Generating Frequent Itemsets Incrementally: Two Novel Approaches Based On Galois Lattice Theory

Petko Valtchev[1)], Rokia Missaoui[2)], Robert Godin[2)], Mohamed Meridji[2)]

[1)] DIRO, Université de Montréal, C.P. 6128, Succ. "Centre-Ville",
Montréal, Québec, Canada, H3C 3J7

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

Abstract

Galois (concept) lattice theory has been successfully applied to the resolution of the association rule problem in data mining. In particular, structural results about lattices have been used in the design of efficient procedures for mining the frequent patterns (itemsets) in a transaction database. As transaction databases are often dynamic, we propose a detailed study of the incremental aspects in lattice construction to support effective procedures for incremental mining of frequent closed itemsets (*FCIs*). Based on a set of descriptive results about lattice substructures involved in incremental updates, the paper presents a novel algorithm for lattice construction that only explores limited parts of a lattice for updating. Two new methods for incremental *FCI* mining are studied: the first one inherits its extensive search strategy from a classical lattice method, whereas the second one applies the new lattice construction strategy to the itemset mining context. Unlike batch techniques based on *FCIs*, both methods avoid rebuilding from scratch the *FCI* family whenever new transactions are added to the database and/or when the minimal support is changed.

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

1 Introduction

Association rule mining from a transaction database has been a very active research area since the publication of the Apriori algorithm [2]. Several improvements to the basic algorithm and many new approaches have been proposed during the last decade.

It is well-known that the most challenging and time-consuming step in association rule mining is the detection of frequently occurring patterns in the transaction sets (*frequent itemsets*) [1, 5, 14]. Such a step may generate a prohibitive number of frequent itemsets (and hence association rules) even from a reasonably large dataset. The frequent *closed* itemsets (*FCIs*) research topic [21, 22, 31, 30] constitutes a promising solution 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 has been repeatedly emphasized in the general data mining literature (see for example [11]), a few incremental algorithms for association rule generation (and hence frequent itemset detection) have been reported so far [6, 7, 9, 25, 23]. The conclusion drawn from some of these studies highlights the need for more storage space due to the impossibility to prune some of the infrequent itemsets at run time.

Our own approach to incremental *FI* generation is motivated by the belief that *FCIs* provide the key to compact rule sets and low storage requirements. Therefore, we have been investigating the potential benefits of using Galois lattice theory and formal concept analysis as a formal framework for the resolution of the *FCIs* mining problem. In this paper, we examine the links between incremental lattice construction and incremental *FI* generation. First, we establish the necessary correspondence between basic elements of both frameworks. Then, we present a way to transform a recent version of classical lattice algorithm into a *FCI*-mining procedure and discuss efficient implementation in terms of a trie structure. As the resulting approach relies on extensive exploration of the temporary *FCI* family upon each update, we investigate possible pruning strategies that limit the number of examined *FCIs*. For that reason, we provide a set of characteristic properties for the lattice substructures involved in incremental updates and embody them into a new incremental algorithm that concentrates on a subset of *relevant* lattice nodes. The new algorithm is in turn transformed into an incremental *FCI*-miner whose performances are compared to those of a recent batch procedure [22].

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 4.3 whereas an efficient implementation based on a trie structure is presented in Section 4.4. Section 5 describes an alternative approach that avoids the search of the entire set of *CI*s when a new transaction is added. Section 6 provides a short survey of related work and Section 7 discusses preliminary results about the practical performances of our method.

2 Association rule mining problem

Let $I = \{i_1, i_2, \cdots, i_m\}$ 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 contain 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 $supp(X)$ 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 $\mathcal{D} = \{1, \cdots, 9\}$ involving items from the set $I = \{a, \cdots, h\}$. The itemsets whose support is higher than 30% of $|\mathcal{D}|$ are given on the right of Table 1.

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%.

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

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

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 (*FI*s) (i.e., itemsets that occur in the database with a support $\geq$ *minsupp*),
- Generating association rules from large itemsets (i.e., rules whose confidence $\geq$ *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* [4] have been suggested to that end [30, 21]. 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):

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

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

Set of CI	**Closed itemsets**
FCI	c, d, g, f, bc, cd, cf, ef, fh, abc, bcd, efh, fgh
CI - FCI	abcd, abcef, cdfgh, efgh, 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 [11, 21] has shown that *CIs* 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* [24] or the lattice of *CIs* [30] (see the next section). Furthermore, the *CIs* lattice structure provides a context for the efficient generation of rules limited to any given frequent item subset [11].

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 $\{1, 2, 4 \cdots, 9\}$ 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*, i.e., $\mathcal{D}$, 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 *CIs* in $\mathcal{D}$ from scratch, an incremental method will use both the new transaction and the existing set of *CIs* from $\mathcal{D}^-$ to compute the new *CIs* in *Increment* and thus obtain the complete set of transactions from $\mathcal{D}$.

Just like in the general case of *FIs*, 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 [4], which provide the basis of our approach towards incremental generation of frequent closed itemsets.

3.1 The basics of ordered structures

$P = \langle G, \leq_P \rangle$ is a *partial order* (poset) over a *ground* set G and a binary relation $\leq_P$ if $\leq_P$ is reflexive, antisymmetric and transitive. For a pair of elements a, b in G, if $b \leq_P a$ we shall say that a *succeeds* (is greater than) b and, inversely, b *precedes* a. If neither $b \leq_P a$ nor $a \leq_P b$, then a and b are said to be *incomparable*. All common successors (predecessors) of a and b are called *upper* (*lower*) *bounds*. The *precedence* relation $\prec_P$ in P is the transitive reduction of $\leq_P$, i.e. $a \prec_P b$ if $a \leq_P b$ and all c such that $a \leq_P c \leq_P b$ satisfy $c = a$ or $c = b$. Given such a pair, a will be referred to as an *immediate predecessor* of b and b as an *immediate successor* of a. Usually, P is represented by its *covering graph* $Cov(P) = (G, \prec_P)$. In this graph, each element a in G is connected to both the set of its immediate predecessors and of its immediate successors, further referred to as *lower covers* (Cov^l) and *upper covers* (Cov^u) respectively. In the following, we shall visualize a partial order by its Hasse diagram, that is the line diagram of the covering graph where each element is located "below" all its successors.

A subset A of G is a chain (anti-chain) in P if all elements in A are mutually comparable (incomparable). A subset B of G is an *order ideal* (*order filter*) if $\forall a \in G, b \in B$, $a \leq_P b \Rightarrow a \in B$ ($b \leq_P a \Rightarrow a \in B$). For a given set $A \subseteq X$, the set $\downarrow_P A = \{c \in X | \exists a \in A, c \leq_P a\}$ is the smallest order ideal containing A. Dually, $\uparrow_P A = \{c \in X | \exists a \in A, a \leq_P c\}$ denotes the smallest order filter containing A. In case of a singleton A, we shall note $\downarrow_P a$ instead of $\downarrow_P \{a\}$ ($\uparrow_P a$ instead of $\uparrow_P \{a\}$). Moreover, the *order interval* $[a, b]$ is the subset of nodes obtained by the intersections of an order filter $\uparrow_P a$ and an order ideal $\downarrow_P b$. A *convex* subset of an order is a subset that includes for any pair of its members the interval they might compose. A mapping ϕ between two posets P and Q such that $\phi : P \to Q$ is said to be *order preserving* if an order relation between two elements of P entails an order relation between their respective images under ϕ in Q:

$$x \leq_P y \Rightarrow \phi(x) \leq_Q \phi(y).$$

Furthermore, ϕ is said to be an *order embedding* of P into Q if the condition is also a sufficient one:

$$x \leq_P y \Leftrightarrow \phi(x) \leq_Q \phi(y).$$

A lattice $L = \langle G, \leq_L \rangle$ is a partial order where any pair of elements a, b has a unique *greatest lower bound* (GLB) and a unique *least upper bound* (LUB). GLB and LUB define binary operators on G called respectively *join* ($a \vee_L b$) and *meet* ($a \wedge_L b$). In a complete lattice L, all the subsets A of the ground set have a GLB and a LUB. In particular, there are unique maximal (top, $\top$) and minimal (bottom, $\bot$) elements in the lattice. A structure with only one of the above operations is called semi-lattice, e.g., the existence of a unique GLB for any couple (set) of elements implies a (complete) *meet* semi-lattice structure.

3.2 Basics

The domain focuses on the partially ordered structure[2], known under the names of *Galois lattice* [4] or *concept lattice* [29], 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:*

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

For example, w.r.t. the table $\mathcal{K}$ in Figure 1, $f(14) = fgh$ and $g(abc) = 127$[4]. Furthermore, the

	a	b	c	d	e	f	g	h
1	X	X	X	X	X	X	X	X
2	X	X	X		X	X		
4					X	X	X	X
5							X	
6					X	X		X
7	X	X	X	X				
8		X	X	X				
9				X				
3			X	X		X	X	X

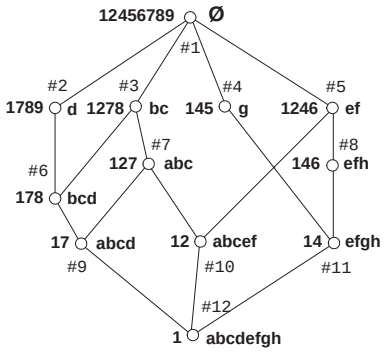

Figure 1: **Left:** Binary table $\mathcal{K}^- = (O = \{1, 2, 4, ..., 9\}, A = \{a, b, ..., h\}, R)$ and the object 3. **Right:** The Hasse diagram of the Galois lattice derived from $\mathcal{K}^-$.

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 $\mathcal{C}^a_\mathcal{K}$ (from *attributes*) and $\mathcal{C}^o_\mathcal{K}$ (from *objects*) respectively. With the example of Figure 1, the attribute set in $\mathcal{C}^a_\mathcal{K}$, 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 $\mathcal{C}^o_\mathcal{K}$ and $\mathcal{C}^a_\mathcal{K}$ form complete lattices which are sub-lattices of $\mathcal{P}(O)$ and $\mathcal{P}(A)$ respectively. Moreover, f and g constitute bijective mappings between $\mathcal{C}^o_\mathcal{K}$ and $\mathcal{C}^a_\mathcal{K}$, 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, $(178, bcd)$ 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.

[2] An excellent introduction to partial orders and lattices may be found in [8].
[3] Hereafter, both f and g are expressed by $'$.
[4] We use a separator-free form for sets, e.g., 127 stands for $\{1, 2, 7\}$, and ab for $\{a, b\}$.

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 $K^- = (\{1, 2, 4, ..., 9\}, A, R)$ 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 which is addressed by the next paragraphs.

3.3 Rules and order

As indicated earlier, association rules can be advantageously generated from *FCIs* rather than *FIs*. However, even in this case, there is still a large set of generated rules with information redundancy. It is therefore more useful and relevant to provide the user with a non redundant rule set.

For example, suppose the following rules are valid with the same support and confidence levels:

$ab \rightarrow cde$	$ab \rightarrow c$	$ab \rightarrow d$
$abc \rightarrow d$	$abcd \rightarrow e$	$abde \rightarrow c$

The first rule $ab \rightarrow cde$ is sufficient because the other ones do not give additional information. It is therefore more efficient to generate only the first one. A subset of rules S of a rule set R which preserves the information of R is called a *cover* of R. In general, it is more relevant from the user point of view to present minimal covers for rule sets. Minimality could be caracterized by several criteria.

Minimal covers for exact rules (100% confidence) have been extensively studied in formal concept analysis (for example the *Guigues-Duquenne* basis [13, 10]) and database design based on functional dependencies [17]). The *Guigues-Duquenne* basis is minimal with respect to the number of rules which is a relevant criteria from a data mining perspective. Rules of the basis are of the form $p \rightarrow p''$ where p is a *pseudo-closed* set (also called pseudo-intent). An item set p is pseudo-closed if it is not closed and it contains the closure of all its subsets which are pseudo-closed. In [20, 24], the generic basis is proposed as an extension of the Guigues-Duquenne basis by taking into account the support and confidence of the rules. The generic basis can be generated from the *FCIs* by using Algorithm 1 previously presented in [11] on the order covering graph (Hasse diagram) of the *FCIs* and computing the confidence levels using the cardinality of the extents. The covering graph can also be used to efficiently generate a basis for rules constrained by a subset of the item sets. It is therefore useful to maintain not only the *FCIs* but also the covering graph for data mining purposes. Partial rules are rules with a less than 100% confidence. The Luxenburger cover basis [16] is a cover for partial rules. The basis corresponds to rules of the form $X \rightarrow Y - X$ where X and Y are closed and Y covers X. The covering relationship corresponds to the Hasse diagram of the CIs. Therefore, a natural way to present these rules is to show the Hasse diagram to the user. This basis is also easily extended to take into account the support of the rules [19]. These facts support the importance of maintaining the order covering relationship between the FCIs from a data mining perspective.

4 Incremental lattice update for closed itemset generation

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 $\mathcal{K}_{i-1} = (\{o_1, ..., o_{i-1}\}, A, I)$. Each incorporation involves a set of structural updates [26].

4.1 Principles of the incremental approach

The basic approach initially described in [12] and then improved in [27], 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 [4]. 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: L a lattice, o an object)
2:
3:  SORT(L)      {in descending order}
4:  for all c̄ in L do
5:    if Intent(c̄) ⊆ {o}' then
6:      ADD(Extent(c̄),o)      {(c̄) is a modified concept}
7:    else
8:      Int ← Intent(c̄) ∩ {o}'      {(c̄) is an old concept}
9:      if not (Int', Int) ∈ L then
10:       c ← NEW-CONCEPT(Extent(c̄) ∪ {o}, Int)      {(c̄) is a generator}
11:       UPDATE-ORDER(c,c̄) ; ADD(L, c)
```

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

4.2 Description of the lattice algorithm

In the sequel, we consider the subset of the algorithm described in [12] 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 sizes[5] (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 $Intent(\bar{c})$ 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 $Intent(\bar{c})$ 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, $Extent(\bar{c})$, augmented by the new object o, a fact we shall use in computing the support for *CI*s.

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

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 hand-side 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 *CIs*) 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.

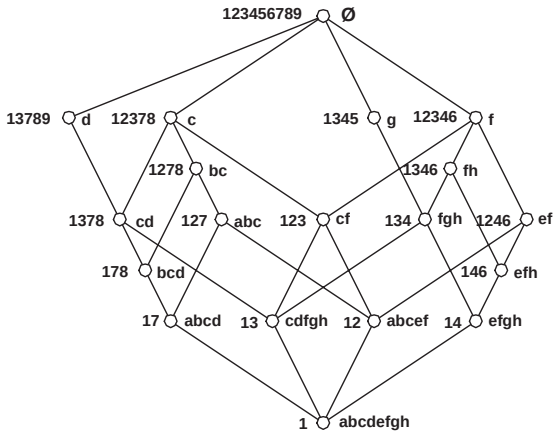

Figure 2: The Hasse diagram of the concept (Galois) lattice derived from $\mathcal{K}$.

4.3 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).

4.3.1 Principles of the approach

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

First, for each transaction T, its itemset I_T is a *CI*. Then, since the family $\mathcal{C}^a_{\mathcal{D}^-}$ 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 it[7]. The set of resulting *CIs*, say $\delta\mathcal{C}^a$, is split into two parts: already existing *CIs* and *new CIs*. 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 minimal[8] *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 *CIs* 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 (maximal[9]) *generator* concept.

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

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

[8] With respect to set-theoretic inclusion.

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

The absolute supports of *CIs* in $\mathcal{C}^a_{\mathcal{D}}$ are obtained from supports in $\mathcal{D}^-$ in the following manner: for all *CIs* 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 *CIs* stay unchanged. This means, in particular, that all the supports of *CIs* corresponding to existing intersections have to be increased by one in $\mathcal{C}^a_{\mathcal{D}}$.

It is important to note that since the approach is incremental, there is a need to keep the whole set of *CIs*, including those which are not frequent. This is due to the fact that after one or many insertions of new transactions, some *CIs* may become frequent while some *FCIs* may become infrequent closed itemsets. Moreover, discarding some *CIs* 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 *FCIs* such as cf, efh, fgh will become infrequent *CIs* (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.

4.3.2 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 *CIs* 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 ← ∅ ; I_n ← T_n.itemset
 6: for all e in FamilyCI do
 7:    I_e ← e.itemset
 8:    if I_e ⊆ I_n then
 9:       e.supp++      {e is modified}
10:    else
11:       Y ← I_e ∩ I_n ; e_Y ← lookup(FamilyCI, Y)      {e is old or potential generator}
12:       if e_Y = NULL then
13:          e_Y ← lookup(NewCI, Y)      {e is a potential generator}
14:          if e_Y = NULL then
15:             node ← new-node(Y, e.support + 1) ; NewCI ← NewCI ∪ {node})
16:          else
17:             e_Y.support ← max(e.support + 1, e_Y.support)
18: FamilyCI ← FamilyCI ∪ 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 *CIs* with maximal support among all *CIs* 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.

[10] A 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.

4.4 Trie-based method

In the following, we describe GALICIA-T, an improved version of GALICIA based on tries.

4.4.1 Trie basics

The *trie* (from re*trie*val) data structure [15] 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 *CIs* over $\mathcal{K}^-$ which is given 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 or insertion of a word 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.

4.4.2 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 *CIs*, *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 items[11] are used to represent transactions and individual *CIs*. 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 ← new-trie() ; I_n ← SORT(T_n.itemset)
6: TRAVERSAL-INTERSECT(I_n, NULL, root(FamilyCI))
7: merge(FamilyCI,NewCI)
```

Algorithm 3: Trie-based update of the *CIs* 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 *CIs* and finally the merge of the two tries[12].

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 ($T_n.itemset$) 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

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

[12] Due 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 ≠ NULL) and (I_n.item = node.item) then
6:    add(Y_curr , I_n.item)
7:  if node.terminal then
8:    n ← lookup(FamilyCI, Y_curr)
9:    if n = 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 ≠ NULL) then
15:   for all n in node.successors do
16:     while (I_n ≠ NULL) and (I_n.item < n.item) do
17:       I_n ← I_n.next
18:     TRAVERSAL-INTERSECT(I_n, Y_curr, n)
```

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

(Y_{curr}) corresponds to a *CI* of $\mathcal{C}^a_{\mathcal{D}}$. In these cases, the status of the set Y_{curr}, i.e., either new *CI* or already existing in $\mathcal{C}^a_{\mathcal{D}^{-1}}$, 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 $\mathcal{C}^a_{\mathcal{D}^{-1}}$). Recall that modified *CIs* are exactly those included, as subsets, in $T_n.itemset$. 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 so, the successors of *node* are listed in a lexicographic order, so that the current itemlist I_n could be gradually reduced (lines $16-17$).

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 $cdfgh$.

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

node.item	I_n	Y_{curr}	terminal	*supp*
e	$cdfgh$	NULL	N	-
f	fgh	f	Y	5
g	gh	fg	N	-
h	h	fgh	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 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*

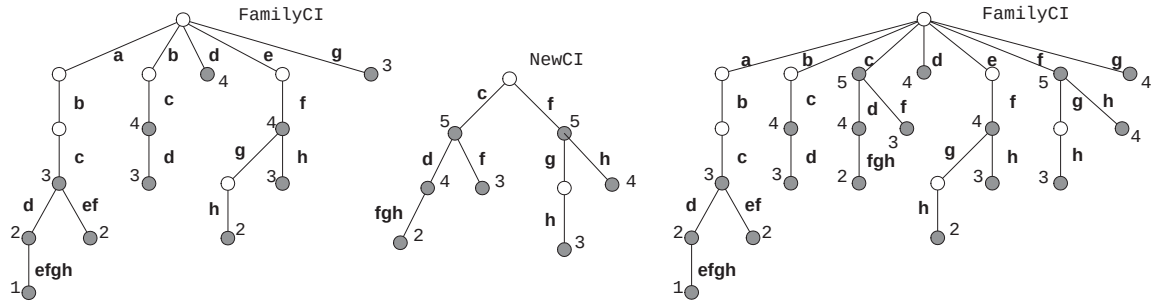

Figure 3: **Left:** The trie *FamilyCI* of the *CIs* generated from $\mathcal{D}^-$. **Middle:** The trie *NewCI* of the new *CIs* relative to transaction #3. **Right:** The trie *FamilyCI* after the insertion of transaction 3.

is provided, the filter would simply enumerate the buckets of *CI* in the index satisfying it.

5 Narrowing the search for updates

The previous algorithm may be improved by the application of some further results about the lattice sub-structures ignored in Algorithm 4 during the process of new object insertion.

5.1 Rationale

Although the use of a trie structure potentially leads to gains both in efficiency and storage, the complete exploration of the entire *CI* family upon each insertion may still prove too expensive for large databases and/or inefficient for sparse ones (see Section 7). In such databases, every transaction insertion concerns only a limited set of either new or exiting *CIs*, whose size is far smaller than the size of the entire *CI* family $\mathcal{C}^a_{\mathcal{K}}$. This fact motivates a smarter incrementation strategy which focuses exclusively on the *relevant* subset of *CIs* instead of traversing completely $\mathcal{C}^a_{\mathcal{K}}$.

Although relevance could be defined in various ways, we start with a narrow definition which amounts to considering only *CIs* that are directly involved in the restructuring of $\mathcal{C}^a_{\mathcal{K}}$, i.e., modified and generator *CIs*. Ideally, a traversal procedure should be able to enumerate those elements according to an unspecified order, so that no other element is even considered. Whatever the feasibility of such an ambitious goal, a tentative discovery of modified and generator nodes will require the storage or just-in-time computation of some order information from the lattice to ease the "jumps" between members of the target set. In turn, upon each insertion, such information would have to be updated, thus leading to a problem which is quite similar to lattice maintenance. Therefore, in the resolution of this problem, we shall reason in a way similar to the extensive search case, i.e., first, define a lattice maintenance procedure and then show how it simplifies to an algorithm for closed itemset update.

5.2 Problem definition and target structures

To solve the problem of selective update of a lattice $\mathcal{L}$, one has to carry out three main steps, each leading to a set of questions to be answered. First, the target set, i.e., $\mathbf{G}(o) \cup \mathbf{M}(o)$ in the current lattice $\mathcal{L}$ should be discovered. Then the new concepts in $\mathbf{N}(o)$ have to be created and their components computed. Finally, the members of $\mathbf{N}(o)$ have to be properly connected to the existing lattice nodes so that $\mathcal{L}$ is transformed into $\mathcal{L}^+$.

Initial work on incremental methods provides few explicit clues on how the tasks one and three are to be addressed: from Godin *et al.* [12] we know that generators are maximal for the attribute set

that results from the intersection between the concept intent and the description of the new object. Moreover, the authors proved that in $\mathcal{L}^+$ every generator is a lower cover of the generated new concept. Later work on a broader class of lattices called *type lattices* [26] has established that the set of all generator concepts induces a meet sub-semi-lattice of both $\mathcal{L}$ and $\mathcal{L}^+$, and that this structure is isomorphic to the sub-order of $\mathcal{L}^+$ induced by $\mathbf{N}(o)$. However, few results have been provided about the precedence relation between new elements. Finally, a recent work [27] has explicitly characterized the precedence relation in $\mathcal{L}^+$ and the way it is obtained from the precedence in $\mathcal{L}$.

In the following paragraph we generalize and complete this partial results into a complete framework that enables the design of "surgical" incremental algorithms.

5.3 Theoretical foundations

We focus on a substructure of $\mathcal{L}^+$ that contains all concepts with o in their respective extents, i.e., both new concepts, $\mathbf{N}(o)$, and modified concepts that will be further noted $\mathbf{M}^+(o)$ to distinguish them from their counterparts in $\mathcal{L}$. This larger structure is the order filter generated by the object-concept of o in $\mathcal{L}^+$, denoted $\nu(o)$. The filter, itself denoted $\uparrow \nu(o)$, induces a complete sublattice of $\mathcal{L}^+$. Its choice for a pivotal structure has been motivated by the existence of an isomorphic structure in $\mathcal{L}$ which is, unsurprisingly, made up of $\mathbf{G}(o)$ and $\mathbf{M}(o)$. Thus, when $\mathbf{N}(o)$ is to be integrated into $\mathcal{L}$, the desired links can be inferred from the structure isomorphic to $\uparrow \nu(o)$ within $\mathcal{L}$.

5.3.1 Set definitions

First, two maps linking the lattices $\mathcal{L}$ and $\mathcal{L}^+$ are defined[13]. The map σ sends a concept c from $\mathcal{L}$ in the concept in $\mathcal{L}^+$ that has an identical intent, whereas a mapping γ sends every c from $\mathcal{L}^+$ to the concept from $\mathcal{L}$ whose extent corresponds to extent of c modulo o.

Definition 3. *The mappings $\sigma : \mathcal{C} \to \mathcal{C}^+$ and $\gamma : \mathcal{C}^+ \to \mathcal{C}$ are established as follows:*

- $\sigma(X, Y) = (Y', Y)$ *where Y' is computed in $\mathcal{K}^+$,*
- $\gamma(X, Y) = (X_1, X_1')$, *where* $X_1 = X - \{o\}$.

Observe that σ is a *join-preserving* order embedding, whereas γ is a *meet-preserving* function. Moreover their subsequent application results in a identity over $\mathcal{C}$, i.e., $\gamma \circ \sigma = id_{\mathcal{C}}$.

To formally define $\mathbf{N}(o)$ within $\mathcal{L}^+$, we use the following fact: if o is dropped out from the context, these concepts will disappear since the result of the subtraction of o from their own extent is an already existing extent.

Definition 4. *The set of new concepts in $\mathcal{L}^+$ is*

$$\mathbf{N}(o) = \{c = (X, Y) | c \in \mathcal{C}^+; o \in X; (X - \{o\})'' = X - \{o\}\}.$$

As opposed to new concepts, the subtraction of o from the extent of modified concepts in $\mathbf{M}^+(o)$, does not change the intent. The corresponding definition may be bridged to $\mathcal{L}$ via γ to define $\mathbf{M}(o)$.

Definition 5. *The sets of modified concepts in $\mathcal{L}^+$ and in $\mathcal{L}$ are:*

- $\mathbf{M}^+(o) = \{c = (X, Y) | c \in \mathcal{C}^+; o \in X; (X - \{o\})' = Y\}$,
- $\mathbf{M}(o) = \{c = (X, Y) | c \in \mathcal{C}; \exists \bar{c} \in \mathcal{M}^+(o), c = \gamma(\bar{c})\}$.

[13] In the following, each time the correspondence operator $'$ is computed in the respective context of the application co-domain (i.e., $\mathcal{K}$ or $\mathcal{K}^+$).

To formally define generators, consider a member $c = (X, Y)$ of $\mathbf{N}(o)$ and observe that by definition the set $X - \{o\}$ is closed in $\mathcal{K}^+$. Hence, there is a concept $\underline{c}$ in $\mathcal{L}^+$ with $\underline{c} = (X - \{o\}, (X - \{o\})')$ and we shall call this concept the *generator* of c in $\mathcal{L}^+$.

Definition 6. *The sets of generator concepts in $\mathcal{L}^+$ and in $\mathcal{L}$ are:*

- $\mathbf{G}^+(o) = \{c = (X, Y) \mid o \notin X; (X \cup \{o\})'' = X \cup \{o\}\}$,
- $\mathbf{G}(o) = \{c = (X, Y) \mid Y \not\subseteq \{o\}'; (X \cup \{o\})'' = X \cup \{o\}\}$, *whereby the closure $''$ is computed within $\mathcal{K}^+$.*

A definition of $\mathbf{G}(o)$ that is closer to the classical one (mentioned in the previous paragraph) relies on the intersection of the concept intent with the description of o. The following property says that generators in $\mathcal{L}$ are those concepts whose intent Y is not included in $\{o\}'$, but Y is the *closure* of its own intersection with $\{o\}'$.

Property 1. *The set of generators in $\mathcal{L}$ is* $\mathbf{G}(o) = \{c = (X, Y) | Y \not\subseteq \{o\}'; Y = (Y \cap \{o\}')''\}$.

5.3.2 Factor structures

We now generalize the intersections with the description of o to the entire set $\mathcal{C}$, i.e., we define a mapping that links $\mathcal{L}$ to the lattice of the powerset of all attributes, 2^A.

Definition 7. *The function $\mathcal{Q} : \mathcal{C} \to 2^A$ computes: $\mathcal{Q}(c) = Y \cap \{o\}'$.*

The function $\mathcal{Q}$ induces an equivalence relation over the set $\mathcal{C}$, whereby the class of a concept c will be denoted $[c]_{\mathcal{Q}}$.

The reverse function of $\mathcal{Q}$ induces an equivalence relation on $\mathcal{C}$. When the set of equivalence classes $\mathcal{C}_{/\mathcal{Q}}$ together with the following order relation:

$$[c_1]_{\mathcal{Q}} \leq_{/\mathcal{Q}} [c_2]_{\mathcal{Q}} \Leftrightarrow \mathcal{Q}(c_2) \subseteq \mathcal{Q}(c_1)$$

are considered, the resulting partial order, $\mathcal{L}_{/\mathcal{Q}}$, is a complete lattice since it is clearly isomorphic to $\uparrow \nu(o)$ (the intents of concepts in $\uparrow \nu(o)$ are all subsets of $\{o\}'$ which are closed in $\mathcal{K}^+$, just as the specific $\mathcal{Q}$ values for each class in $\mathcal{C}_{/\mathcal{Q}}$).

Property 2. *$\mathcal{L}_{/\mathcal{Q}} = \langle \mathcal{C}_{/\mathcal{Q}}, \leq_{/\mathcal{Q}} \rangle$ is a complete lattice.*

Furthermore, a substructure in $\mathcal{L}$ similar to $\uparrow \nu(o)$ may be defined by considering a unique representative for each class in $\mathcal{C}_{/\mathcal{Q}}$. Actually, such a class happens to have a unique maximal element which corresponds to the closure of the respective $\mathcal{Q}$ value.

Property 3. *$\forall\ c = (X, Y) \in \mathcal{C}$, $\exists\ \bar{c} = \max([c]_{\mathcal{Q}})$, whereby $\bar{c} = (\bar{X}, \bar{Y})$ with $\bar{Y} = (Y \cap \{o\}')''$.*

Let us denote by $\mathbf{E}(o)$ the set of all class maxima. From Property 1 and from the trivial observation that $\mathbf{M}(o) \subseteq \mathbf{E}(o)$, we deduce the fact that class maxima are exactly the set of all generators and modified concepts.

Property 4. *The set of all class maxima in $\mathcal{L}$ is* $\mathbf{E}(o) = \mathbf{G}(o) \cup \mathbf{M}(o)$.

The set $\mathbf{E}(o)$ taken as a suborder of $\mathcal{L}$ is clearly isomorphic to $\mathcal{L}_{/\mathcal{Q}}$ (same reasons as for $\uparrow \nu(o)$).

Property 5. $\langle \mathbf{E}(o), \leq_{|\mathbf{E}(o)} \rangle \cong \mathcal{L}_{/\mathcal{Q}} \cong \langle \uparrow \nu(o), \leq^+_{|\uparrow \nu(o)} \rangle$.

The above property generalizes our previous findings expressed in the following property stating isomorphism between key structures for our maintenance algorithm.

Corrolary 1. $\langle \mathbf{G}(o), \leq_{|\mathbf{G}(o)} \rangle \cong \langle \mathbf{N}(o), \leq^+_{|\mathbf{N}(o)} \rangle$, *whereby both structures represent* meet *sub-semi-lattices of their respective global lattices.*

In sum, the increase of a context $\mathcal{K}$ by a new object o results in the integration of a (possibly empty) *meet* semi-lattice into the underlying lattice $\mathcal{L}$, which is isomorphic to an existing sub-semi-lattice.

5.3.3 Precedence relation in $\mathcal{L}^+$

Following Property 5, we are now looking for an efficient way of inferring the structure of $\uparrow \nu(o)$ with respect to $\prec^+$ from the information in $\prec$. Two questions have to be answered, a first one concerning the way the new concepts will be integrated within $\mathcal{L}^+$, i.e., the new precedence links that are to be created, and second, emphasizing on the obsolete links from $\prec$ to be removed in $\prec^+$.

At a preliminary step, a mapping can be defined between $\mathcal{C}$ and $\mathcal{C}^+$ which – when restricted to $\mathbf{E}(o)$ – represents the isomorphism between both sets related to o. The mapping χ sends a concept c from $\mathcal{L}$ to the concept in $\mathcal{L}^+$ whose intent is $\mathcal{Q}(c)$.

Definition 8. *The function $\chi : \mathcal{C} \to \mathcal{C}^+$ is established as $\chi(X, Y) = (Y_1', Y_1)$, where $Y_1 = Y \cap \{o\}'$.*

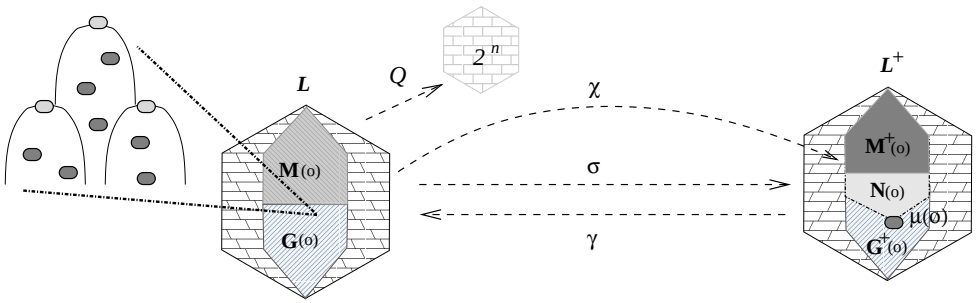

Figure 4: The lattices $\mathcal{L}$, $\mathcal{L}^+$ and the auxiliary lattice $\mathbf{2}^A$ together with the related functions χ, σ, γ and $\mathcal{Q}$.

The entire set of mappings defined in the previous paragraphs is illustrated in Figure 4.

A first step in the solution is, given a new concept c in $\mathbf{N}(o)$, to define its *upper covers* in $\mathcal{L}^+$ (which are clearly among the members of $\uparrow \nu(o)$). These can be identified by looking at the upper covers of the respective generator in $\mathcal{L}$, $\gamma(c)$ in the semi-lattice $\langle \mathbf{E}(o), \leq_{|\mathbf{E}(o)} \rangle$ and taking the images of those concepts by χ. However, the precedence relation of $\langle \mathbf{E}(o), \leq_{|\mathbf{E}(o)} \rangle$ is not directly available and its construction could be an expensive task. Fortunately, the necessary information may be inferred $\prec$, as it was pointed out in [27], with a little additional computation. In fact, using the monotony property of $\mathcal{Q}$ on $\mathcal{L}$, we prove that whenever the class of a generator c_1 is an upper cover of another class in $\mathcal{L}_{/\mathcal{Q}}$ whose generator is c_2, there is an upper cover of c_1 that belongs to $[c_2]_{\mathcal{Q}}$.

Property 6. *For each c_1, c_2 in $\mathbf{G}(o)$: $[c_1]_{\mathcal{Q}} \prec_{/\mathcal{Q}} [c_2]_{\mathcal{Q}} \Rightarrow \exists \bar{c} \in [c_2]_{\mathcal{Q}} : c_1 \prec \bar{c}$.*

Consequently, the upper covers of a new concept can be detected by looking at the upper covers of the respective generator in $\mathcal{L}$ and by taking the minima of their images by χ.

Corrolary 2. *For each c in $\mathbf{N}(o)$ and each $\bar{c}$ in $\mathcal{C}^+$: $c \prec^+ \bar{c} \Leftrightarrow \bar{c} \in \min(\{\chi(\hat{c}) | \gamma(c) \prec \hat{c}\})$.*

At a second step, we consider the upper covers of concepts in $\mathcal{L}^+$ that lay beyond $\mathbf{N}(o)$. In [26], we have shown that the only elements of $\mathcal{C}^+ - \mathbf{N}(o)$ which got new upper covers with respect to $\mathcal{L}$ are the generators in $\mathbf{G}^+(o)$. Thus, given a generator c in $\mathbf{G}^+(o)$ the unique new upper cover is the respective new element $\chi(\gamma(c))$.

Property 7. *For each c in $\mathbf{N}(o)$ $\gamma(\sigma(c)) \prec^+ c$. Moreover, for each $\bar{c}$ in $\mathcal{C}^+ - \mathbf{N}(o)$:*

$$\bar{c} \leq^+ c \Leftrightarrow \bar{c} \leq^+ \gamma(\sigma(c)).$$

Finally, the links to be dropped in $\prec^+$ are exactly those linking a generator to a modified concept in $\mathcal{L}$, as we pointed out in [27]. The following property sums up the results of the above paragraph.

Property 8. *The relation $\prec^+$ is obtained from $\prec$ as follows:*

$$\begin{array}{rlll} \prec^+ = & \{(c_1,c_2) \mid (\gamma(c_1),\gamma(c_2)) \in \prec\} & \cup & \{(\sigma(\gamma(c)),c) \mid c \in \mathbf{N}(o)\} \\ \cup & \{(c,\bar{c}) \mid c \in \mathbf{N}(o),\ \bar{c} \in \mathrm{Min}(\{\chi(\hat{c}) \mid \gamma(c) \prec \hat{c}\})\} & - & \{(c_1,c_2) \mid \gamma(c_1) \in \mathbf{G}(o),\ \gamma(c_2) \in \mathbf{M}(o)\} \end{array}$$

5.4 The algorithm

The structural results from the previous paragraphs underlie a procedure that, given an object o, transforms $\mathcal{L}$ into $\mathcal{L}^+$. The procedure is a first-class lattice construction algorithm whose merits will be examined in a separate study. As our concern is limited to itemset mining, we provide a slightly simplified version that computes a lattice where nodes are closed itemsets with support information and precedence links. The initial *CI* mining procedure that strictly follows the lattice structure is further adapted to work on a flat set of *CIs*, i.e., with no order links.

5.4.1 Principles of the method

The key idea of the algorithm is to discover the set $\mathbf{E}(o)$ in the most efficient way. For that reason, a traversal of the lattice $\mathcal{L}$ discovers the equivalence classes in $\mathcal{C}_{/\mathcal{Q}}$, whereby at each class processing, its maximal element is detected. Thus, unlike previous incremental methods which are exclusively top-down strategies, our method applies a bottom-up traversal which starts at the lattice bottom element. Once the maximum of a class is found, the method relies on Property 6 to move further upwards. Actually, the computation continues with the examination of the upper covers of the current element. While modified concepts are processed by the method the very first time they are met, a separate step represents the creation of new elements and the computation of the appropriate precedence links.

5.4.2 Data structures

The method relies on three abstract data structures: a regular trie, a more advanced trie structure, called KLS-trie, and a stack. The KLS-trie (for Key-Length Sorted trie) is an extensible collection of nodes indexed by unique keys (an intemset or itemlist) with the additional possibility of retrieving nodes in an order depending on the length of the respective keys. The primitives of the KLS-trie structure include insertion of a new node (put()) and lookup (get()). More advanced operations are the retrieval of the longest key not yet examined (get-longest-key()) and the resolution of key unicity conflicts (put-update()). The last operation allows, whenever a key already exists in the KLS-trie, to selectively replace the indexed node with the new node, depending on the value of numerical criteria to optimize. It is noteworthy that a KLS-trie can be efficiently simulated by a regular trie extended with an index based on key-length.

In our algorithm, *Classes* represents the minimal nodes of all classes in $\mathcal{C}_{/\mathcal{Q}}$. It is a KLS-trie whereby the nodes are *CIs*, keys are the respective $\mathcal{Q}$ values and the conflict resolution criterion is the length of the respective *CI*. More precisely, shorter elements are favored, a fact which corresponds to the intuition that the shorter a *CI*, the higher it lays in the lattice, and therefore, the closer it lays to the respective class maximum. The structure is used to guide the exploration of the lattice in such an order that minimizes the efforts of detecting the maximal element of each equivalence class. The remaining structures are *Generators*, a traditional stack of *CIs* used to store generators in $\mathcal{L}$, and *Classes*$^+$, a trie of *CIs* indexed on $\mathcal{Q}$ values which represents $\uparrow \nu(o)$. The first one enables new *CI*s creation while both contribute to order computation.

5.4.3 Algorithmic code

The pseudo-code of our method is given in Algorithm 5. After the initialization step (line 8), the traversal starts from the bottom *CI* that corresponds to the entire itemset which enters the *Classes*

```
 1: procedure UPDATE-LATTICECI(ℒ : a lattice of Cls, T_n : a new transaction)
 2:
 3: Local   : Generators: a stack of Cls
 4:         : Classes⁺: a trie of Cls indexed by itemsets
 5:         : Classes: a KLS-trie of Cls indexed by itemsets
 6:         : M_t the set of modified Cls
 7:
 8: Generators ← ∅ ; Classes ← ∅ ; Classes⁺ ← ∅
 9: I_n ← T_n.itemset ; ⊥_ℒ.valQ ← I_n
10: put(Classes, I_n, ⊥_ℒ)  {insert the bottom in the KLS-trie}
11: while not empty(Classes) do
12:    c ← get-longest-key(Classes)  {extract the biggest intersection yet to process}
13:    c̄ ← CLASSMAX(c)
14:    if is-generator(c̄) then
15:       push(Generators, c̄)
16:    else
17:       c̄.support + +
18:       add(M_t, c̄); put(Classes⁺, c̄.valQ, c̄)
19:    for all ĉ ∈ c̄.succ do
20:       ĉ.valQ ← ĉ.itemset ∩ I_n ; put-update(Classes, ĉ.valQ, ĉ)
21: for all c ∈ Generators do
22:    ĉ ← create-node(c.valQ, c.support + 1) ; UPDATEORDER(ĉ, c)
23:    put(Classes⁺, c.valQ, ĉ)
```

Algorithm 5: Lattice-based construction of the *CI* family of a transaction database.

KLS-trie as a first value (lines 9-10). The next step is the gradual discovery of the equivalence classes in $\mathcal{L}$ (lines 11-20) starting from the bottom in $\mathcal{L}_{/\mathcal{Q}}$. The traversal is guided by the key-length-based order in *Classes* (line 12) which insures that the classes are examined in an order that is compatible with $\leq_{/\mathcal{Q}}$ (although the order of their discovery may be different). The rationale behind that assertion is that the later a class is processed, the greater is the chance of its representative c node to be close to the class maximum $\bar{c}$ (line 13) (and therefore, there is a smaller search effort to find that maximum above c by CLASSMAX). Once the maximum of the currently examined class is discovered, its status, i.e., either generator or modified, is established (line 14). Generators are simply memorized (line 15) while modified nodes are completely processed (lines 17-18). First, the support is increased, then the node is registered both as modified and as class maximum in $\mathcal{L}^+$ (line 18). In both cases, the upper covers of $\bar{c}$ are tentatively inserted into *Classes* with the respective keys (lines 19-20). Finally, the *Generators* stack is gradually examined, at each step pulling out its head and processing the respective node (lines 21-23). This includes the creation of a new *CI* and the computation of the adjacent precedence links (line 22). Finally, the new node is registered in *Classes*$^+$ as the maximum of its respective class in $\mathcal{L}^+$ so that the computation of further precedence links involving the potential lower covers of this node is enabled. It is noteworthy that this procedure relies strongly on a top-down traversal of the set $\mathbf{G}(o)$, insured by the stack data structure and the order of discovery of all generators which is compatible with $\leq$.

5.4.4 Main primitives

The above algorithm uses two main primitives, CLASSMAX and UPDATEORDER. The first one is intuitive. It admits various algorithms ranging from a naive exhaustive exploration of the respective class to an advanced procedure that directly finds the shorted up-going path in the cover graph of the lattice that leads to the maximum class. The second primitive implements the results summarized by Properties 6 and 8. It is detailed in Algorithm 1. The respective new element is connected to its

```
1: procedure UpdateOrder(In: c_n, c_g nodes)
2:
3: Global   : Classes+ a trie of CIs indexed by itemsets
4:
5: Candidates ← ∅
6: for all c ∈ c_g.succ do
7:    c̄ ← get(Classes+, c.valQ)
8:    add(Candidates,c̄)
9: TrueCovers ← Minima(Candidates)
10: for all ĉ in TrueCovers do
11:    New-Link(c_n,ĉ)
12:    if ĉ ∈ M_t then
13:       Drop-Link(c_g, ĉ)
```

Algorithm 6: Computation of the precedence relation for a new node.

upper covers which are chosen as the minima of the candidate set. The candidates are the maxima of the classes in $\mathcal{L}^+$ (found in $Classes^+$) for each upper cover of the generator. Obsolete links are finally dropped out.

The only primitive used by the algorithm, Minima, computes the minimal elements of a set of *CIs* with respect to inverse set inclusion (maxima when inclusion is considered).

5.4.5 Example

In the following, we provide the trace of the algorithm execution on an example including the already augmented dataset from Figure 1 with its corresponding lattice (see Figure 2) and a new object (transaction) 10 with description $bcgh$. The following table provides the state of the main variables and data structures after each step of the main loop from Algorithm 5. The concepts of the lattice in Figure 2 are identified by numbers that correspond to a bottom-up, level-wise breadth-first traversal of the lattice, from left to right. For example, the following list of concepts (identified by their intents) illustrates this numbering: #1 = $abcdefgh$, #2 = $abcd$, #6 = bcd, #8 = cd, #15 = d, and #19 = $\emptyset$. Moreover, in the columns corresponding to *Classes* and $Classes^+$, the elements represent pairs (key, concept-id), where key is actually the respective value for $\mathcal{Q}$ of the second element.

c	$\bar{c}$	*Classes*	$Classes^+$	$\mathbf{M}_t$	$\mathbf{G}_t$
#1	#1	$(bcgh, \#1)$	$\emptyset$	$\emptyset$	$\emptyset$
#1	#1	$(cgh, \#3)$,$(bc, \#2)$ $(gh, \#5)$	—	—	#1
#3	#3	$(bc, \#2)$,$(gh, \#11)$ $(c, \#8)$	—	—	#1,#3
#2	#13	$(gh, \#11)$, $(c, \#16)$	$(bc, \#13)$	#13	#1,#3
#11	#11	$(g, \#17)$,$(c, \#16)$ $(h, \#14)$	$(bc, \#13)$	#13	#1,#3,#11
#14	#14	$(g, \#17)$,$(c, \#16)$ $(\emptyset, \#18)$	$(bc, \#13)$	#13	#1,#3,#11,#14
#16	#16	$(g, \#17)$,$(\emptyset, \#19)$	$(bc, \#13)$,$(c, \#16)$	#13,#16	#1,#3,#11,#14
#17	#17	$(\emptyset, \#19)$	$(bc, \#13)$,$(c, \#16)$ $(g, \#17)$	#13,#16,#17	#1,#3,#11,#14
#19	#19	$\emptyset$	$(bc, \#13)$,$(c, \#16)$, $(g, \#17)$,$(\emptyset, \#19)$	#13,#16,#17,#19	#1,#3,#11,#14

5.5 Closed itemset mining

The above algorithm relies on order information to keep the effort on restructuring $\mathcal{C}^a_{\mathcal{K}}$ minimal. However, besides the clear reduction in the number of examined lattice nodes, there is a new computation and storage overhead due to the presence of the order links. This can be a serious drawback with datasets which generate large number of *CI*s and therefore require additional storage.

5.5.1 Principles

To cope with the memory shortage, we define a more economical version of Algorithm 5 which does not require order information, but rather extracts the necessary minimum from the flat set $\mathcal{C}^a_{\mathcal{K}}$. Thus, the new algorithm, further called GALICIA-M, does not represent a traversal of any structure but rather a search for modified and generator *CI*s through their respective equivalence classes induced by the function $\mathcal{Q}$. Once those classes are available, the *relevant CI*s are detected by taking the elements that are smallest in size or, alternatively, have the strongest support (which also means they are minimal with respect to the set inclusion). For this purpose, the classes need first to be constituted explicitly.

The main improvement with respect to GALICIA-T is in the fact that the algorithm avoids the explicit construction of the upper most class in $\mathcal{L}_{\mathcal{Q}}$, i.e., the class of the top element in $\mathcal{L}$ (corresponding to the itemset included in each transaction). The gain of the new strategy is particularly high with sparse transaction sets where no item is shared by all transactions and the number of existing *CIs* whose intersection with the new transaction is empty approaches the size of the entire *CIs* family. In other words, the algorithm examines explicitly only those *CI* whose intersection with T_n is non-trivial, i.e., does not represent a subset of the universal *CI*. Thus, compared to the lattice-based method, the new strategy trades the parsimony in the examination of candidate *CIs* (lattice nodes) for lower storage requirements and no order computation overhead.

```
 1: procedure UPDATE-FAMILYCI(C^a : an indexed set of CIs, T_n : a transaction)
 2:
 3: Local   : Classes a set of sets of CIs
 4:         : Candidates a set of CIs
 5:
 6: ⊤_{C^a} ← MAX(C^a) ; I_top ← ⊤_{C^a}.itemset
 7: I_n ← T_n.itemset ; Candidates ← ∅
 8: for all i ∈ I_n - I_top do
 9:    UPDATE(Candidates, lookup(C^a,i)) {gradually construct all non-trivial intersections}
10: Classes ← SORT(Candidates) {separate the classes in C^a_Q}
11: for all Θ ∈ Classes do
12:    c ← MAX(Θ) {extract the node of maximal support in Θ}
13:    if is-generator(c) then
14:       ĉ ← create-node(c.itemset ∩ I_n, c.support + 1)
15:       for all i ∈ ĉ.itemset do
16:          put-at-item(C^a,ĉ,i)
17:    else
18:       c.support + +
```

Algorithm 7: Order-free update of the *CI* family of a transaction database.

5.5.2 Description of the algorithm

The algorithm starts by filtering all classes of $\mathcal{L}_{\mathcal{Q}}$ except for the upper most one. Then it computes the maximum of each class and determines its status, i.e., modified or generator. At a final step, the

necessary updates (support increase, new *CI* creation, etc.) are carried out. Algorithm 7 describes the main steps of the approach.

5.5.3 Example

The efficient construction of the classes in $\mathcal{C}^a_{\mathcal{Q}} - [\top_{\mathcal{C}^a}]$ is supported by a simple indexing structure that allows all *CIs* that share a given item i to be found in a direct manner. The structure can be thought of as a vector of (ordered) lists of *CI* identifiers whereby each entry in the vector is indexed by an item. Consider now the example given in section 5.4, i.e., the insertion of a transaction 10 with description *bcgh* into the TDB $\mathcal{D} = \{1, 2, 3, \cdots, 9\}$. The following table illustrates the vector entries associated to the four items in I_n (itemsets are given instead of identifiers for clarity reasons).

Item	Indexed *CIs*
b	bc, abc, bcd, $abcd$, $abcef$, $abcdefgh$
c	c, bc, cd, cf, abc, $abcd$, $abcef$, $cdfgh$, $abcdefgh$
g	g, fgh, $efgh$, $cdfgh$, $abcdefgh$
h	fh, efh, fgh, $efgh$, $cdfgh$, $abcdefgh$

The content of *Classes* after the sorting step (line 10) is presented in the table below, together with the indication of the class maximum and its respective status (generator or modified).

$\mathcal{Q}(c)$	$[\,]_{\mathcal{Q}}$	MAX($[\,]_{\mathcal{Q}}$)	Status	$\mathcal{Q}(c)$	$[\,]_{\mathcal{Q}}$	MAX($[\,]_{\mathcal{Q}}$)	Status
c	c, cd, cf	c	$\mathbf{M}_t$	bc	bc, abc, bcd, $abcd$, $abcef$	bc	$\mathbf{M}_t$
g	g	g	$\mathbf{M}_t$	cgh	$cdfgh$	$cdfgh$	$\mathbf{G}_t$
h	fh, efh	fh	$\mathbf{G}_t$	$bcgh$	$abcdefgh$	$abcdefgh$	$\mathbf{G}_t$
gh	fgh, $efgh$	fgh	$\mathbf{G}_t$				

At any creation of a new *CI* $\hat{c}$ the lists corresponding to each item in $\hat{c}.itemset$ are updated by adding the identifier of $\hat{c}$ to them (lines 15-16 in Algorithm 7).

6 Related work

Since the first publication of the *Apriori* algorithm [2], there have been an impressive number of approaches to the problem of association rule mining, most of them aimed at improving the efficiency of the initial algorithm. For a survey about the subject, the reader is refereed to [14]. In the following, we report in a non-exhaustive way, some studies that present one or both of the key features discussed in our paper, i.e., being incremental or computing the *FCIs*.

One of the earliest work on incremental mining is due to Cheung *et al.* [6] 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 FUP_2 is proposed [7] for updating association rules when insertion, deletion, and modification of transactions occur. Both FUP and FUP_2 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. Two other incremental algorithms were proposed independently by [9] and [25]. Both of them are based on the notion of negative border [14] and allow the update of large itemsets when a set of transactions are added to or deleted from the initial transaction database. The update is made possible by maintaining support counters for the frequent itemsets and the negative border. In [3], an incremental algorithm called $UWEP$, handles a look-ahead pruning by discarding any itemset that will become non-frequent as early as possible.

[14] The negative border of a collection of itemsets $L \subseteq \mathcal{P}(A)$, closed under $\subseteq$, contains all the minimal itemsets in A that are not in L [18].

A recent work reported in [23] extends the limits of incremental approaches by allowing changes to the basic parameters of the mining process such as support threshold, and analyzing the impact of the increment (new transactions) on the mining process.

Alternative approaches to mining *CIs* from a database have been presented in [30, 21], both following the theoretical guidelines of the *Galois lattice/FCA* domain [4, 10]. 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 [14] for a survey). The CLOSET algorithm [22] relies on a recursive construction of *FP-trees* to build the set of *FCIs*.

Based on the criteria described in [23], we believe that our approach has the following attractive features: (i) it is incremental, (ii) it allows flexible changes to the support threshold, and (iii) it helps capture the effects of the increment by highlighting the newly discovered *FCIs* and the changes in the support of some existing ones. The last feature helps analyze 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).

7 Experimental results

We conducted a set of tests in which both variants of GALICIA have been compared to the non-incremental algorithm CLOSET [22]. Such a choice was motivated by the features shared by both procedures: (i) the computation of *FCIs*, and (ii) the use of a 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. The experiments were performed on a 1.3 GHz AMD TB processor with 1.2 GB main memory, running Windows 2000. Both algorithms were implemented in `Java`^{TM}, 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.

Two synthetic databases [2], namely T25.I20.D100K and T25.I10.D10K were used in the experiments. The dataset T25.I20.D100K is a large but relatively *sparse* one: it includes 100,000 transactions over 10,000 items where each transaction has 25 items on average, and the average size of the maximal potentially frequent itemset is 20. This dataset generates 12,868,438 closed itemsets of which 313,409 are of support larger than 0.05% (50 transactions) and 27,112 of support larger than 0.5% (500 transactions). The second dataset, T25.I10.D10K, is a smaller but rather *dense* one: 10,000 transactions over 1,000 items with average values of 25 and 10 for transaction and maximal frequent itemset sizes, respectively. A total of 3,530,786 closed itemsets are generated by this dataset, with 23,852 of them being of support larger than 0.5% (50 transactions). Table 2 offers a more detailed picture of the way the above figures evolve when increasing subsets of the entire datasets are considered.

TDB size	Nb of *CIs*	Nb of *FCIs* support = 50
2,000	281,209	544
4,000	826,114	2,275
6,000	1,562,211	6,977
8,000	2,479,770	14,701
10,000	3,530,786	23,852

TDB size	Nb of *CIs*	Nb of *FCIs* support = 50	Nb of *FCIs* support = 500
10,000	420,144	22,326	11
20,000	1,148,803	73,851	52
...	...	...	...
90,000	10,895,757	271,074	22,998
100,000	12,868,438	313,409	27,112

Table 2: **Left:** T25I10D10K, the evolution of respective sizes for the CI and FCI families (support of 50). **Right:** T25I20D100K, the evolution of respective sizes for the CI and FCI families (supports of 50 and 500).

The main statistics that were collected for each algorithm and dataset include the execution time

for three types of tasks: processing a single new transaction, processing an increment of several new transactions, and processing the entire dataset. To provide a better idea about the trends that lay behind each algorithm, we recorded the above statistics for datasets of variable size. Thus, both datasets have been separated into increments of fixed size, 2,000 transactions for T25.I10.D10K and 10,000 for T25.I20.D100K. For each increment, the tests have been carried out with a fixed absolute support threshold for CLOSET (50 for T25.I10.D10K, 50 and 500 for T25.I20.D100K).

Another important aspect of our study puts the focus on memory requirements. In a very general manner, we have registered a surge in the storage space required by GALICIA-T. For example, its consumption in the case of T25.I20.D100K exceeded the available 1 GB[15] for 45,000 transactions which prevented a further sensible comparison of performances. Therefore, in the rest of this section, we only provide the statistics of the GALICIA-M variant. The following table summarizes the total memory consumption of both algorithms on the various settings:

Dataset	GALICIA-M	CLOSET support = 50	CLOSET support = 500
T25.I10.D10K	456 MB	63 MB	—
T25.I20.D100K	1 GB (swap after 85 K transactions)	823 MB	389 MB

Two types of comparisons have been carried out. The first one (see the left-hand side of Figures 5 and 6) aimed at comparing the performance of both algorithms as batch procedures, i.e. when applied on static datasets. The results of these tests show the clear advantage of CLOSET (and most probably of some other batch techniques such as CHARM or A-CLOSE) over our method. For reasonable values of the support threshold, CLOSET proved to be up 30 times faster on T25.I20.D100K and up to 100 times faster on T25.I10.D10K. Only tiny support values, i.e., when almost all the *CIs* are to be kept, tend to favor our method.

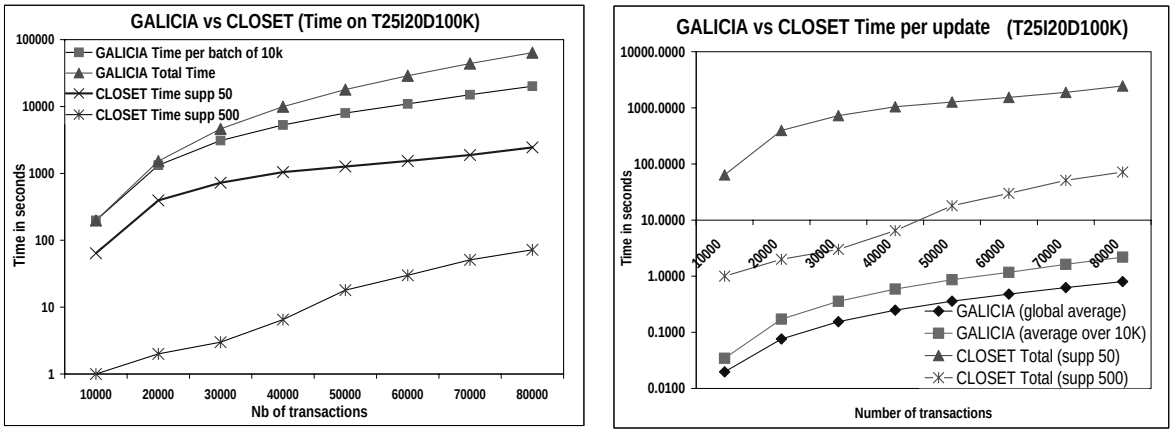

Figure 5: **Left**: Total CPU-time for both GALICIA and CLOSET for increasing subsets of T25I20D100K, with *min-supp* fixed to absolute values (50 and 500). **Right**: CPU-time for the insertion of a single transaction, average over both the total set and the current batch of 10 000 transactions compared to the CPU-time for running CLOSET on the entire transaction set.

The second type of tests (see the right-hand side of Figures 5 and 6) highlights the overhead induced by re-running CLOSET on the whole updated database versus running GALICIA with the increment only. Both the diagrams show important trends. First, while the total time taken by CLOSET might

[15] This seems to be the maximally allowed RAM allocation for the Java VM.

lay orders of magnitude lower than the total time of GALICIA, it lays also orders of magnitude higher than the update time for a single new transaction. For example, when T25I20D100K is concerned, the processing of half the database, i.e., 50,000 transactions, may well take five hours for GALICIA and only 20 minutes for CLOSET (see Figure 5 on the left). In the same time, the insertion of a single transaction in GALICIA 'costs' just below a second (0.8 seconds, Figure 5 on the right). Next, with the sparse dataset, the average insertion cost for GALICIA and the total mining cost for CLOSET are quasi-linear functions of the dataset size. The above facts provide some evidence to support the benefits

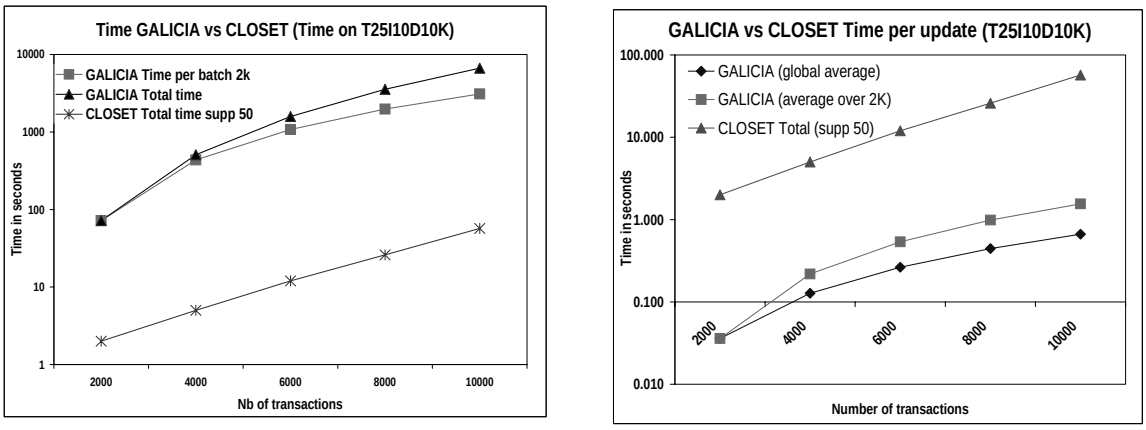

Figure 6: **Left**: Total CPU-time for both GALICIA and CLOSET for increasing subsets of T25I10D10K, with *min-supp* fixed to an absolute value of 50. **Right**: CPU-time for the insertion of a single transaction, average over both the total set and the current incrment of 2,000 transactions, compared to the CPU-time for running CLOSET on the entire transaction set.

of the incremental approach. In fact, running CLOSET once with an augmented dataset may cost up to hundred times more than the time spent for inserting a single transaction with GALICIA. In other words, one may run, say, several hundreds of insertions with GALICIA 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 execution time remains too high. However, with a dynamic database, the mining process is spread over the entire database life-cycle (usually long) so that the main question becomes the establishment of a proper trade-off between the update costs and the urgent need for intermediate results.

When taken as a whole, the experimental results suggest that the benefits of the parsimonious update strategy in GALICIA-M are more substantial with sparse datasets than with dense ones. This may be due to the fact that execution time gains with respect to GALICIA-T inversely depend on the ratio between the number of examined elements and the total size of the *CI* family, a value which is weaker in the former case.

8 Discussion

Incrementality is a major challenge for data mining methods. The proposed framework for incrementally mining frequent closed itemsets is a first step towards achieving that goal. The framework is based on the theory of Galois lattice and FCA whose benefits for the association rule mining problem have already been demonstrated. Two concrete mining algorithms have been devised within the framework, one straightforward and the other one using a pruning mechanism, with an additional valuable feature which is the low-cost response to a readjustment in the *minsupp*. Both algorithms were derived from lattice update procedures, whereby we provided a set of lattice structural properties that underlie

the pruning strategy and formulated a novel incremental algorithm for lattice construction.

Appropriate implementation of the basic algorithms have been discussed as well, and their respective practical performances were compared to those of a major batch algorithm. The results of a preliminary experimental study on two synthetic datasets of contrasted profiles revealed some potential benefits but also important limitations in the incremental paradigm. When taken as a whole, they seem to suggest that a straightforward incremental approach of the kind described here will most probably prove inefficient in purely static databases when the target support threshold is known *a priori*. However, the approach will certainly be more appealing for 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, incremental mining procedures may be very helpful in environments where the user may want to frequently: (*i*) modify the support threshold of *FIs* for a given *TDB*, and/or (*ii*) process new transactions in dynamic databases and analyze the impact of such new transactions on the mining result.

The scalability of our incremental approach is clearly obstructed by the necessity of maintaining the whole set of frequent closed itemsets. 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 efficient method for *FI* computation which relies on *CIs*, e.g., CLOSET, A-CLOSE or CHARM. The latter could be applied as a preprocessing 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 [28]. The underlying framework offers a large choice of possible operations on results upon updates in the dataset (e.g., insert or remove individual transactions or transaction batches). 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] N. Ayan, A. Tansel, and M. Arkun. An efficient algorithm to update large itemsets with early pruning. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 99)*, pages 287–291, San Diego, CA, 1999. ACM Press.

[4] M. Barbut and B. Monjardet. *Ordre et Classification: Algèbre et combinatoire*. Hachette, 1970.

[5] 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.

[6] 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.

[7] 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.

[8] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1992.

[9] R. Feldman, Y. Aumann, A. Amir, and H. Mannila. Efficient Algorithms for Discovering Frequent Sets in Incremental Databases. In *2nd SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages 59–70, 1997.

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

[11] R. Godin and R. Missaoui. An Incremental Concept Formation Approach for Learning from Databases. *Theoretical Computer Science*, 133:378–419, 1994.

[12] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept formation algorithms based on galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.

[13] J.L. Guigues and V. Duquenne. Familles minimales d'implications informatives résultant d'un tableau de données binaires. *Math. Sci. Humaines*, 95:5–18, 1986.

[14] J. Hipp, U. Guentzer, and G. Nakhaeizadeh. Algorithms for Association Rule Mining - A General Survey and Comparison. *SIGKDD Explorations*, 2(1):58–64, 2000.

[15] D. E. Knuth. *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison-Wesley, Reading (MA), second edition, 1998.

[16] M. Luxenburger. Implications partielles dans un contexte. *Mathématiques et Sciences Humaines*, 29(113):35–55, 1991.

[17] D. Maier. *The theory of Relational Databases*. Computer Science Press, 1983.

[18] H. Mannila, H. Toivonen, and A. Verkamo. Efficient algorithms for discovering association rules. In U. Fayyad and R. Uthurusamy, editors, *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, pages 181–192, Seattle, WA, 1994. AAAI Press.

[19] N. Pasquier. Extraction de bases pour les règles d'association à partir des itemsets fermés fréquents. In *Proceedings of the 18th INFORSID'2000*, pages 56–77, Lyon (FR), 2000.

[20] N. Pasquier, Y. Bastide, T. Taouil, and L. Lakhal. Closed Set Based Discovery of Small Covers for Association Rules. In *Proceedings of 15èmes Journées Bases de Données Avancées (BDA'99)*, pages 361–381, 1999.

[21] 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.

[22] 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.

[23] V. Pudi and J. R. Haritsa. Quantifying the Utility of the Past in Mining Large Databases. *Information Systems*, 25(5):323–343, 2000.

[24] 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, Febuary 2000. IEEE Computer Society.

[25] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining (KDD 97)*, pages 263–266, New Port Beach, CA, 1997.

[26] P. Valtchev. An algorithm for minimal insertion in a type lattice. *Computational Intelligence*, 15(1):63–78, 1999.

[27] 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, Stanford (CA)*, volume 2120 of *Lecture Notes in Computer Science*, pages 290–303. Springer-Verlag, 2001.

[28] P. Valtchev, R. Missaoui, and P. Lebrun. A partition-based approach towards building Galois (concept) lattices. *to appear in Discrete Mathematics*, 2001.

[29] 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.

[30] M.J. Zaki. Generating Non-Redundant Association Rules. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining (KDD'00)*, 2000.

[31] M.J. Zaki and C.-J. Hsiao. ChARM: An Efficient Algorithm for Closed Association Rule Mining. Rpi technical report 99-10, Rensselaer Polytechnic Institute, 1999.