity source, i.e., the intersection calculations, remains untouched.

on the left part of Figure 3. Here, terminal nodes are drawn as filled circles and inner nodes as empty ones.

Tries offer a highly compact representation since all prefixes common to two or more words are represented only once in the trie. Such factorization not only reduces the storage space, but also provides for more efficient operations, e.g., search of a word or insertion of a new one into the trie. Tries where words represent sets – as in our case – provide very efficient operations which can be carried out in a time linear in the size of the alphabet, regardless of the size of the trie.

5.2 Description of the algorithm

In our framework, two tries are used to represent closed itemsets (in terminal nodes): one for the current *CI* family, *FamilyCI*, and another one for the increment set of *CI*s, *NewCI*. The `trie` type used here is basically a tree of nodes with a distinguished `root`. A `node` is a record with `item`, `terminal`, `successors`, `support` and `depth` fields. The `successors` collection is a sorted, indexed and extensible collection with primitives for lookup, order-sensitive traversal, insertion of a new member, etc. Sorted lists of items¹¹ are used to represent transactions and individual *CI*s. T_n is the new transaction with its itemset I_n , and Y_{curr} is the current intersection between a *CI* and I_n .

```

1: procedure UPDATE-CLOSED-TRIE(In:  $T_n$  a transaction)
2:
3: Global : FamilyCI a trie of itemlists ; Local : NewCI a trie of itemlists
4:
5:  $NewCI \leftarrow \text{new-trie}()$  ;  $I_n \leftarrow \text{SORT}(T_n.\text{itemset})$ 
6: TRAVERSAL-INTERSECT( $I_n$ , NULL, root(FamilyCI))
7: merge(FamilyCI, NewCI)

```

Algorithm 3: Trie-based update of the *CI*s upon a new transaction arrival.

Algorithm 3 describes the main steps of an update with a single new transaction T_n , namely the creation of the increment trie, the sorting of the T_n itemset, the traversal of the trie with the generation of the new *CI*s and finally the merge of the two tries¹².

Algorithm 4 is a recursive procedure that models the simultaneous traversal (with detection of common elements) of two sequences of items: the I_n , representing the yet unseen part of the new itemset ($T_n.\text{itemset}$) and the path of the trie starting from the root and leading to the current trie node (*node*). In general, the second sequence can be completed to a full *CI* in several manners, each of them corresponding to a suffix stored in the trie starting from the current node on.

The traversal starts with a tentative expansion of the current intersection over the current node (lines 5 – 6). Each time a terminal node is reached (line 7), the currently generated intersection (Y_{curr}) corresponds to a *CI* of $\mathcal{C}_{\mathcal{D}}^a$. In these cases, the status of the set Y_{curr} , i.e., either new *CI* or already existing in $\mathcal{C}_{\mathcal{D}-1}^a$, should be established by checking whether it is already in the basic trie *FamilyCI* (line 8). The result of the check may in turn trigger an (tentative) insertion of Y_{curr} into the *NewCI* trie, whenever it is a new *CI* (line 10), or an update of the current node support (line 12 – 13). The second case occurs when the current *CI* of the trie, i.e., the one ending by *node*, happens to be a modified element of $\mathcal{C}_{\mathcal{D}-1}^a$. Recall that modified *CI*s are exactly those included, as subsets, in $T_n.\text{itemset}$. This fact is established by comparing the length of the current intersection, $\|Y_{curr}\|$, to the depth of the current node, i.e., the length of the path from the root to *node* (line 12). Unless a termination condition is reached (end of I_n and terminal *node* - line 14), TRAVERSAL-INTERSECT is recursively called for each suffix (lines 15 – 18). In doing that, the successors of *node* are listed in a lexicographic order, so that the current itemlist I_n could be gradually reduced (lines 16 – 17).

¹¹Itemlist nodes provide `item` and `next` primitives.

¹²Due to space limitation, details of the merge operation are omitted.

```

1: procedure TRAVERSAL-INTERSECT(In:  $I_n$ ,  $Y_{curr}$  itemlists, node a trie node)
2:
3: Global : FamilyCI, NewCI tries of itemlists
4:
5: if ( $I_n \neq \text{NULL}$ ) and ( $I_n.item = node.item$ ) then
6:   add( $Y_{curr}$ ,  $I_n.item$ )
7: if node.terminal then
8:    $n \leftarrow \text{lookup}(\textit{FamilyCI}, Y_{curr})$ 
9:   if  $n = \text{NULL}$  then
10:    update-insert(NewCI,  $Y_{curr}$ , node.support + 1)
11:  else
12:    if node.depth =  $\|Y_{curr}\|$  then
13:      n.support ++
14: if (not node.terminal) or ( $I_n \neq \text{NULL}$ ) then
15:   for all  $n$  in node.successors do
16:     while ( $I_n \neq \text{NULL}$ ) and ( $I_n.item < n.item$ ) do
17:        $I_n \leftarrow I_n.next$ 
18:     TRAVERSAL-INTERSECT( $I_n$ ,  $Y_{curr}$ ,  $n$ )

```

Algorithm 4: Trie-based update of the *CI*s: single node processing.

The following table illustrates the work of Algorithm 4 on two distinct branches of the trie, *abcdefgh* and *efgh*, upon the insertion of the itemlist *cdfgh*.

<i>node.item</i>	I_n	Y_{curr}	terminal	<i>supp</i>
<i>a</i>	<i>cdfgh</i>	NULL	N	-
<i>b</i>	<i>cdfgh</i>	NULL	N	-
<i>c</i>	<i>cdfgh</i>	<i>c</i>	Y	4
<i>d</i>	<i>dfgh</i>	<i>cd</i>	Y	3
...
<i>h</i>	<i>h</i>	<i>cdfgh</i>	Y	2

<i>node.item</i>	I_n	Y_{curr}	terminal	<i>supp</i>
<i>e</i>	<i>cdfgh</i>	NULL	N	-
<i>f</i>	<i>fgh</i>	<i>f</i>	Y	5
<i>g</i>	<i>gh</i>	<i>fg</i>	N	-
<i>h</i>	<i>h</i>	<i>fgh</i>	Y	3

It should be read as follows: the first column provides the item in *node*, the second is the value of I_n (available part of T_n), the third column is the intersection computed so far, and the fourth one indicates, whether *node* is terminal, i.e., whether the value of Y_{curr} represents a *CI*. The fifth column provides, whenever a terminal node is reached, the computed support.

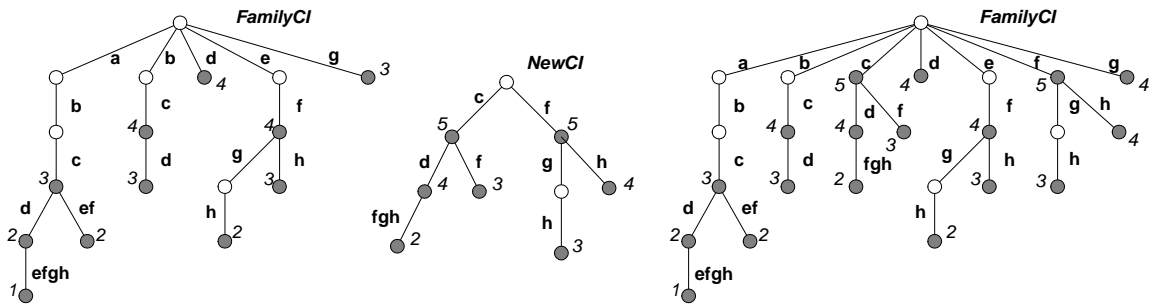


Figure 3: **Left:** The trie *FamilyCI* of the *CI*s generated from \mathcal{D}^- . **Middle:** The trie *NewCI* of the new *CI*s relative to transaction #3. **Right:** The trie *FamilyCI* after the insertion of transaction 3.

Figure 3 depicts the result of the entire trie traversal. On the left, the state of *FamilyCI* before the

insertion of transaction 3 is shown. On the right, the situation before the merge of both tries *FamilyCI* and *NewCI* is shown.

The above Algorithm 3 can be completed to a first-class procedure for mining *FCI* from a transaction database. Here, details concerning the filtering of infrequent *CI* are ignored, but the task could be easily carried out through a rough index for *CI* based on support values: once a value for the *minsupp* is provided, the filter would simply enumerate the buckets of *CI* in the index satisfying it.

6 Tests

In order to study the strengths and limits of our incremental algorithm, we have undertaken some preliminary tests in which it was compared to the algorithm CLOSET [14]. Such a choice of the reference algorithm was motivated by the features shared by both procedures:

- computation of *FCIs*,
- use of trie-like data structure for compact storage.

We were additionally motivated by the fact that Closet is one of the most efficient algorithms for *FCIs* generation.

Both algorithms were implemented in Java, whereas we used an improved version of CLOSET where the search of inclusion between a candidate *FCI* and an existing *FCI* is powered by a trie structure. The experiments (see Figure 4) were done on a Windows 2000 station (Pentium III with 512 M RAM) using various subsets of the IBM transaction database T25I10D100K.

Three groups of tests have been carried out. The first group of tests aimed at comparing the performance of the two algorithms without any concern about the incremental feature of GALICIA-T. The results of those tests (not provided here) have shown the clear advantage of CLOSET (and most probably of some other batch techniques) over our method. Only tiny values of support, i.e., when almost all the *CI*s are to be kept, tend to favour our method with a ration between the CPU time taken going up to four.

The second group of tests highlighted the overhead induced by re-running CLOSET on the whole updated database versus running GALICIA-T with the increment only. The results of CLOSET are summarized by the left-hand side diagram in Figure 4. The diagram includes several graphs each corresponding to a sample subset of T25I10D100K of a particular size. The points recorded correspond to specific supports varying from 0.1 to 0.9¹³. The right-hand side diagram in Figure 4 shows the performances of our method whereby the CPU-time is computed as the average insertion cost over packages of thousands of transactions: for example, the point corresponding to the point 2000 of the *X*-axis is the average time of the insertion of transactions 1001 to 2000. Two trends may be easily observed on those diagrams: (i) the time taken by CLOSET grows rapidly with the decrease of the support, and (ii) both the average insertion cost for GALICIA-T and the total mining cost for CLOSET are quasi-linear function of the sample size. Furthermore, the cost of CLOSET for support values lower than 0.6 is, in general, orders of magnitude greater than the time of a single insertion. This apparently trivial fact underlines the possible benefits of using an incremental approach: running CLOSET once with an augmented dataset may cost a time up to hundred times more than the time spent for inserting a single transaction with GALICIA-T. In other words, one may run, say, several hundreds of insertions with GALICIA-T while CLOSET is working on the entire dataset. Of course, this does not make our algorithm more efficient for the whole task as the total time taken remains too high. However, the question becomes a trade-off between frequency of updates and the urgency of the need for intermediate results. For example, with 20000 transactions and 0.1 support, the insertion of 1000 new transactions

¹³The interval has been chosen since the 0.9 is the highest threshold leading to the discovery of a significant number of non-trivial *CFIs* (i.e., of cardinality more than one).

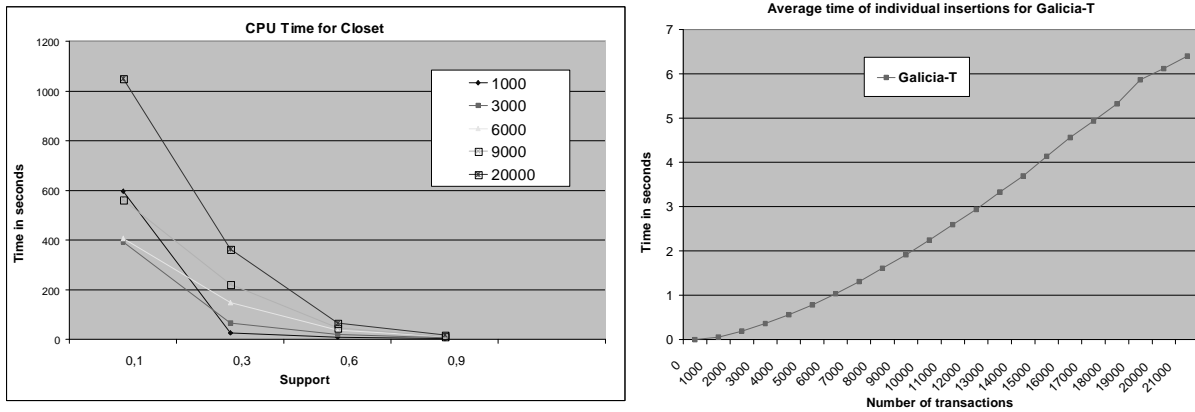


Figure 4: **Left:** CPU-time for CLOSET for an extract from T25I10D100K, variation over sample size and *min-supp*. **Right:** CPU-time for the insertion of a single object, average over 1000 transactions.

with up-to-date result after each insertion will be enough for running CLOSET 4 to 5 times, i.e., on each set of 200 to 250 transactions.

The third group helped evaluate the impact of a varying support on the overall efficiency of the two algorithms. Although the respective results are not explicitly shown, the graphs in Figure 4 may be used to provide a general idea about the situation. First, observe that for GALICIA-T, modifying the support, whether to a higher or to a lower value, amounts to a simple filtering of the *CI* family. This is also the case when the result of CLOSET is to be revised upon support threshold increases, but not with decreases which require a complete execution of the algorithm. One might argue that a single run of CLOSET with a sufficiently low support would prevent falling in this trap. However, with sparse datasets as in T25I10D100K, the target support value may lay hasardously close to numbers that make the algorithm even less efficient than the incremental procedure¹⁴.

When taken as a whole, the above results suggest that a straightforward incremental approach of the kind described here will most probably prove inefficient in purely static databases when the support is fixed. However, it is certainly more appealing in typical database applications and data mining tasks where data stores are very dynamic and the mining task is carried out in an exploratory manner. More precisely, it may be very helpful in the environments where the user may want to frequently:

- modify the support treshold of *FIs* for a given *TDB*. In such a case, there is no need to construct the set of *FCIs* from search, but it is rather a question of filtering the available *CIs*.
- process new transactions in dynamic databases and analyze the impact of such transactions on the mining result.

7 Related work

A tremendous number of approaches to the problem of association rule mining have been reported since the first publication of the *Apriori* algorithm [2]. Comparative studies are research subjects in their own (see for example [11]). Therefore, we limit our attention to methods that present one or both of the key features discussed in our paper, i.e., being incremental or computing the *FCIs*.

¹⁴With the above datasets, a value of 0.07 made CLOSET work longer than GALICIA-T.

One of the earliest work on incremental mining is due to Cheung *et al.* [5] where the *FUP* algorithm updates association rules when new transactions are added. *FUP* first stores the counts of all frequent itemsets found in a previous mining process, and then exploits these counts and the newly added transactions to generate a very small number of candidates. A more general incremental technique called *FUP₂* is proposed [6] for updating association rules when insertion, deletion, and modification of transactions occur. Both *FUP* and *FUP₂* are based on the Apriori framework (e.g., there is a candidate generation step) that exploits the previous mining output to avoid the generation of useless candidates. A recent work reported in [15] extends the limits of incremental approaches by allowing changes to the basic parameters of the mining process (e.g., *minsupp* threshold, number of transactions added at a time, etc.).

Alternative approaches to mining *CI*s from a database have been presented in [21, 13], both following the theoretical guidelines of the *Galois lattice/FCA* domain [3, 8]. However, both approaches suggest complex and expensive computations of *CI* from candidate itemsets.

Finally, some existing techniques use compact representations of the *FI* family based on trie-like structures such as *prefix-trees*, *FP-trees*, and *digital trees* (see [11] for a survey). The CLOSET algorithm [14] relies on recursive construction of *FP-trees* to build the set of *FCIs*.

Based on some key features, the following table compares a (non-exhaustive) set of approaches that were concerned either with the *incremental* generation of frequent itemsets (and sometimes the association rules too) or with the discovery of frequent *closed* itemsets. Approaches are compared with respect to criteria found in one or both of the two groups. The parameters *k*, *n* and *m* stand for the size of the largest *FI*, the number of *FI* in the result, and the number of insertions, respectively:

Method	Candidate generation	CI computation	Incremental	DB passes (nb)	multi-support environment
A-CLOSE[13]	Y	Y	N	$O(k)$	N
CHARM[21]	Y	Y	N	$O(n)$	N
FUP[6]	Y	N	Y	$O(k)$	N
DELTA[15]	N	N	Y	$O(m)$	Y
CLOSET [14]	N	Y	N	1	N
GALICIA	N	Y	Y	1	Y

Besides the incremental and multi-support features, our approach may be easily extended so that the effects of the increment on a given set of discovered *FCIs* become visible. Such a feature helps estimate the impact of some actions (e.g., new business strategies) taken between a previous mining process and the current one (i.e., the mining of the increment only) [15].

8 Discussion

We have presented a framework for mining frequent closed itemsets in an incremental manner, which enjoys solid foundations from Galois lattice and FCA theory. We also suggested a straightforward incremental algorithm with an additional valuable feature which is the low-cost response to a readjustment in the *minsupp* from the part of the user. An implementation of the algorithm in terms of tries has been studied in the paper and its performances was compared to those of a major batch algorithm.

Incrementality is an important challenge for data mining methods and our framework is a first step towards this goal. The experimental results show its potential benefits in particular for the case of small support thresholds.

The scalability of our approach is limited by the necessity of maintaining the whole set of frequent closed itemsets particularly for the case of dense data. Therefore, our next step is to address this problem by introducing the notion of *border* in order to limit the number of *FCIs* to maintain while preserving enough information for its incremental maintenance. A promising track seems to reside in

the joint application of GALICIA with another method that computes *FIs* that is both efficient and relies on *CIs*, e.g., CLOSET, A-CLOSE or CHARM. The latter could be applied as a pre-processing subroutine that extracts the *FCIs* plus the border from the known part of a dataset while leaving the subsequent maintenance of the result to GALICIA. The idea naturally generalizes to a somewhat different aspect of our lattice-based framework, i.e., the incremental integration of batches of transactions by lattice merge procedures as developed in [19]. The underlying framework offers a large choice of possible operations on results upon updates in the dataset such as add and remove of groups or individual transactions. It enables the combination of several concrete algorithms working on fragments of the dataset and may favor the distribution of the computation.

References

- [1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast Discovery of Association Rules. In U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *In Proceedings of the 20th International Conference of Very Large Databases (VLDB)*, pages 487–499, Santiago, Chile, September 1994.
- [3] M. Barbut and B. Monjardet. *Ordre et Classification: Algèbre et combinatoire*. Hachette, 1970.
- [4] R.J. Bayardo and R. Agrawal. Mining the most interesting rules. In *Proceedings of the 5th International Conference on Knowledge Discovery and Data Mining (KDD'99)*, 1999.
- [5] D. W. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique. In *Proc. 12th IEEE International Conference on Data Engineering*, New Orleans (LA), 1996.
- [6] D. W. Cheung, S. D. Lee, and B. Kao. A General Incremental Technique for Maintaining Discovered Association Rules. In *Database Systems for Advanced Applications*, pages 185–194, 1997.
- [7] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1992.
- [8] B. Ganter and R. Wille. *Formal Concept Analysis, Mathematical Foundations*. Springer-Verlag, 1999.
- [9] R. Godin and R. Missaoui. An Incremental Concept Formation Approach for Learning from Databases. *Theoretical Computer Science*, 133:378–419, 1994.
- [10] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept formation algorithms based on galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.
- [11] J. Hipp, U. Guentzer, and G. Nakhaeizadeh. Algorithms for Association Rule Mining - A General Survey and Comparison. *SIGKDD Explorations*, 2(1):58–64, 2000.
- [12] D. E. Knuth. *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison-Wesley, Reading (MA), second edition, 1998.
- [13] N. Pasquier, Y. Bastide, T. Taouil, and L. Lakhal. Efficient Mining of Association Rules Using Closed Itemset Lattices. *Information Systems*, 24(1):25–46, 1999.
- [14] J. Pei, J. Han, and R. Mao. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In *Proceedings of the ACM-SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.
- [15] V. Pudi and J. R. Haritsa. Quantifying the Utility of the Past in Mining Large Databases. *Information Systems*, 25(5):323–343, 2000.
- [16] R. Taouil, N. Pasquier, Y. Bastide, and L. Lakhal. Mining bases for association rules using closed sets. In *Proceedings of the 16th International Conference on Data Engineering (ICDE'2000)*, pages 307–??, San Diego, February 2000. IEEE Computer Society.
- [17] P. Valchev. An algorithm for minimal insertion in a type lattice. *Computational Intelligence*, 15(1):63–78, 1999.

- [18] P. Valtchev and R. Missaoui. Building concept (Galois) lattices from parts: generalizing the incremental methods. In H. Delugach and G. Stumme, editors, *Proceedings of the ICCS 2001, Satanford (CA)*, volume 2120 of *Lecture Notes in Computer Science*, pages 290–303. Springer-Verlag, 2001.
- [19] P. Valtchev, R. Missaoui, and P. Lebrun. A partition-based approach towards building Galois (concept) lattices. *to appear in Discrete Mathematics*, 2001.
- [20] R. Wille. Restructuring the lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered sets*, pages 445–470, Dordrecht-Boston, 1982. Reidel.
- [21] M.J. Zaki. Generating Non-Redundant Association Rules. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining (KDD'00)*, 2000.