2m11.3275.10

Université de Montréal

Mining Dynamic Databases for Frequent Closed Itemsets

par

Jun Jing

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des Études Supérieures
en vue de l'obtention du grade de
maîtrise ès sciences M.Sc.
en Informatique

Juillet 2004

© Jun Jing, 2004

QA
76
U54
2004
v.047

Université de Montréal

Faculté des Études Supérieures

Ce mémoire intitulé :

Mining Dynamic Databases for

Frequent Closed Itemsets

présenté par:

Jun Jing

a été évalué par un jury composé des personnes suivantes :

Geňa Hahn

Président Rapporteur

Petko Valtchev

Directeur de recherche

Miklós Csűrős

Membre du Jury

Mémoire accepté : 19 octobre 2004

Résumé

La fouille de données, aussi connue sous le nom de la découverte de connaissance dans la base de données (DDBC), consiste à découvrir des informations cachées et utiles dans des grandes bases de données. La découverte des règles d'association est une branché importante de la fouille de données. Elle est utilisée pour identifier des dépendances entre articles dans une base de données. L'extraction des règles d'association a été prouvée pour être très utile dans le commerce et les champs d'activité qui lui sont proche.

La plupart des algorithmes d'extraction des règles d'association appliquent à des bases de données statiques seulement. Si la base de données se développe (nouvelles transactions sont ajoutées), nous devrions re-exécuter ces algorithmes du début pour toutes les transactions afin de produire le nouvel ensemble des règles d'association parce que l'ajout de nouvelles transactions peut rendre des itemsets fréquents (itemsets fréquents fermés) invalide ou générer de nouveaux itemsets fréquents (itemsets fréquents fermés), ce qui influence les règles d'association. Ces algorithmes n'essayant pas d'exploiter les résultats obtenus de l'ensemble des transactions. À ce jour, plusieurs algorithmes incrémentaux de mise à jour progressive ont été développés pour la maintenance des règles d'association.

Dans cette thèse, nous traitons les aspects algorithmiques de l'extraction des règles d'association. Particulièrement, nous nous concentrons sur l'analyse de quelques algorithmes incrémentaux basés sur les connexions de Galois, comme *GALICIA* et *GALICIA-T*. Nous avons aussi étudié certains algorithmes de calcul d'itemsets fréquents, comme *Apriori* algorithm. En se basant sur ces algorithmes, nous proposons un nouvel algorithme, appelé l'algorithme de treillis d'iceberg (*ILA*), qui utilise peu d'opérations pour maintenir à jour la structure de l'iceberg lors de l'insertion d'une nouvelle transaction dans l'ensemble des transactions. Ceci devrait être utile pour l'amélioration de la performance des algorithmes existants basés sur le treillis de Galois (le treillis de concept).

Mots clés: treillis de Galois (concept), treillis iceberg, algorithme de construction de treillis, méthodes incrémentales.

Abstract

Data Mining, also known as Knowledge Discovery in Databases (**KDD**), is the discovery of hidden and meaningful information in a large database. Association rule mining is an important branch of data mining. It is used to identify relationships within a set of items in a database (or transaction set). Association rule mining has been proven to be very useful in the retail communities, marketing and other more diverse fields.

Most association rule mining algorithms apply to static transaction sets only. If the transaction set evolves (i.e. new transactions are added), one needs to execute these algorithms from the beginning to generate the new set of association rules, since adding new transactions may invalidate existing frequent itemsets (or frequent closed itemsets) or generate new frequent itemsets (or frequent closed itemsets), which will influence the association rules. These algorithms do not attempt to exploit the results obtained from the original transaction sets. To date, many incremental updating proposals have been developed to maintain the association rules.

In this thesis, we deal with the algorithmic aspects of association rule mining. Specifically, we focus on analyzing some incremental algorithms based on Galois connection, such as *GALICIA*, and *GALICIA-T*. We also study some plain frequent itemsets mining algorithms, such as *Apriori* algorithm. Based on these, we propose a new algorithm, called Iceberg Lattice Algorithm (*ILA*), which uses only a few operations to maintain the iceberg structure when a new transaction is added to transaction set. It should be helpful in improving the performance of existing algorithms that are based on Galois lattices (concept lattices).

Key words: Galois (concept) lattices, iceberg lattices, lattice constructing algorithms, incremental methods.

Contents

List of Tables

List of Figures

Acknowledgments

Many thanks must go to my director, Professor Petko Valtchev, for his knowledgeable contribution and guidance throughout the research. It is he who patiently led me into this area.

I would like to thank my parents, my parents-in-law and my wife for their encouragement and support of me whenever I encountered difficulty. Special thanks go to my son. He brings my family much joy.

I would also like to extend gratitude to my classmates and friends, especially Amine, who has helped me to solve a lot of technical problems. Without them, my work could not have been finished so soon.

Chapter I Introduction

1.1 What is data mining?

Data mining [PS1991] is the process of discovering hidden and useful information in a large database. It is a decision-making tool based on Artificial Intelligence and statistical techniques that consists of analyzing the automatically acquired data, making inferences and abstracting a potential model for demonstrating the correlations among the elements in databases. One of the important operations behind data mining is finding trends and regularities, commonly called *patterns*, in large databases.

Current technology makes it is easy to collect data, but it tends to be slow and expensive to carry out the data analysis with traditional database systems since these systems offer little functionality to support analysis. Before the data mining era, massive amounts of data were left unexplored or on the verge of being thrown away. However, there may be valuable and useful information hiding in the huge amount of unanalyzed data and therefore new methods for digging interesting information out of the data are necessary.

There are many kinds of basic patterns that can be mined, such as *associations*, *causalities*, *classifications*, *clusterings*, *sequences* and so on. Association rule mining finds patterns where one data item is connected to another one. Causality mining discovers the relationship between causes and effects. Given a set of cases with class labels, classification builds an accurate and efficient model (called *classifier*) to predict future data item for which the class label is unknown. Clustering is the discovery of fact groups (called clusters) that are not previously known. Sequence mining discovers frequent sequences of items in large databases. Sequences are similar to associations, but they focus on an analysis of order between two data items. In this thesis, we focus on associations. Indeed, discovery of interesting associations among items is useful to decision making, as it allows us to make predictions based on the recorded previous observations.

Many data mining approaches apply to static datasets only. If the set is frequently updated (as with dynamic datasets), a new problem arises since adding new data may invalidate existing frequent patterns or generate new ones. A simple solution to the update problem is to re-mine the whole updated datasets. This is clearly inefficient because all frequent patterns mined from the old datasets are wasted. A more suitable approach consists in incremental data mining [HSH1998]. It attempts to exploit the results obtained from the original datasets while analyzing only with small additional effort on the original set.

1.2 Association rule mining

Association rules were introduced in 1993 by Rakesh Agrawal, Tomasz Imielinski, and Arun Swami [AIS1993]. There are two steps on the process: finding all frequent itemsets and generating association rules from them. Frequent pattern mining is the core in mining associations. Many methods have been proposed for this problem. These methods can be classified into two categories: frequent plain pattern mining [AS1994, HF1995, PCY1995, BA1999] and frequent closed pattern mining [PBTL1999-2, PHM2000]. The main challenge here is that the mining step often generates a large number of frequent itemsets and hence association rules. The frequent closed pattern mining is a promising solution to the problem of reducing the number of the generated rules.

Closed patterns or itemsets mining are rooted in the Formal Concept Analysis (FCA) [GW1999]. FCA provides the theoretical framework for association rule mining. It focuses on the partially ordered structure, known as Galois lattice [BM1970] or concept lattice [W1982], which is induced by a binary relation R over a pair of sets T (transactions) and I (items). In 1982, Wille proposed to regard each element in a lattice as a concept and the corresponding graph (Hasse diagram) as the relationship between concepts [W1982]. Association mining approaches based on the Galois (concept) lattice construction have been proposed. [GMA1995, CHNW1996, VMG2002].

GALICIA is an incremental frequent closed itemsets mining algorithm based on Galois lattices. It exploits the results obtained from the original datasets when new data are added. But it is inefficient since it explores the entire set of closed patterns, i.e. the frequent and infrequent. Actually one needs to limit the set of generated closed patterns to the frequent ones.

1.3 Our contribution

In this thesis, in order to attack the problem of mining frequent closed itemsets incrementally, we introduce and implement a new algorithm, *ILA* (Iceberg Lattice Algorithm). *ILA* algorithm is an incremental method based on iceberg lattice construction. Unlike *GALICIA*, *ILA* maintains only the upper most part of a concept lattice. Therefore, it improves the efficiency of the mining task by reusing the previous results, thus avoiding unnecessary computation.

1.4 Thesis organization

The organization of the rest of the thesis is organized as follows. Chapter II introduces the basic concepts of association rule mining and Galois (concept) lattice, describes current state of research on association mining algorithms and reviews two typical algorithms. Since Iceberg Lattice Algorithm is an enhanced *GALICIA* approach, Chapter III reviews the *GALICIA* approach. Chapter IV presents the motivation and theoretical foundation of Iceberg Lattice Algorithm. Chapter V presents Iceberg Lattice Algorithm step by step and accompanied by a detailed example. We also discuss complexity issues. Chapter VI implements our algorithm and studies its performance. Chapter VII summarizes the thesis and discusses possible directions for future work. Appendix presents the proofs of properties used in this thesis, and the validation of the Iceberg Lattice Algorithm.

Chapter II Data Mining

2.1 Association rules mining

2.1.1 Basic concepts of association rules

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of distinct items. A transaction set T is a multi-set of subsets of I that are identified.

Definition 2-1: We suppose a function TID: $T \rightarrow N$, where T is a transaction set and N is a set of natural numbers.

Definition 2-2: A subset $X \subseteq I$ with $|X| = \mathrm{k}$ is called a *k-itemset*. The fraction of transactions that contain X is called the *support* (or *frequency*) of X, denoted by *supp* (X):

$$supp\ (X) = \frac{|\{t \in T \mid X \subseteq t\}|}{|T|}.$$

Definition 2-3: A set of TID is called *tid-set*.

Example

Assume $I = \{a, b, c, d, e, f, g, h\}$ is a set of distinct items.

A transaction set is as follows:

TID	Itemsets
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	$\{d\}$

Table 2-1: An example of a transaction set

(1, $\{a, b, c, d, e, f, g, h\}$) and (8, $\{b, c, d\}$) are transactions, 1 and 8 are the TID of these transactions. The size of $\{a, b, c, d, e, f, g, h\}$ is 8, so $\{a, b, c, d, e, f, g, h\}$ is an 8-itemset. The size of $\{b, c, d\}$ is 3, so $\{b, c, d\}$ is a 3-itemset. Since three transactions (#1, #7 and # 8) contain itemset $\{b, c, d\}$ and the total number of transactions is 9, so $supp\ (\{b, c, d\}) = \frac{3}{9}$.

Definition 2-4: If the support of an itemset X is above a user-defined minimal threshold (*minsupp*), then X is frequent (or large) and X is called a *frequent itemset* (*FI*).

For example: given $minsupp = 0.3$

$supp\ (\{b, c, d\}) = \frac{3}{9} > 0.3$, so $\{b, c, d\}$ is frequent (large), we call $\{b, c, d\}$a frequent itemset

$supp\ (\{a, b, c, e, f\}) = \frac{1}{9} < 0.3$, so $\{a, b, c, e, f\}$ is non-frequent (non-large).

Rakesh Agrawal, Tomasz Imielinski, and Arun Swami proposed Property 2-1 and Property 2-2 in [AIS1993].

Property 2-1: All subsets of a frequent itemset are frequent.

Property 2-2: All supersets of an infrequent itemset are infrequent.

Definition 2-5: An *association rule* is an expression $X \Rightarrow Y$, where X and Y are subsets of I, and $X \cap Y = \varnothing$.

The *support* of a rule $X \Rightarrow Y$ is defined as $supp\ (X \Rightarrow Y) = supp\ (X \cup Y)$. The *confidence* of this rule is defined as $conf\ (X \Rightarrow Y) = supp\ (X \cup Y)\ / supp\ (X)$.

For example: $\{b, c\} \Rightarrow \{d\}$ is an association rule, $supp(\{b, c\} \Rightarrow \{d\}) = supp\{b, c, d\} = \frac{3}{9}$,

$$conf\ (\{b, c\} \Rightarrow \{d\}) = supp(\{b, c, d\})\ /\ supp(\{b, c\}) = \frac{3/9}{4/9} = \frac{3}{4} = 0.75.$$

Mining association rules in a given transaction set T means generating all assciation rules that reach a user-defined *minimal support* (*minsupp*) and *minimal confidence* (*minconf*). This problem can be divided into two steps.

- Finding all frequent itemsets.
- Generating association rules from frequent itemsets.

The generation of association rules from frequent itemsets is relatively straightforward [AS1994], since one divides a FI into two complementary parts to make a rule between premise and conclusion. Therefore the research in the domain has focused on determining frequent itemsets and their support.

2.1.2 Basics of concept lattices

Concept lattices are used to represent conceptual hierarchies that are inherent in some data. They form the core of the mathematical theory of Formal Concept Analysis (FCA) [W1982]. Initially FCA was introduced as formalization of the notion of *concept*, now it is a powerful theory for data analysis, information retrieval and knowledge discovery [GW1999]. In Artificial Intelligence, FCA is used as a knowledge representation mechanism. In database theory, it has been used for class hierarchy design and management [SS1998, WTL1997]. In the Knowledge Discovery in Databases (KDD), FCA has been used as a formal framework for discovering association rules [STBPL2000]; furthermore, it has been successful in improving the performance of algorithms that mine association rules [PBTL1999-1].

The basics of ordered structures

Definition 2-6: Consider a set G and $a, b, c \in G$. A *partial order* $\leq$ on G is a reflexive ($a \in G \mid a \leq a$), anti-symmetric ($a, b \in G \mid a \leq b \;\&\; b \leq a \Rightarrow a = b$) and transitive ($a, b, c \in G \mid a \leq b \;\&\; b \leq c \Rightarrow a \leq c$) relation.

Definition 2-7: The set G in conjunction with an associated partial ordering relation $\leq_G$ is called a *partially ordered set* or *poset* or *partial order* and is denoted by $(G, \leq_G)$.

Definition 2-8: Let $P = (G, \leq_G)$ be a partial order. For a pair of elements, $s, p \in G$, if $p \leq_G s$, we shall say that s succeeds (is greater than) p and p precedes s. All common successors of s and p are called *upper bounds* of s and p. All common predecessors of s and p are called *lower bounds* of s and p.

Definition 2-9: Let $P = (G, \leq_G)$ be a partial order and A be a subset of G, if there is an element $s \in G$ such that s is the minimal of all upper bounds of A, then s is called the *least upper bound* of A (LUB); if there is an element $p \in G$ which is the maximal of all lower bounds of A, then p is called the *greatest lower bound* of A (GLB).

Definition 2-10: The precedence relation $<_G$ in P is the transitive reduction of $\leq_G$, i.e. $s <_G p$ if $s \leq_G p$ and all t such that $s \leq_G t \leq_G p$ satisfy $t = s$ or $t = p$. If $s <_G p$, s will be referred to as an *immediate predecessor* of p and p as an *immediate successor* of s.

Usually, P is represented by its covering graph $Cov\ (P) = (G, <_G)$, also called the Hasse diagram. In this graph, each element s 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.

Definition 2-11: If a subset A of G satisfies $\forall\ s, p \in A, s \leq_G p \vee p \leq_G s$, then the set A is called a *chain* and the elements are said to be *pair wise comparable.*

Definition 2-12: If a subset A of G satisfies $\forall\ s \in G, \forall\ p \in A,\ s \leq_G p \Rightarrow s \in A$, then the set A is called an *order ideal.*

Definition 2-13: If a subset A of G satisfies $\forall\ s \in G, \forall\ p \in A,\ p \leq_G s \Rightarrow s \in A$, then the set A is called an *order filter.*

Definition 2-14: A lattice $L = (G, \leq_L)$ is a partial order in which every pair of elements s, p has an unique *greatest lower bound* (GLB) and an unique *least upper bound* (LUB). LUB and GLB define binary operators on G called, respectively, *join* ($s \vee_L p$) and *meet* ($s \wedge_L p$).

Definition 2-15: Given a lattice $L = (G, \leq_L)$, all the subsets A of the G have a GLB and a LUB, we call this lattice a complete lattice.

Definition 2-16: A structure with only one of the *join* and *meet* operations is called a *semi-lattice.*

The existence of a unique GLB for every pair of elements implies a meet semi-lattice structure and the existence of a unique LUB for every pair of elements implies a join semi-lattice structure.

Definition 2-17: A formal *context* is a triplet $K = (T, I, R)$ where T, I are sets and $R \subseteq T \times I$ is a binary relation. The elements of T are called *transactions* (or *objects*) and the elements of I *items* (or *attributes*). Each pair $(t, i) \in R$ indicates that i is an *item* of transaction t.

Definition 2-18: T, I are sets, the (f, g) is a *Galois connection* between 2^T and 2^I, $f: 2^T \rightarrow 2^I$, $g: 2^I \rightarrow 2^T$ iff, for all $X \in 2^T$ and $Y \in 2^I$, $f(X) \subseteq Y \Leftrightarrow g(Y) \subseteq X$.

Definition 2-19: Let $K = (T, I, R)$ be a formal context, the function f maps a set of transactions onto a set of items that are common, whereas g is the dual function for the set of items. f and g are defined by '.

$$f(X) = X' = \{i \in I \mid \forall t \in X, t\,R\,i\}$$

$$g(Y) = Y' = \{t \in T \mid \forall i \in Y, t\,R\,i\}$$

For example: $f(\{1, 6\}) = \{1, 6\}' = \{e, f, h\}$ and $g(\{e, f, h\}) = \{e, f, h\}' = \{1, 4, 6\}$.

R.Wille proposed Property 2-3 and Property 2-4 in [W1982].

Property 2-3: (f, g) is a Galois connection of the formal context.

Property 2-4: Compound operators $f \circ g\ (Y)$ and $g \circ f\ (X)$ are Galois *closure operators* over 2^I and 2^T respectively. Hereafter, both $f \circ g\ (Y)$ and $g \circ f\ (X)$ are expressed by ".

$f \circ g\ (Y) = f(g(Y)) = Y''$

$g \circ f\ (X) = g\ (f\ (X)) = X''$

For example: $\{e, f, h\}'' = f(g\ (\{e, f, h\})) = f(\{1, 4, 6\}) = \{e, f, h\}$,

$\{1, 4, 6\}'' = g\ (f(\{1, 4, 6\})) = g\ (\{e, f, h\}) = \{1, 4, 6\}$.

X'' is the closure of X, which is the smallest closed itemset containing X.

For example: $X = \{a, b\}$,

$X'' = \{a, b\}'' = \{a, b, c\}$.

Definition 2-20: An itemset X is closed if $X = X''$.

If an itemset X is closed, adding an arbitrary item i from $I\text{-}X$ to X resulting a new itemset $\underline{X}$ which is less frequent [PHM2000].

Property 2-5: Suppose X is closed, then $\forall\ i \in I\text{-}X,\ supp\ (X \cup \{i\}) < supp\ (X)$.

For example: $X = \{a, b, c, e, f\}$, $I\text{-}X = \{d, g, h\}$,

X is a closed itemset and $supp\ (\{a, b, c, e, f\}) = \frac{2}{9}$.

$supp\ (\{a, b, c, d, e, f\}) = supp\ (\{a, b, c, e, f, g\})$

$= supp\ (\{a, b, c, e, f, h\}) = \frac{1}{9} < \frac{2}{9}$.

Every itemset has the same support as its closure. This property has been proven by [PBTL1999-1]

Property 2-6: $supp(X) = supp\ (X'')$.

Definition 2-21: If a closed itemset X is frequent, then we call it a *frequent closed itemset*

(*FCI*). Namely: $FCI = \{X \mid X \in CIs \wedge supp(X) \geq minsupp\}$.

For example: given $minsupp = 0.3$,

$\{a, b, c\}$ is a closed itemset and $supp(\{a, b, c\}) = \frac{3}{9} > 0.3$, so $\{a, b, c\}$ is a frequent closed itemset.

Definition 2-22: A *concept c* is a pair of sets (X, Y) where $X \in 2^T$, $Y \in 2^I$, $X = Y'$ and $Y = X'$. X is called the *extent* of concept c and denoted by $ext(c)$, Y is called the *intent* of the concept c and denoted by $int(c)$.

For example: in Table 2-1, $\{1,2,7\}' = \{a, b, c\}$ and $\{a, b, c\}' = \{1,2,7\}$, so $(\{1,2,7\}, \{a, b, c\})$ is a concept. $\{1, 2, 7\}$ is its extent and $\{a, b, c\}$ is its intent.

For a concept $c = (X, Y)$, since $X = Y' = \{X'\}' = X''$, so the intent of a concept is a closed itemset.

Definition 2-23: The support of a concept equals to that of its extent, it is defined as follows.

For a concept $c = (X, Y)$, $supp(c) = \frac{|X|}{|T|}$.

For example: in Table 2-1, $(\{1,2,7\}, \{a, b, c\})$ is a concept, $supp((\{1,2,7\}, \{a, b, c\})) = \frac{3}{9}$.

Definition 2-24: If the support of a concept c is above a user-defined minimal threshold (*minsupp*), then c is frequent (or large) and c is called a *frequent concept*.

Definition 2-25: Let C be the set of concepts derived from a context. The partial order $L = (C, \leq_L)$ is a complete lattice called a *concept lattice*. The partial order is defined as follows.

$\forall (X_1, Y_1), (X_2, Y_2) \in C$, $(X_1, Y_1) \leq (X_2, Y_2)$ iff $X_1 \subseteq X_2 \wedge Y_2 \subseteq Y_1$.

A concept lattice $L = (C, \leq_L)$ is a partial order in which every pair of concepts, c_1, c_2, has a unique *greatest lower bound* and a unique *least upper bound.* The binary operators on C denoted, respectively, *join* $(c_1 \vee_L c_2)$ and *meet* $(c_1 \wedge_L c_2)$[W1982]:

$$(X_1, Y_1) \vee_L (X_2, Y_2) = (\{X_1 \cup X_2\}'', \{Y_1 \cap Y_2\}),$$

$$(X_1, Y_1) \wedge_L (X_2, Y_2) = (\{X_1 \cap X_2\}, \{Y_1 \cup Y_2\}'').$$

The Hasse diagram of the concept lattice L from Table 2-1 is shown in Figure 2-1. Intents and extents are indicated in rectangles below the nodes. For example, the join and meet of $c_{\#5} = (\{1,2,3\}, \{c, f\})$ and $c_{\#4} = (\{1,3,7,8,9\}, \{d\})$ are $c_{\#0} = (\{1,2,3,4,5,6,7,8,9\}, \varnothing)$ and $c_{\#12} = (\{1,3\}, \{c, d, f, g, h\})$ respectively.

Two functions, μ and ν, are defined on the concept lattice.

Definition 2-26: The function μ: $T \to L$ is defined as follows:

$$\mu(t) = \wedge\{c | \, t \in ext(c)\} = (\{t\}'', \{t\}').$$

Given a transaction t, this function is used to find a *minimal concept c* (according to the size of extent) in L and $ext(c)$ includes the transaction t.

For example, within the concept lattice in Figure 2-1, $\mu(2) = c_{\#11}$ and $\mu(6) = c_{\#13}$.

Definition 2-27: The function ν: $I \to L$ is defined as follows:

$$\nu(i) = \vee\{c | \, i \in int(c)\} = (\{i\}', \{i\}'').$$

Given an item i, this function is used to find a *maximal concept c* (according to the size of extent) in L and *int* (c) includes the item i.

For example, within the concept lattice in Figure 2-1, $\nu(d) = c_{\#4}$ and $\nu(f) = c_{\#1}$.

Hereafter, $\uparrow c$ denotes the set of all successors of concept c (the order filter generated by c) and $\downarrow c$ denotes the set of all predecessors of concept c (the order ideal generated by c).

The notion and properties of iceberg were introduced in [STBPL2000].

Definition 2-28: Given *minsupp* $\alpha \in [0, 1]$, C^{α} is the set of all α-frequent concepts and the partial order $(C^{\alpha}, \leq_P)$ is called the *iceberg concept lattice.*

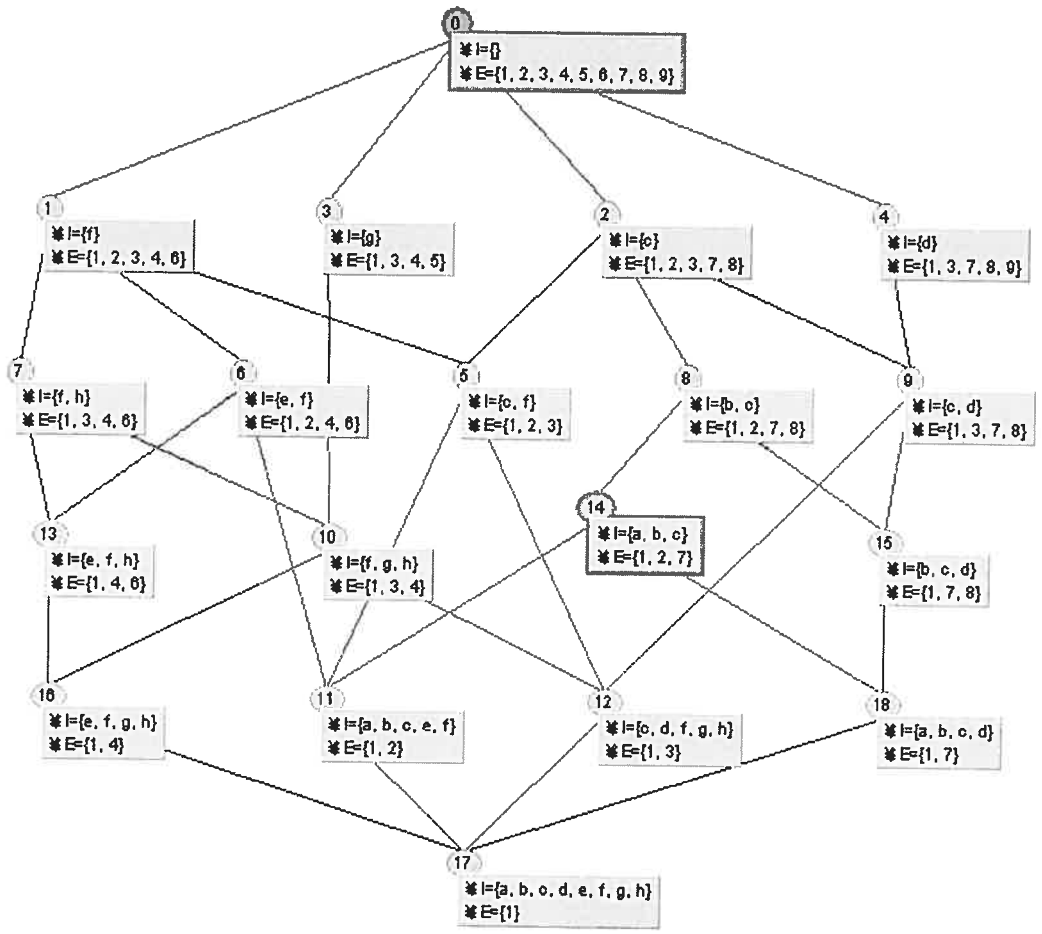

Figure 2-1: The concept lattice from Table 2-1

Property 2-7: L^{α} is an upper-semi-lattice (or join-semi-lattice) of L.

The set of all α-infrequent concepts in L forms a sub-semi-lattice (join-semi-lattice) of L.

For example, given a *minsupp* $\alpha = 0.3$, the iceberg lattice from Table 2-1 is shown in Figure 2-2. The support of every concept in Figure 2-2 is greater than 0.3.

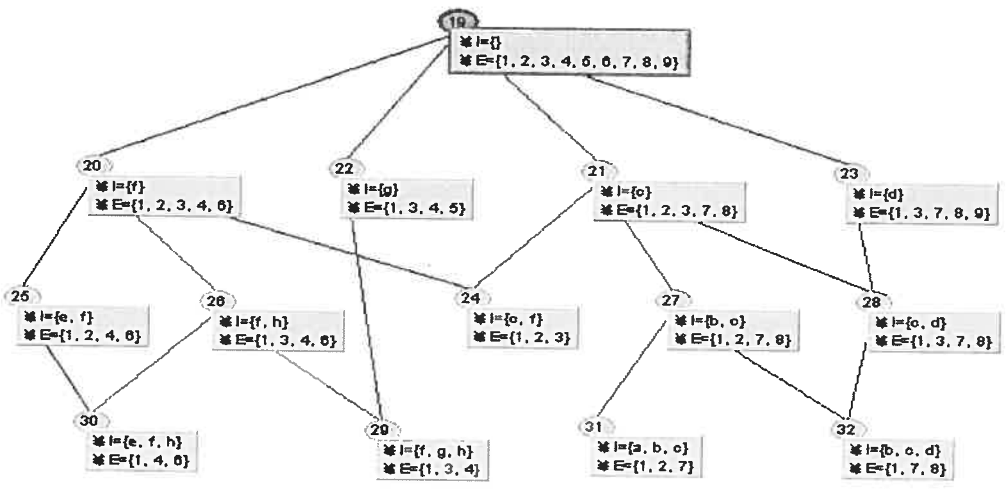

Figure 2-2: The iceberg lattice $L^{0.3}$ from Table 2-1 (α=0.3)

2.2 Current state of research on association mining algorithms

Many algorithms for association rule mining have been designed, we can classify them into several categories.

2.2.1 Plain frequent itemsets mining algorithms

In 1993, Rakesh Agrawal, Tomasz Imielinski, and Arun Swami proposed the notion of an association rule and a corresponding algorithm, called *Apriori*, to discover all significant association rules between itemsets in a large transaction set [AIS1993]. *Apriori* is a famous algorithm and enumerates every single frequent itemset. It uses the downward closure property of itemsets support to prune the search space - all subsets of a frequent itemset must be frequent. Only the frequent k- itemsets are used to construct candidate ($k+1$)-itemsets. A pass is executed over the transaction set to find the ($k+1$)-frequent itemsets from the ($k+1$)-candidates.

Many variants of *Apriori* achieve improved performance by reducing the number of candidates. Some algorithms reduce the number of transactions to be scanned [AS1994, HF1995, PCY1995] and some reduce the number of transaction set scans [BMUT1997, SON1995, T1996].

FP-growth is anther well-known algorithm which finds complete frequent itemsets [HPY2000]. It first constructs a compressed data structure, *frequent-pattern tree* (*FP-tree*), to hold the entire transaction set in memory and then recursively builds conditional FP-trees to mine frequent patterns. FP-tree is an extended prefix-tree structure and all transactions with the same prefix share the portion of a path from the root. FP-growth algorithm avoids the problem inherent to candidate generate-and-test approach, thus its performance is reported to be better than that of *Apriori*. However, the number of conditional FP-trees is in the same order of magnitude as number of frequent itemsets. The algorithm is not scalable to sparse and very large transaction sets.

2.2.2 Frequent closed itemsets mining algorithms

Frequent itemsets mining often generates a large number of frequent itemsets and rules. This process reduces the efficiency of mining since one has to filter a large number of mined rules to get useful ones. *A-close* algorithm is an important alternative that was proposed by N.Pasquier, Y.Bastide, R.Taouil, and L.Lakhal [PBTL1999-2]. It uses closure operators to calculate the frequent closed itemsets and their corresponding rules.

As a continued study on *FP-growth*, [PHM2000] proposed *CLOSET*. It is another efficient algorithm for mining frequent closed itemsets based on FP-tree. The special features of this particular algorithm are the three techniques developed for the purpose of complexity reduction. First, *CLOSET* applies an extended frequent-pattern tree to mine closed itemsets without candidate generation. Secondly, to quickly identify frequent closed itemsets, it develops a single prefix path compression technique. Finally, this scheme explores a partition-based projection mechanism for patterns on subsets of items.

CHARM [ZH2002] is another efficient algorithm for mining all frequent closed itemsets. This algorithm implements a hybrid search technique, called dual itemset-tidset search tree (IT-tree) which enables it to skips many levels of the IT-tree to locate the frequent closed itemsets quickly. A fast hashtable-based approach is also used in order to remove non-closed sets found during computation.

In a sparse transaction set, the majority of frequent itemsets are closed itemsets. The performance of *A-Close* is therefore close to that of *Apriori*. The advantage of *CLOSET* over *A-Close* is essentially the same as that of *FP-Growth* over *Apriori* [PHM2000]. In this kind of transaction set, *CHARM* also outperforms *Apriori* due to the fast hash-table and the dual itemset-tidset search tree. If the *minsupp* is small and the TID sets for frequent itemsets are small, *CHARM* will be efficient. However, its performance is inferior than that of *CLOSET* since *CLOSET* employs the closure mechanism on a more elaborate scale. The benefit of *CLOSET* becomes even more significant on dense transaction sets, since *CLOSET* only scans the transaction sets twice and the mining process is confined to the frequent pattern tree after that. Also, regardless of how many times the transaction sets are being iterated, the frequent pattern tree maintains the same shape with respect to the constant *minsupp*. Hence, the runtime of *CLOSET* over real transaction sets increases at a much slower rate than that of the sizes of transaction sets [PHM2000].

A recent algorithm *TITANIC* [STBPL2000] is another algorithm based on Galois connections for mining frequent closed itemsets. It is inspired by the *Apriori* algorithm, as well as adopts a more powerful pruning strategy. This strategy determines the support of all k-itemsets that remain at the k^{th} iteration, and computes the closure of all $(k\text{-}1)$ -itemsets after the $(k\text{-}1)^{th}$ iteration.

2.2.3 Incremental FI or FCI mining algorithms

The most important problem with association mining is the huge number of frequent itemsets and association rules that can be generated from a large transaction set. The methods based on the frequent closed itemsets are a promising solution to the problem of reducing the number of association rules. However, confronting a dynamic transaction set, another problem arises since the transaction set is frequently updated. Adding new transactions may invalidate existing frequent patterns or generate new ones, thus one needs to re-execute the algorithms from the beginning. So far, a few incremental algorithms for association mining have been proposed [GMA1995, CHNW1996, STBPL2000, VMG2002].

[VMG2002] proposed an incremental algorithm for mining frequent closed itemsets based on lattice construction (*GALICIA*). The difference between it and other FCI-based techniques is: it avoids reconstructing the frequent closed itemsets completely when transactions are added to the transaction set and / or the *minsupp* is changed. However, as mentioned earlier, it is inefficient since one needs filter frequent closed itemsets from closed itemsets.

2.3 Review of two typical algorithms

In this section, we review two best-known association rule algorithms: *Apriori*, a plain frequent itemsets mining algorithm, and *A-Close*, a frequent closed itemsets mining algorithm.

2.3.1 *Apriori* algorithm

Apriori algorithm uses Property 2-1 and Property 2-2. It performs a number of iterations. In each iteration (i), it first constructs a set of candidate itemsets based on frequent itemsets obtained from the preceding iteration (i-1); then scans the transaction set to filter the frequent i-itemsets.

The procedures used in the *Apriori* algorithm are shown in Figure 2-3 and Figure 2- 4 (C_k represents the set of candidate k-itemsets, FI_k represents the set of frequent k-itemsets).

Aprior_Gen() is a sub-function of the algorithm. It generates the candidate itemsets by joining the frequent itemsets of the previous pass that have the same items except for the last one, and then generated candidates that contain an infrequent subset are dropped.

Apriori () is the main procedure of the *Apriori* algorithm. It has three main steps. First, it considers the itemsets with only one item (lines 2-3) and calculates frequent 1-itemsets. Secondly, it executes an iterative process, calling *Aprior_Gen*(), to get the frequent itemsets (lines 4-9). This iterative process terminates when no new frequent itemsets can be found. Finally, all frequent itemsets are accumulated (line 10).

```
1:  procedure: Aprior_Gen (input: a set of all frequent $(k-1)$- itemsets $FI_{k-1}$;
2:                            output: the set of all $k$-itemsets candidate $C_k$)
3:      $I_i$ = item $i$                  // beginning of join step
4:      Insert ($C_k, I_i$)               // insert $I_i$ into $C_k$
5:      for each $p$ and $q \in FI_{k-1}$ do
6:          if $p.I_1 = q.I_1$ and … and $p.I_{k-2} = q.I_{k-2}$ and $p.I_{k-1} < q.I_{k-1}$
7:          then insert ($C_k$,($p.I_1, p.I_2, .. ,p.I_{k-1}, q.I_{k-1}$))
8:      for $c \in C_k$ do    // beginning of pruning step
9:          for all $(k-1)$-subsets $s$ of $c$ do
10:             if ($s \notin FI_{k-1}$)
11:             then delete $c$ from $C_k$
```

Figure 2-3: The procedure of *Apriori-Gen ()*

```
1: procedure Apriori (input: $I$, $T$, $\alpha$; output: $FIs$)
2: for all $i \in I$ do
3:     $FI_1 \leftarrow$ {large 1-itemsets} // generate $FI_1$ by traversing transaction set and counting the
                                        // support for elements in $I$
4: for ($k$=2; $FI_{k-1} \neq \varnothing$; $k$++) do
5:   begin
6:     $C_k$ = Apriori-Gen($FI_{k-1}$);
7:     count_supp ($T$, $C_k$,) // calculate the support of $C_k$ in $T$
8:     $FI_k = \{c \in C_k \mid supp(c) \geq \alpha\}$
9: end
10: $FIs = \cup_{1 \text{ to } k} FI_k$
```

Figure 2-4: The algorithm of *Apriori* ()

For example, according to the transaction set in Table2-1, FI_2 ={{*a, b*}, {*a, c*}, {*b, c*}, {*b, d*}, {*c, d*}, {*c, f*}, {*e, f*}, {*e, h*}, {*f, g*}, {*f, h*}, {*g, h*}}, after the joining step in *Apriori-Gen ()*, C_3 will be {{*a, b, c*}, {*b, c, d*}, {*c, d, f*}, {*e, f, h*}, {*f, g, h*}}. The prune step will delete {*c, d, f*} because itemset {*d, f*} is not in FI_2, so C_3 is left with {{*a, b, c*}, {*b, c, d*}, {*e, f, h*}, {*f, g, h*}}. After calculating their supports by traversing the transaction set, we get FI_3 = {{*a, b, c*}, {*b, c, d*}, {*e, f, h*}, {*f, g, h*}}.

According to the experimental results, *Apriori* outperforms other plain frequent itemset mining algorithms [AIS1993]. Since its introduction, two enhanced versions of *Apriori* algorithms were developed: *Apriori-TID* [AIS1994] and *Apriori-Hybrid* [AIS1994]. The main difference between *Apriori* and *Apriori-TID* is that *Apriori* scans the entire transaction set in each pass to count the support in order to discover frequent itemsets. *Apriori-TID* does not use the transaction set for counting support after the first pass. It employs an encoding of the candidate itemsets used in the previous pass. *Apriori-Hybrid* algorithm is a combination of *Apriori* and *Apriori-TID*. As mentioned in [AIS1994], *Apriori* has better performance in earlier passes; *Apriori-TID* has better performance in later passes. *Apriori-Hybrid* technique uses *Apriori* in the initial passes, and switches to *Apriori-TID* for the later passes if necessary. *Apriori-Hybrid* technique improves the performance greatly.

2.3.2 *A-Close* algorithm

A-Close is a non-incremental algorithm for mining frequent closed itemsets which is inspired by *Apriori* algorithm.

The *A-Close* algorithm contains several main procedures. The first is the *AC-Generator* function, which is based on the properties of closed itemsets. It determines a set of generators. Here the generator is defined as follows: an itemset p is a generator of a closed itemset c_i if it is the smallest itemset that will determine c_i using the Galois closure operator: $p'' = c_i$. [W1982]. In this procedure, it applies *Apriori-Gen* () to the *i*-generator set to obtain the (*i+1*)-generator candidate set. It joins two *i*-generators with the same first *i-1* items to produce a new potential (*i+1*)-generator, after getting (*i+1*)-generator candidates, their supports are calculated, then the infrequent $(i+1)$-generators and $(i+1)$-generators that have the same closure as one of their i-subsets are discarded.

The second is the *AC-Closure* function. Once all frequent generators are found, they will help us to get all frequent closed itemsets by using Galois closure operators ". In order to reduce the cost of the closure computation, *A-Close* algorithm adopts an optimized pruning strategy by locating a $(i+1)$-generator that was pruned because it had the same closure as one of its i-subsets to start the first iteration. All iterations before i^{th}, the generators are closed, so it is unnecessary to carry out the closure computation for them, and then we just perform the closure computation for generators of size greater or equal to i. For this purpose, the algorithm uses the *level* variable to indicate the first iteration for which a generator was pruned by this pruning strategy [PBTL1999-2].

A-Close algorithm finds the candidate generators during iterations. It is necessary to traverse the transaction set to calculate the support for the candidate generators. If a generator is not closed, it will require one more pass to determine its closure; if all generators are closed, this pass is not needed.

For example, from transaction set in Table 2-1, *A-Close* algorithm discovers the *FCIs* as follows with $minsupp = 0.3$. First, the algorithm discovers the set of 1-generators, G_1 and the support for each element, no generator is deleted, because all are frequent. Then 2-genetators in G_2 are determined by applying the *AC-Generators* function to G_1, all infrequent 2-generators are pruned (all infrequent 2-generators are not shown due to the limitation of the space), meanwhile, $\{a, b\}$, $\{a, c\}$, $\{e, f\}$ and $\{f, h\}$ are pruned since $supp(\{a, b\}) = supp(\{a\})$, $supp(\{a, c\}) = supp(\{a\})$, $supp(\{e, f\}) = supp(\{e\})$ and $supp(\{f, h\}) = supp(\{h\})$. The *level* variable is set to 2. Calling *AC-Generators* function with G_2 to produce G_3, we get $\{b, c, d\}$ and $\{c, d, f\}$, but $\{b, c, d\}$ is pruned since $supp(\{b, c, d\}) = supp(\{b, d\})$ and $\{c, d, f\}$ is pruned since $\{c, f\} \notin G_2$. At last, since the *level* variable is 2, so the candidate generators G' includes the generators from G_1 and G_2, the closure function *AC-Closure* is applied to G' to discover the closures of all generators in G' and duplicate closures are removed from G', we get all frequent closed itemsets. Figure 2-5 illustrates the whole process.

G_1				G_1		
Support_count	Generator	Support		Pruning infrequent generators	Generator	Support
	{a}	3			{a}	3
	{b}	4			{b}	4
	{c}	5			{c}	5
	{d}	5			{d}	5
	{e}	4			{e}	4
	{f}	5			{f}	5
	{g}	4			{g}	4
	{h}	4			{h}	4
G_2 (all frequent 2-generators)				G_2		
AC-Generator	{a,b}	3		Pruning	{b,c}	4
	{a,c}	3			{b,d}	3
	{b,c}	4			{c,d}	4
	{b,d}	3			{c,f}	3
	{c,d}	4			{e,h}	3
	{c,f}	3			{f,g}	3
	{e,f}	4			{g,h}	3
	{e,h}	3				
	{f,g}	3				
	{f,h}	4				
	{g,h}	3				
G_3				G_3		
AC-Generator	{b,c,d}	3		Pruning		
	{c,d,f}	2				
G'				*FCIs*		
	Generator	closure	support	Pruning	Closure	support
AC-Closure	{a}	{a,b,c}	3		{a,b,c}	3
	{b}	{a,b,c}	3		{c}	5
	{c}	{c}	5		{d}	5
	{d}	{d}	5		{e,f}	4
	{e}	{e,f}	4		{f}	5
	{f}	{f}	5		{g}	4
	{g}	{g}	4		{f,h}	4
	{h}	{f,h}	4		{b,c}	4
	{b,c}	{b,c}	4		{b,c,d}	3
	{b,d}	{b,c,d}	3		{c,d}	4
	{c,d}	{c,d}	4		{c,f}	3
	{c,f}	{c,f}	3		{e,f,h}	3
	{e,h}	{e,f,h}	3		{f,g,h}	3
	{f,g}	{f,g,h}	3			
	{g,h}	{f,g,h}	3			

Figure 2-5: *A-Close* frequent closed itemsets discovery for *minsupp* = 0.3

After studying above algorithms, in this thesis, we explore the problems of mining frequent closed itemsets. We attack these problems by implementing our proposed algorithm – *ILA* (Iceberg Lattice Algorithm). *ILA* algorithm takes advantage of incremental methods and maintains only the upper most part of a concept lattice. As a critical feature, *ILA* algorithm improves the efficiency of frequent closed itemsets mining by avoiding useless computation and taking advantage of the previous iceberg structure. Namely, we improve *ILA* by (1) only scanning the transaction sets once and (2) only storing current frequent closed itemsets when a new transaction is added.

Chapter III Review of the GALICIA approach

GALICIA is a family of algorithms that generate frequent closed itemsets incrementally. Its aim is to construct new closed itemsets based on current family of closed itemsets by looking on the new transaction t_{i+1}. The incremental construction of the concept lattice may help design the effective methods for frequent closed itemsets mining.

3.1 Lattice updates

Incremental methods construct the concept lattice L starting from the initial lattice $L_0 = (\{\varnothing, I\}, \varnothing)$. When adding a new transaction t_{i+1}, we incorporate it into the concept lattice L_i. Each incorporation causes a series of structural updates. The basic approach was defined for concept lattices [GMA1995] and was improved upon later [VM2001]. It is based on the fundamental property of the Galois connection established by f and g on (T, I): both families of closed subsets are themselves closed under set intersection [BM1970]. So when inserting a new transaction t_{i+1}, we should insert into L_i all new concepts whose intent is the intersection of $\{t_{i+1}\}$' and the intent of an existing concept where the intersection is not an already existing intent.

We now define a mapping γ that links the concept lattices L_i and L_{i+1}. The mapping γ sends every c from L_{i+1} to the concept from L_i whose extents correspond to the extent of c modulo t_{i+1}.

Definition 3-1: The mappings γ: $C_{i+1} \rightarrow C_i$ are established as follow.

$$\gamma(X, Y) = (X_1, X_1'), \text{ where } X_1 = X - \{t_{i+1}\}.$$

All concepts in L_i can be classified into three categories (hereafter C_i and C_{i+1} denote the sets of concepts in L_i and L_{i+1} respectively).

Genitor concepts ($\boldsymbol{G}(t_{i+1})$) -- generate new concepts by intersecting with new transaction t_{i+1} and help calculate the respective new intents and extents.

Definition 3-2: The sets of genitor concepts in L_{i+1} and in L_i are

$$G^{+}(t_{i+1}) = \{c = (X, Y) \mid t_{i+1} \notin X;\ (X \cup \{t_{i+1}\})'' = X \cup \{t_{i+1}\}\},$$

$$G(t_{i+1}) = \{c = (X, Y) \mid Y \not\subseteq \{t_{i+1}\}'; Y = (Y \cap \{t_{i+1}\}')''\}$$ respectively.

Modified concepts ($\boldsymbol{M}(t_{i+1})$) -- their intents are included in the new transaction's itemsets, so their intents will remain stable, only the TID of new transaction t_{i+1} is integrated into their extents.

Definition 3-3: The sets of modified concepts in L_{i+1} and in L_i are

$$M^+(t_{i+1}) = \{c = (X, Y) \mid c \in C^+; t_{i+1} \in X; (X - \{t_{i+1}\}') = Y\},$$

$$M(t_{i+1}) = \{c = (X, Y) \mid c \in C, \exists e \in \mathrm{M}^+(t_{i+1}), c = \gamma(e)\}$$ respectively.

Old concepts ($\boldsymbol{O}(t_{i+1})$) -- remain completely unchanged when adding a new transaction t_{i+1}.

Definition 3-4: The set of new concepts in L_{i+1} is

$$N^+(t_{i+1}) = \{\{c = (X, Y) \mid c \in C_{i+1}; t_{i+1} \in X; (X - \{t_{i+1}\})'' = X - \{t_{i+1}\}\}.$$

The incremental algorithms focus on a substructure of L_{i+1} that contains all concepts with t_{i+1} in their respective extents, i.e. both new concepts, $N^+(t_{i+1})$ and $M^+(t_{i+1})$. We regard this structure as an order filter, which is generated by the transaction-concept of t_{i+1} in L_{i+1}, denoted by $\mu(t_{i+1})$. The order filter $\uparrow\mu(t_{i+1})$ induces a complete sub-lattice of L_{i+1}. The choice of a pivotal structure is determined by the isomorphic structure in L_i, which is composed of $G(t_{i+1})$ and $M(t_{i+1})$. Thus when $N^+(t_{i+1})$ is integrated into L_i, the desired links can be inferred from the structure isomorphic to $\uparrow\mu(t_{i+1})$ within L_i.

In order to generalize the intersections of the description of t_{i+1} and the entire set C, we define a mapping that links L to the lattice of the power-set of all attributes, 2^{I}.

Definition 3-5: The function Q: $C \rightarrow 2^{\mathrm{I}}$ computes: $Q(c) = Y \cap \{t_{i+1}\}'$.

The function Q induces an equivalence relation on the set C, and the class of a concept c is denoted by $[c]_Q$. The set of equivalence classes $C_{/Q}$ is considered together with the following order relation

$$[c_1]_Q \leq_{/Q} [c_2]_Q \Leftrightarrow Q(c_2) \subseteq Q(c_1),$$

Since the intents of concepts in $\uparrow\mu(t_{i+1})$ are all subsets of $\{t_{i+1}\}'$ which are closed in K^{+}, the resulting partially ordered structure, $L_{/Q}$, is isomorphic to $\uparrow\mu(t_{i+1})$ and is a complete lattice. The following algorithm (see Figure 3-1) is a generic scheme for the incremental task. It detects the three categories of concepts with the creation of the new concepts and their subsequent integration into the existing lattice structure [VM2001].

This algorithm takes a lattice and a new transaction as arguments and outputs the updated lattice using the same data structure. It includes three main computation steps. The first step is a traversal of the set L with a simultaneous calculation of the intersections between the respective concepts and the itemset of the new transaction t_{i+1} (i.e. $\{t_{i+1}\}'$), the partitioning of L into classes with respect to Q (line 3-4), and the detection of class *maximal concept* for every class $[]_Q$ with the subsequent identification of the status of the maximal element (lines 5-6). Secondly, it deals with modified concepts (lines 7-8). It updates their extents and increases the corresponding supports. Finally, it deals with the genitors (lines 10-14). It includes the creation of a new concept and the order update in the lattice [VHM2003].

```
1: procedure add-transaction (In / Out: $L$ a lattice, $t_{i+1}$ a new transaction)
2:
3: for all $c$ in $L$ do
4:   put $c$ in its class in $L_{/Q}$ w.r.t. $Q(c)$
5:   for all $[]_Q$ in $L_{/Q}$ do
6:   find $e = \max([]_Q)$
7:   if $int(e) \subseteq \{t_{i+1}\}'$ then
8:       $add(ext(e), t_{i+1})$  {$e$ is modified concept}
9: else
10:    int $\leftarrow int(e) \cap \{t_{i+1}\}'$       {$e$ is old or potential genitor}
11:      if (not (int', int) $\in L$) then
12:        { $\underline{e} \leftarrow New\text{-}Concept(ext(e) \cup t_{i+1}$, int)  {$e$ is genitor}
13:          $Update\text{-}Order(e, \underline{e})$
14:          $add(L, \underline{e})$}
```

Figure 3-1: The algorithm of an incremental approach

For example: we take the third transaction out from Table 2-1, we get Table 3-1:

TID	Itemsets
1	{*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*}
2	{*a*, *b*, *c*, *e*, *f*}
4	{*e*, *f*, *g*, *h*}
5	{*g*}
6	{*e*, *f*, *h*}
7	{*a*, *b*, *c*, *d*}
8	{*b*, *c*, *d*}
9	{*d*}

Table 3-1: An example of transaction set with 8 transactions

Following the above algorithm, we get concept lattice L_8 as Figure 3-2.

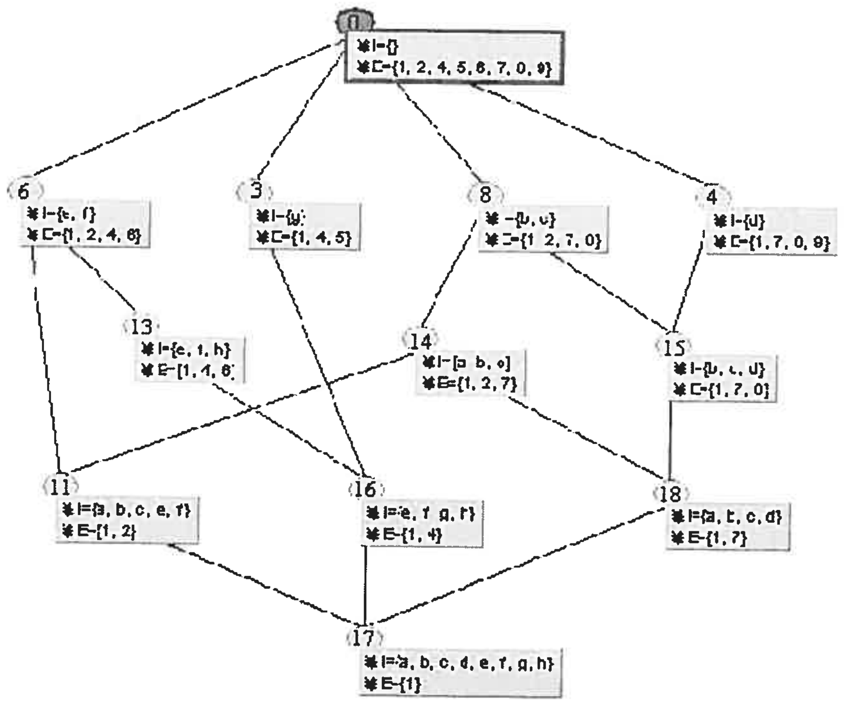

Figure 3-2: The concept lattice from Table 3-1

When adding a new transaction 3 = {*c, d, f, g, h*} to Table 3-1, transaction set T^+ is the same as Table 2-1. When inserting the transaction 3 to L_8, three categories of concepts are:

1. Old concepts = {$c_{\#14}$, $c_{\#18}$}

2. Modified concepts = {$c_{\#0}$, $c_{\#4}$, $c_{\#3}$}

3. Genitor concepts = {$c_{\#8}$, $c_{\#6}$, $c_{\#15}$, $c_{\#13}$, $c_{\#11}$, $c_{\#16}$, $c_{\#17}$}

The new concepts = {$c_{\#1}$, $c_{\#2}$, $c_{\#5}$, $c_{\#7}$, $c_{\#9}$, $c_{\#10}$, $c_{\#12}$}

Integrating new concepts into L_8, we can get new concept lattice L_9 as Figure 3-3.

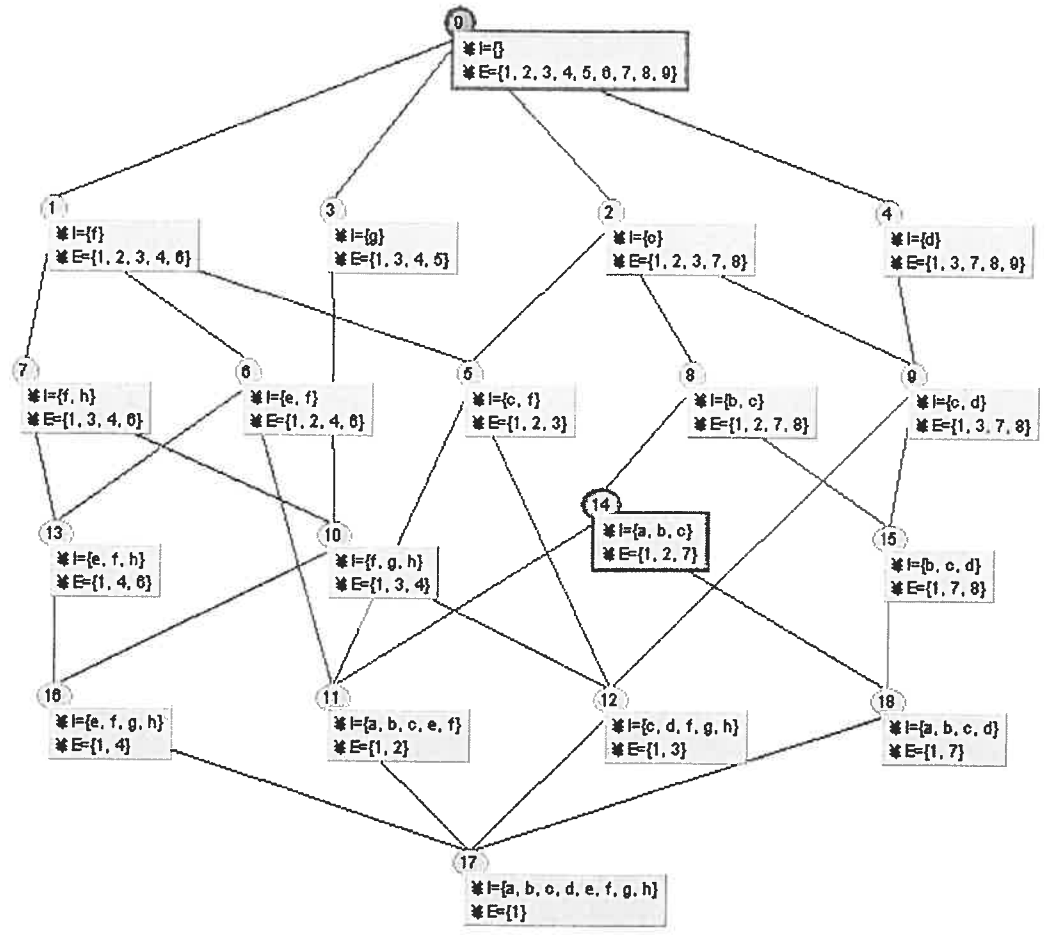

Figure 3-3: The concept lattice from Table 2-1

3.2 GALICIA family

3.2.1 GALICIA scheme

There are several differences between lattice update and closed itemsets update. Here, closed itemsets is a set of intents of concepts, and there is no order between the elements. When one updates closed itemsets, only the intents and supports are used. Lattice update should consider the ordered link among the concepts. The algorithm in Figure 3-4 can help us understand the characteristic of this approach [VM2001].

```
1:Procedure Update-Closed (In: $t_{i+1}$ a new transaction, Family-CI a set of closed itemsets)
2:Local: New-CI a set of closed itemsets
3: New-CI←∅; $I_n$← { $t_{i+1}$}'
4: for all e in Family-CI do
5: if        e. itemset ⊆ $I_n$
6: then      e. supp++;    //e is modified
7: else
8:      Y← e. itemset ∩ $I_n$;
9:      $e_y$←Lookup(Family-CI, Y)    //e is old or potential genitor
10:      if  $e_y$=NULL
11:     then $e_y$←lookup(New-CI, Y)              // e is a potential genitor
12:            if     $e_y$=NULL
13:           then   {node ← new-node(Y, e.supp++);
14:                      New-CI←New-CI ∪{node}}
15:           else
16:                   $e_y$. supp ←max(e.supp++, $e_y$. supp)
17: Family-CI←Family-CI ∪New-CI
```

Figure 3-4: Update of the closed itemsets family upon a new transaction arrival in *GALICIA* algorithm

Every closed itemset is examined in order to establish its specific category (modified, old or genitor). Modified closed itemsets simply get their support increased (line 6). Old ones remain unchanged (line 8-9). Actually, every new closed itemset is stored together with the maximal support already reached for it, i.e. since multiple closed itemsets can generate the same closed itemset with different support, the current support is the maximal support of the new closed itemset, but not yet confirmed that it is the maximum support for this closed itemset, thus each time the closed itemset is generated (line 11- 16), the support is tentatively updated. Furthermore, the storage of new closed itemsets is organized separately (*New-CI*), so that unnecessary tests can be avoided. This computation yields the correct support at the end of closed itemsets traversal. Genitors are closed itemsets with maximum support of all closed itemsets that generated new closed itemsets. This fact is strongly reinforced by an implementation that utilizes *trie* structures to reduce redundancy in both the storage and the update of the closed itemsets.

3.2.2 GALICIA-T

GALICIA-T is a version of *GALICIA* based on *tries* [K1998]. In general, the trie data structure is used to store sets of words over a finite alphabet. It is a tree structure in which letters can be assigned to edges. Each word corresponds to a unique path in the tree. All nodes can be classified into two categories. One category compiles the terminal nodes that correspond to the end of words. The other category compiles inner nodes that correspond to prefixes. Trie offers high efficiency storage. All prefixes common to two or more words are represented only once in the trie. As a consequence, a trie reduces the storage space and manipulation cost. We can regard an item as a letter and an itemset as a word.

In *GALICIA-T*, one can implement two tries to represent the closed itemsets where one trie for the current closed itemsets family(*Family-CI*), and the other for the new closed itemsets (*New-CI*). A node denotes a record with *item*, *terminal*, *successors*, *support* and *depth* fields. *Item* provides the item in node and represents transactions and individual closed itemset. *Successors* is a sorted, indexed and extendable collection for lookup, order-sensitive traversal and insertion of a new member. *Terminal* indicates whether the node is terminal, i.e. whether Y_{curr}, the current intersection between a closed itemset and I_n, represents a closed itemset. S*upport* records current node support. *Depth* is the length of the path from the root to node. The new transaction with its itemsets I_n is denoted by t_{i+1}.

The algorithm in Figure 3-5 describes the main steps of an update with a single new transaction t_{i+1}. First, it creates a new trie to store the new closed itemsets, secondly, it sorts the $\{t_{i+1}\}$', thirdly, it traverses the trie and generates new closed itemsets, finally it merges both tries.

```
1: procedure Update-Closed-Trie (In: $t_{i+1}$ a new transaction)
2: Global: Family-CI a trie of itemsets;
3: Local:  New-CI a trie of itemsets
4:
5: New-CI← new-trie (); $I_n$← sort ({$t_{i+1}$}')
6: Traversal-Intersect ($I_n$, NULL, root (Family-CI))
7: Merge (Family-CI, New-CI)
```

Figure 3-5: Trie-based update of the closed itemsets upon a new transaction arrival

The algorithm in Figure 3-6 is a recursive procedure that describes the simultaneous traversal (with detection of common elements) of two sequences of items. Each trie traversal starts from the root and goes to a terminal node. If the currently generated intersection (Y_{curr}) is a new closed itemset, then we insert it into the *New-CI* trie; if it is already in the basic *Family-CI* trie, then we update the current node support. If the length of the current intersection, $|Y_{curr}|$, equals the depth of the current node, the second case occurs. It means that the current closed itemset of the trie is a modified element of L. An intersection is finished whenever a terminal node is reached. The resulting intersection is tested for being new (line 7), if it is the case, it is added to the *New-CI* trie (line 9), else it is an existing itemset, so it corresponds to a modified. To know whether the itemset that is currently examined is the modified or it is just an old from the same equivalent class, one tests the equality of the size of the intersection to the size of the current itemset, i.e. the depth of the node in the graph of the trie. If the node has successors, the intersection goes on.

Figure 3-7 depicts the result of the entire trie traversal. On the left, the state of *Family-CI* before the insertion of transaction 3 is shown. On the middle, the *New-CI* is shown, and on the right, the situation of *Family-CI* after insertion of transaction 3 is shown [VMGM2002].

```
procedure $Traversal\text{-}Intersect$ (In: $I_n$, $Y_{curr}$ item-lists, $node$ a trie node)
Global: $Family\text{-}CI$, $New\text{-}CI$ tries of item-lists

if     ($I_n \neq$NULL) and ($I_n$.item = $node$.item)
then   $add$ ($Y_{curr}$, $I_n$.item)
if     $node$.terminal
then   n ← $lookup$($Family\text{-}CI$, $Y_{curr}$)
       if     n = NULL
       then   $update\text{-}insert$($New\text{-}CI$, $Y_{curr}$, $node$.supp++)
       else
              if     $node$.depth=|$Y_{curr}$|
              then   n.supp++
if (not $node$.terminal) or ($I_n \neq$NULL)
then
       for all n in $node$.successors do
         while ($I_n \neq$NULL) and ($I_n$.item <n.item) do
             $I_n$ ← $I_n$.next
         $Traversal\text{-}Intersect$ ($I_n$, $Y_{curr}$, n)
```

Figure 3-6: Trie-based update of the closed itemsets: single node processing

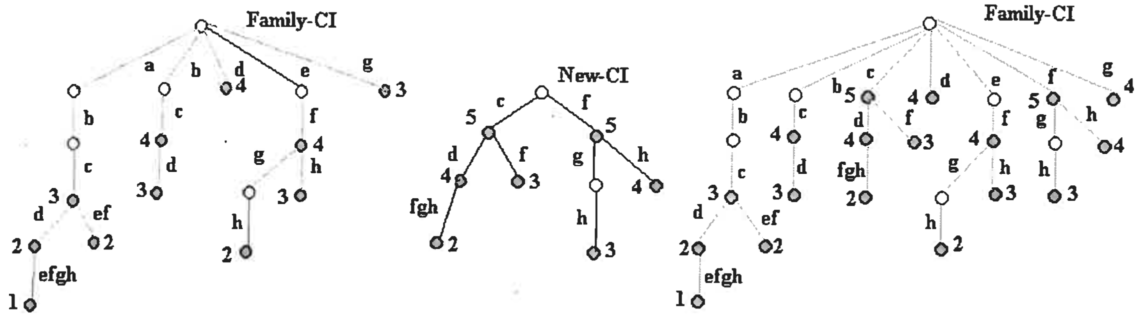

Figure 3-7: **Left**: The trie Family-CI of the closed itemsets generated from *T*.

Middle: The trie NewCI of the new closed itemsets related to transaction #3.

Right: The trie Family-CI after the intersection of transaction #3.

The following table illustrates the advancement of this algorithm on one branch of the trie, $\{a, b, c, d, e, f, g, h\}$, upon the insertion of the item list $\{c, d, f, g, h\}$.

node.item	I_n	Y_{curr}	terminal	support
a	$\{c, d, f, g, h\}$	NULL	N	-
b	$\{c, d, f, g, h\}$	NULL	N	-
c	$\{c, d, f, g, h\}$	$\{c\}$	Y	4
d	$\{d, f, g, h\}$	$\{c, d\}$	Y	3
……	……	……	……	……
h	$\{h\}$	$\{c, d, f, g, h\}$	Y	2

The first column is the items in a node, the second one is the value of I_n (available part of $\{t_{i+1}\}$'), the third column represents the current result of the intersection, and the fourth one indicates whether a node is terminal, i.e. whether the value of Y_{curr} represents a closed itemset, the fifth column records, whenever reaching a terminal node, the value of the support.

Incrementality is a major breakthrough in data mining methods and *GALICIA* is one of the first algorithms to adopt this method. The experimental results indicated its advantages for small *minsupp* cases. However, the efficiency of the algorithm is hindered by the requirement to preserve all closed itemsets. One solution of solving the dilemma is by maintaining only crucial parts of the closed itemsets (e.g. the frequent closed itemsets) and work on each particular set separately. So in our algorithm (*ILA*), we store only the part above the threshold in closed itemsets, which improves the performance.

Chapter IV Maintain iceberg lattice only with FCIs

4.1 Incremental iceberg lattice update

The algorithms mentioned in section 3.2 can be used to compute frequent closed itemsets. There are two steps in the process: the first step finds closed itemsets from transaction sets; the second step filter the frequent ones with a defined *minsupp* α. The concept lattice L contains all closed itemsets. In previously mentioned algorithms, during the process of updating concept lattice L, we ignored the value of *minsupp* α. Once given a *minsupp*, we can divide L into two parts. An upper part, denoted as $L^{\alpha}= (C^{\alpha}, \leq_K)$, where all concepts have supports greater than or equal to *minsupp* α. We call it an iceberg lattice. For example, Figure 4-1 is an iceberg lattice $L^{0.3}$ from the complete concept lattice with $\alpha= 0.3$; similarly, a lower part, denoted as $L_{\alpha}= (C_{\alpha}, \leq_K)$. In this way, we should be able to maintain (store and update) all closed itemsets, even if some closed itemsets are infrequent.

However, there are two disadvantages in these algorithms. One is that we must travel through all closed itemsets when adding a new transaction t_{i+1}. Also we must calculate the support for every closed itemset to find frequent one. These two procedures increase the computational cost. Now, we will propose a novel algorithm that only executes a few operations to maintain an existing iceberg lattice without excessive computations. These operations are based on the current iceberg lattice structure.

Given a context K, iceberg lattice L^{α} is the part above the threshold α of the complete concept lattice L of K. After adding a new transaction t_{i+1} to K, an incremental algorithm executes the same operations as with a complete lattice (*add-transaction* ( ), see section 3.1) to maintain the structural integrity of an iceberg lattice L^{α} [VMG2002]. However, there are some additional tasks, such as eliminating all concepts which become infrequent in L^+ (L^+ is constructed with the transaction set in K+ t_{i+1}) and adding some concepts which are not frequent in L, but when adding a new transaction t_{i+1} to their extents (e. g., modified concepts), they become frequent in L^+, or new frequent concepts in L^+ which are produced by non-frequent genitors in L. So the maintenance of an iceberg lattice includes not only creating new frequent concepts, but also deleting some old ones.

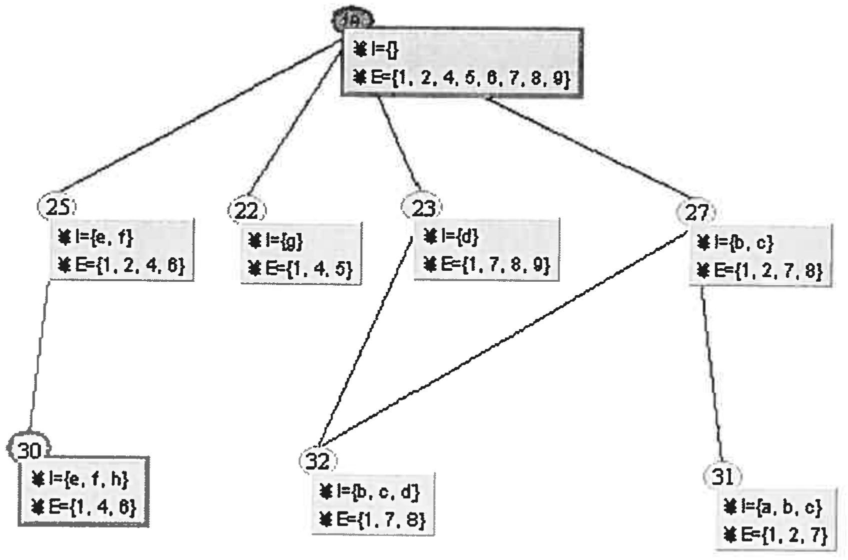

Figure 4-1: The iceberg lattice $L^{0.3}$ with $T = \{1,2,4,5,6,7,8,9\}$ & $\alpha = 0.3$

For example, Figure 4-1 is the iceberg lattice $L^{0.3}$, after adding the new transaction *3*, the new iceberg lattice $L^{0.3+}$ is as Figure 4-2 and the respective concept categories are as follows.

Old concepts = $\{c_{\#31}\}$

Modified concepts = $\{c_{\#22}, c_{\#23}\}$

Genitor concepts = $\{c_{\#25}, c_{\#27}, c_{\#30}, c_{\#32}\}$

The new concepts in $L^{0.3+} = \{c_{\#20}, c_{\#21}, c_{\#26}, c_{\#28}\}$

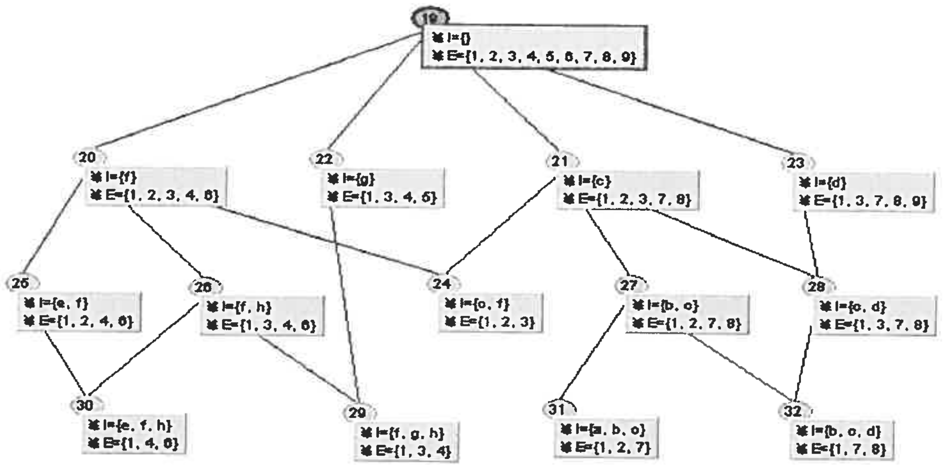

Figure 4-2: The iceberg lattice $L^{0.3+}$ with $T = \{1,2,3,4,5,6,7,8,9\}$ and α=0.3

4.2 Theoretical development

Our objective is to produce and maintain an iceberg lattice that only includes frequent closed itemsets. When adding a new transaction t_{i+1}, we should consider every element in iceberg lattice L^{α}. In this section, we present a new incremental method to maintain the integrity of the iceberg lattice.

Variable	Stands for
c	A concept in L^{α}
T	A transaction set
$\|T\|$	The number of transactions in T
$\|T^{+}\|$	The number of transactions in T+ t_{i+1}
α	Minsupp
L	Complete concept lattice constructed with transaction set T
L^{α}	Iceberg lattice constructed with transaction set T and threshold support α
L_{α}	The "lower" part of the lattice with transaction set T and threshold support α
t_{i+1}	New transaction or the TID of new transaction
$\{t_{i+1}\}$'	The itemsets of new transaction t_{i+1}
L^{+}	Complete concept lattice constructed with transaction set T+ t_{i+1}
$L^{\alpha+}$	Iceberg lattice constructed with transaction set T+ t_{i+1} and threshold support α
L_{α}^{+}	The "lower" part of the lattice with transaction set T+ t_{i+1} and threshold support α

Table 4-1: The meaning of variables in *ILA*

According to the category of a concept, we must consider the outcome of each case and subsequently prove the result.

Let c be a concept in iceberg lattice (L^{α}) and t_{i+1} is a new transaction. If c is a modified concept, then $e = (ext(c) \cup t_{i+1}, int\ (c))$ is still a frequent closed itemset, it will be in the new iceberg lattice ($L^{\alpha+}$).

Property 4-1: $\forall c \in L^{\alpha}$, if $int(c) \subseteq \{t_{i+1}\}'$ then $e = (ext(c) \cup t_{i+1}, int\ (c)) \in L^{\alpha+}$

For example, in Figure 4-1, when add the new transaction 3 = $\{c, d, f, g, h\}$, $c_{\#22}$ and $c_{\#23}$ are modified concepts, i.e. $int(c_{\#22}) \subseteq \{c, d, f, g, h\}$ and $int(c_{\#23}) \subseteq \{c, d, f, g, h\}$, so $c_{\#22}$ and $c_{\#23}$ are in Figure 4-2.

Let c be a concept in iceberg lattice (L^{α}) and t_{i+1} is a new transaction. If c is a genitor, then new generated concept $e = (ext(c) \cup t_{i+1}, int\ (c) \cap \{t_{i+1}\}')$ will be a new frequent closed itemset, it will be in $L^{\alpha+}$, but c may no longer be a frequent closed itemset.

Property 4-2: $\forall c \in L^{\alpha}$, if c is genitor, then $e = (ext(c) \cup t_{i+1}, int(c) \cap \{t_{i+1}\}') \in L^{\alpha+}$

For example, in Figure 4-1, when add the new transaction 3 = $\{c, d, f, g, h\}$, $c_{\#25}$, $c_{\#27}$, $c_{\#30}$ and $c_{\#32}$ are genitors, the new concepts they generate, $c_{\#20}$, $c_{\#21}$, $c_{\#26}$ and $c_{\#28}$, are in Figure 4-2. After checking the genitors themselves, they all keep frequent and go into Figure 4-2.

If c is an old concept, we should check if c is still a frequent closed itemset.

In the general *GALICIA* algorithm, when adding a new transaction t_{i+1}, one should update the complete concept lattice L. Although some modified concepts are not frequent in L, they may become frequent in L^{+}. Furthermore, some genitors that are non-frequent in L may generate new frequent closed itemsets in L^{+}. According to Property 4-1 and 4-2, it is relatively obvious to find concepts in $L^{\alpha+}$ that has a counterpart in L^{α}. Therefore, the main challenge in *ILA* would be to discover the concepts that are in $L^{\alpha+}$ without having a counterpart in L^{α} (such as new frequent concepts in $L^{\alpha+}$ that are produced by the modified concepts in L_{α}, or the new generated concepts whose genitors are in L_{α}). These concepts are called *hidden concepts* and denoted by $\mathbf{H}^{+}(t_{i+1})$.

Definition 4-1: $\mathbf{H}^{+}(t_{i+1}) = \{(X, Y) \in \uparrow\mu(t_{i+1}) \mid |X| \geq \alpha * |T^{+}|, |X - \{t_{i+1}\}| < \alpha * |T|\}$.

From the definition 4-1, we observe that all hidden concepts (X, Y) are frequent within transaction set T^+, they are in $L^{\alpha +}$, but the concepts $(X- \{ t_{i+1}\}, (X- \{ t_{i+1}\})'')$ are not frequent within transaction set T, they are in L_{α}.

For example, in Figure 4-2, the hidden concepts discovered after adding the new transaction $3 = \{c, d, f, g, h\}$ to Figure 4-1 are $\mathbf{H}^+(3) = \{c_{\#24}, c_{\#29}\}$, since $c_{\#24}$ and $c_{\#29}$ are generated by $c_{\#11}$and $c_{\#16}$ in Figure 3-2 those are not in Figure 4-1.

From the definition 4-1, the property hereafter follows trivially.

Property 4-3: $(X, Y) \in \mathbf{H}^+(t_{i+1})$ iff $\alpha * |T| + \alpha \le |X| < \alpha * |T| + 1$

Before we apply Property 4-3, we need to introduce another trivial property. We already know that it is obvious that within the range $[\alpha * |T| + \alpha, \alpha * |T| + 1]$, there can be at most one integer, thus we can formulate following property.

Property 4-4: $\forall n \in N, \forall \alpha \in [0..1], | [\alpha * n + \alpha, \alpha * n+1] \cap N | \le 1$

From the property 4-4, we can observe that the cardinalities of all hidden concepts are equal and there is no order among the elements of $\mathbf{H}^+(t_{i+1})$ in $L^{\alpha +}$. Suppose there is an order between c_1 & $c_2 \in \mathbf{H}^+(t_{i+1})$, then $ext(c_1) \subseteq ext(c_2)$. Since $|ext(c_1)| = |ext(c_2)|$, then $ext(c_1) = ext(c_2)$, it indicates that c_1 and c_2 are not closed itemsets, which is a contradiction. Moreover, the successors of a hidden concept are frequent concepts which the extents contain t_{i+1}. As such, the main objective of *ILA* is equivalent to computing $\mathbf{H}^+(t_{i+1})$ and linking its elements to their respective successors. These successors, called *visible concepts* and denoted by $\mathbf{V}^+(t_{i+1})$, are in an upper-set which consists of frequent concepts containing t_{i+1}.

Definition 4-2: $\mathbf{V}^+(t_{i+1})=\{(X, Y) \in \uparrow\mu(t_{i+1}) \mid |X| \ge \alpha * |T|+1\}$

For example, in Figure 4-1, the visible concepts = $\{c_{\#20}, c_{\#21}\ c_{\#19}, c_{\#26}, c_{\#22}, c_{\#28}, c_{\#23}\}$.

For each visible concept (X, Y), its counterpart in L, $(X- \{t_{i+1}\}, (X- \{t_{i+1}\})'')$ must be frequent, i.e. $(X- \{t_{i+1}\}, (X- \{t_{i+1}\})'')$ must be in L^{α}. For example, the counterparts of $c_{\#20}$ and $c_{\#22}$ in

Figure 4-2 are $c_{\#25}$ and $c_{\#22}$ in Figure 4-1 respectively. Property 4-5 shows us that all immediate successors of a hidden concept are visible concepts.

Property 4-5: $\forall c \in \mathbf{H}^{+}(t_{i+1})$, $\forall \underline{c} \in L^{+}$, if $c <^{+} \underline{c}$ then $\underline{c} \in \mathbf{V}^{+}(t_{i+1})$

For example, in Figure 4-2, $c_{\#24}$ and $c_{\#29}$ are hidden concepts, the immediate successors of $c_{\#24}$ are $c_{\#20}$ and $c_{\#21}$, the immediate successors of $c_{\#29}$ are $c_{\#22}$ and $c_{\#26}$, and all of these 4 concepts, $c_{\#20}$, $c_{\#21}$, $c_{\#22}$ and $c_{\#26}$, are visible concepts.

Thus, in order to discover hidden concepts, we may examine visible concepts in $\mathbf{V}^{+}(t_{i+1})$ and generate their immediate predecessors in $\mathbf{H}^{+}(t_{i+1})$.

We can prove that for each comparable pair of concepts c_1, c_2 in $\uparrow\mu(t_{i+1})$ (i.e., $c_1 \leq^{+} c_2$), there is an attribute (item) i in $\{t_{i+1}\}'$ (even in $int(c_1)$ -$int(c_2)$), such that c_1 is the meet of c_2 and the attribute-concept of i, $\nu(i)$.

Property 4-6: $\forall c_1, c_2 \in \uparrow\mu(t_{i+1})$, if $c_1 \leq^{+} c_2$, then $\forall\ i \in int(c_1) - int(c_2) \mid c_1 = \nu(i) \wedge c_2$.

For example, in Figure 4-2, $int(c_{\#29}) - int(c_{\#26}) = \{g\}$, $c_{\#29} = \nu(g) \wedge c_{\#26} = c_{\#22} \wedge c_{\#26}$

In order to find the lower covers $cov^{l}(c)$ of a concept c in $\mathbf{V}^{+}(t_{i+1})$, one should examine the set of its meets with appropriate attribute-concepts. The primitive operator of such an attribute i is denoted by $\wedge_i$: $\wedge_i c = \nu(i) \wedge c$. We should now be able to calculate the extent and intent of $\wedge_i c$. The former is $ext(\wedge_i c) = ext(c) \cap ext(\nu(i))$, which equals to $ext(c) \cap \{i\}'$; and the latter, the intent of $\wedge_i c$, is more expensive to obtain in the general. In order to avoid this costly calculation, we derive the following property.

Property 4-7: $\forall c_1, c_2 \in \mathbf{V}^{+}(t_{i+1})$, s. t $c_1 < c_2$, $int(c_1) - int(c_2) = \{ i \in \{t_{i+1}\}' \mid \wedge_i c_2 = c_1 \}$

For example, in Figure 4-2, $c_{\#23}$, $c_{\#28}$ are visible concepts and $c_{\#28}$ is a predecessor of $c_{\#23}$, the intent difference between $c_{\#28}$ and $c_{\#23}$ is c which belongs to $\{t_{i+1}\}'$, and $c_{\#28}$ is the meet of $c_{\#23}$ and the attribute-concept of c, $c_{\#21}$.

Property 4- 7 indicates that if c_1 precedes c_2, then the attributes in the difference of their intents are exactly those of that $c_1 = \nu(i) \wedge c_2$.

We can obtain the intent of the concept $\wedge_i c_2$ by adding all the attributes $\underline{i}$ to *int* (c_2), such that *ext* $(c_2) \cap \{\underline{i}\}' =$ *ext* $(c_2) \cap \{i\}'$. Given a concept, since this procedure fits all of its immediate predecessors, we can find several subsets of attributes that lead to generate valid intents. On the other hand, the attributes that are not in those subsets will generate only partial intents since the corresponding concept $\wedge_i c_2$ is not an immediate predecessor. Furthermore, the set of attributes that generate hidden lower covers for a given concept c is denoted by $gen_h(c)$. So the main problem in *ILA* is to determine the attributes in $gen_h(c)$.

So far we do not have a direct way to determine this attribute set. However, what we can do is to conduct some tests using a set of candidate attributes. In each test, we compare the size of intersection *ext* $(c_2) \cap \{\underline{i}\}'$ to a constant value that is calculated based on the *minsupp* α and $|T|$ (see Property 4-3). Because the computation of acquiring the intersection of extent is an expensive task, we should reduce the number of candidates, primarily by eliminating those attributes for which $\wedge_i c_2$ is certainly not a hidden concept.

When we are calculating the lower covers $\wedge_i c$ for a given concept c in $\mathbf{V}^+(t_{i+1})$ with an attribute i in $\{t_{i+1}\}'$- $int(c)$, the resulting concepts, which is a predecessor of c, will fall into one and only one of these three cases (i.e., orthogonal): (*i*) $\wedge_i c \in \mathbf{V}^+(t_{i+1})$ (a visible concept), (*ii*) $\wedge_i c \in \mathbf{H}^+(t_{i+1})$ (a hidden concept) and (*iii*) $\wedge_i c \in L_\alpha^+$ (non frequent concept). In order to avoid generating non immediate predecessors in $\mathbf{H}^+(t_{i+1})$, we should find the hidden concepts using the visible concepts that are their actual *upper covers*. For cases (*ii)* and (*iii)*, it is difficult to avoid those without intersection computation, thus we only consider the case (*i)*.

We now design a method to prevent the concepts in $\mathbf{V}^+(t_{i+1})$ from producing their known predecessors in the same set $\mathbf{V}^+(t_{i+1})$. A mechanism of intent propagation from concepts in $\downarrow c \cap \mathbf{V}^+(t_{i+1})$ to c is adopted, namely, with a bottom-up and breadth-first traversal of $\mathbf{V}^+(t_{i+1})$. This traversal helps to proliferate the cumulated intents of all predecessors using a simple lookup of the known lower covers. It also requires us to add an additional field to the data structure representing a concept to record the set $T(c)$.

Definition 4-3: $T(c) = \cup_{\underline{c} \in \downarrow c \cap \mathbf{V}^{+}(t_{i+1})} int(\underline{c})$

Given a visible concept c, from definition 4-3, we know that $T(c)$ is the union of intents that come from all predecessors of c and these predecessors belong to $\mathbf{V}^{+}(t_{i+1})$. The $T(c)$ prevents c from generating its predecessors that are already in $\mathbf{V}^{+}(t_{i+1})$ because the generated concepts have support greater than $|T|*\alpha$, therefore it is impossible for them to be in $\mathbf{H}^{+}(t_{i+1})$. For example, in Figure 4-2, $T(c_{\#20}) = \{f, h\}$. $T(c_{\#20})$ prevents $c_{\#20}$ from generating $c_{\#26}$, because $c_{\#26}$ is a predecessor of $c_{\#20}$ and in $\mathbf{V}^{+}(3)$.

In fact, we can obtain the $T(c)$ for any visible concept before computing a hidden candidate, but we prefer to compute it during the traversal of $\mathbf{V}^{+}(t_{i+1})$ for the sake of efficiency. If a hidden concept which is a non-immediate predecessor of c has been found by a visible predecessor of c, the attributes of this hidden concept can be used to extend the $T(c)$. For example, in Figure 4-2, before calculating $T(c_{\#20})$, the visible predecessor of $c_{\#20}$ has generated a hidden concept, $c_{\#29}$, so the attributes of $c_{\#29}$ can be combined to $T(c_{\#20})$, now $T(c_{\#20}) = \{f, g, h\}$.

Given a concept c, we define an additional set, denoted by T^{h}, as follows.
Definition 4-4: $T^{h}: \mathbf{V}^{+}(t_{i+1}) \rightarrow 2^{I}$,

$$T^{h}(c) = \{i \mid i \in int(e), e \in \downarrow c \cap \mathbf{V}^{+}(t_{i+1}) + cov^{l}(e) \cap \mathbf{H}^{+}(t_{i+1}) - cov^{l}(c) \cap \mathbf{H}^{+}(t_{i+1})\}$$

Definition 4-4 tells us that the set $T^{h}(c)$ keeps track of all the attributes of visible predecessors of c and attributes of hidden concepts generated by visible predecessors of c except those attributes that generate immediate hidden concepts of c. For example, in Figure 4-2, the visible predecessors of $c_{\#20}$ is $c_{\#26}$, and $c_{\#26}$ generated a hidden concept $c_{\#29}$, in fact, $cov^{l}(c_{\#20}) \cap \mathbf{H}^{+}(3) = \varnothing$, so $T^{h}(c_{\#20})$ is $\{f, g, h\}$ which is the union of $int(c_{\#26})$ and $int(c_{\#29})$.

For a given visible concept c, $T^{h}(c)$ represents the maximal set of attributes that can be reduced from $\{t_{i+1}\}'$. We can compute $T^{h}(c)$ from the values of T^{h} on immediate visible predecessors of c and the sets of hidden concept generating attributes for the same concepts.

Property 4-8: $T^h(c) = \cup_{e \in cov^l(c) \cap V^+(t_{i+1})} (T^h(e) \cup h(e))$

Here, $h(e)$ represents the hidden concepts that are immediate predecessors of e. For example, in Figure 4-2, $c_{\#26}$ does not have any visible predecessor, so $T^h(c_{\#26}) = \varnothing$, however, $c_{\#20}$ has a visible predecessor $c_{\#26}$ which generated the hidden concept $c_{\#29}$, so $T^h(c_{\#20}) = T^h(c_{\#26}) \cup h(c_{\#26}) = \{f, g, h\}$.

For each visible concept c, we defined a set of attributes, called the *domain* of c and denoted by *domain*(c), is $\{t_{i+1}\}' - T^h(c)$, which should be examined for generating the hidden concepts.

Definition 4-4: $\forall c \in V^+(t_{i+1})$, $domain(c) = \{t_{i+1}\}' - T^h(c)$.

For example, in Figure 4-2, add transaction 3 = $\{c, d, f, g, h\}$, $domain(c_{\#20}) = \{c, d, f, g, h\} - T^h(c_{\#20}) = \{c, d, f, g, h\} - \{f, g, h\} = \{c, d\}$.

In this chapter, we proposed a new method to implement incremental iceberg update and described related theories while deduced the proofs for these theories. In this method, the most difficult and expensive step is discovering the hidden concepts for visible concepts. In order to improve the efficiency, we applied some mechanisms, such as a bottom-up and breadth-first traversal of $V^+(t_{i+1})$, to minimize the candidate attributes. In the next chapter, we will apply these theories to our incremental update algorithm.

Chapter V The incremental method

When a new transaction is being inserted into the context, the new method described in Chapter IV can be transformed into an algorithmic procedure, named Iceberg Lattice Algorithm (*ILA*), which incrementally updates an iceberg lattice.

5.1 The description of Iceberg Lattice Algorithm

Iceberg Lattice Algorithm (see Figure 5-1) maintains the structure of an iceberg lattice. It has two main parts. In the first part, *ILA* calls the procedure *Add_Transaction_IceBg* ( )(line 3) to insert a new transaction t_{i+1} to an iceberg lattice L^{α} where the iceberg lattice is regarded as an complete lattice. This procedure, showed in Figure 5-2, is a modification of algorithm 1 (*Add_Object* ( ) in [VRM2003]) which updates the lattice incrementally. In this part, all concepts in L^{α} will be checked, every modified concept will be updated (adds t_{i+1} to its *extent* and *intent* keeps unchanged); every genitor will generate a new concept (its *extent* is the combination of genitor's *extent* with t_{i+1} and its *intent* is the intersection between genitor's *intent* and $\{t_{i+1}\}$') and the old concepts will be unaffected. The second part (line 4-10) is more important, complicated and expensive one in *ILA*. It accomplishes the core task of *ILA*.

```
1: procedure Update_Iceberg_Lattice (In:  $L^{\alpha}$ an iceberg lattice,  $T$ an indexed set of
2:                                         transactions, $t_{i+1}$ a new transaction)
3: Add_Transaction_IceBg ($L^{\alpha}$, $t_{i+1}$)
4: for all $c$ in $L^{\alpha}$ do
5:   if $s(c) < \alpha$ then
6:     Drop ($L^{\alpha}$, $c$)
7: Sort ($\mathbf{V}^{+}(t_{i+1})$)                //in descending order of the Intent size
8: $\mathbf{H}^{+}(t_{i+1}) \leftarrow$ Find_Lower_Covers ($\mathbf{V}^{+}(t_{i+1})$, $T$, $t_{i+1}$)
9: $L^{\alpha+} \leftarrow$ add ($L^{\alpha}$, $\mathbf{H}^{+}(t_{i+1})$)
10: return $L^{\alpha+}$
```

Figure 5-1: The algorithm of Iceberg Lattice Algorithm

At the beginning, it filters out and drops all infrequent old concepts and genitors in L^{α} (lines 4-6), then it sorts concepts in $\mathbf{V}^{+}(t_{i+1})$ according to the descending order of intent size (line 7) and calls *Find_Lower_Covers* () to compute immediate predecessors of visible concepts, i.e. discover hidden concepts in $\mathbf{H}^+(t_{i+1})$ (line 8). The purpose of sorting the concepts in $\mathbf{V}^+(t_{i+1})$ is to ensure the application of the bottom-up and breadth-first traversal mechanisms in finding hidden concepts. Finally, the new discovered hidden concepts are integrated into L^α to form $L^{\alpha+}$ (lines 9-10).

```
1: procedure Add_Transaction_IceBg (In/ Out: $L^\alpha$ an iceberg lattice, $t_{i+1}$ a new transaction)
2:
3: for all $c$ in $L^\alpha$ do
4:    put $c$ in its class in $L^\alpha{}_{/Q}$ w.r.t. $Q(c)$ // create index according to int(c) $\cap \{t_{i+1}\}$'
5: for all $[]_Q$ in $L^\alpha{}_{/Q}$ do
6:    find $c = max([]_Q)$
7:    if $int(c) \subseteq \{t_{i+1}\}$' then {
8:       Add (ext (c), t)                    // c is modified concept
9:       Add ($\mathbf{V}^+(t_{i+1})$, c)}
10: else
11:    int ← int (c) $\cap \{t_{i+1}\}$'         // c is old or potential genitor
12:  if (not (int', int) $\in L^\alpha$) then {
13:    e ←New_Concept (ext (c) $\cup$t, int)  //  c is genitor
14:    Update_Order (c, e)
15:    Add ($L^\alpha$,  e)
16:    Add ($\mathbf{V}^+(t_{i+1})$, e)}
```
Figure 5-2: The algorithm of inserting a new transaction to an iceberg lattice

The procedure, *Add_Transaction_IceBg* ( ), takes an iceberg lattice L^α and a new transaction t_{i+1} as arguments and outputs an updated iceberg lattice $L^{\alpha'}$ which is between L^α and $L^{\alpha+}$ ($L^{\alpha'}$ contains all concepts in $L^{\alpha+}$ except all hidden concepts, Figure 5-3 is an example). Here, we use the same data structure to represent both the initial and the resulting lattices. There are three main computation steps within this procedure. The first one is a traversal of the set L^α

with a simultaneous calculation of the intersection between the intent of the respective concept and the itemset of the new transaction t_{i+1}, i.e. $\{t_{i+1}\}'$. It then partitions L^{α} into classes with respect to Q (lines 3-4) and finally detects the class maximal concept for every class $[]_Q$ with the subsequent identification of the status of the maximal elements (lines 5-6). The second step deals with modified concepts (lines 7-9). It updates their extent (adds t_{i+1} to the *extent*), increases the corresponding supports and adds the modified concepts to visible concept set (i.e., $V^{+}(t_{i+1})$) for computing their lower covers. The final step deals with the genitors (lines 11-16). It includes the creation of a new concept, where its *extent* is the combination of genitor's *extent* with t_{i+1} and its *intent* is the intersection between genitor's *intent* and $\{t_{i+1}\}'$, the insertion of generated concept to iceberg lattice, the order update in iceberg lattice and the insertion of generated concept to visible concept set.

For example, given *minsupp* α=0.3, when we have finished dealing with transactions in Table 3-1, we have obtained iceberg lattice L^{α}showed in Figure 4-1, it includes 8 concepts, $c_{\#30}$ = ({1,4,6}, {*e*, *f*, *h*}), $c_{\#31}$ = ({1,2,7}, {*a*, *b*, *c*}), $c_{\#25}$ = ({1,2,4,6}, {*e*, *f*}), $c_{\#22}$ = ({1,4,5}, {*g*}), $c_{\#27}$ = ({1,2,7,8}, {*b*, *c*}), $c_{\#32}$ = ({1,7,8}, {*b*, *c*, *d*}), $c_{\#23}$ = ({1,7,8,9}, {*d*}) and $c_{\#19}$ = ({1, 2,4,5,6,7,8,9}, $\varnothing$).

Consider now the insertion of the transaction 3 = {*c*, *d*, *f*, *g*, *h*} into L^{α} with the procedure *Add_Transaction_IceBg* (). The content of $L^{\alpha}{}_{/Q}$ after performing the first step (line 3-4) is shown in the following list, together with the indication of class maximum and its respective status (*mod* for modified and *gen* for genitor) using the format ($Q(c)$, $[]_Q$, Max($[]_Q$), Status): ({*g*}, $c_{\#22}$, $c_{\#22}$, *mod*), ({*d*}, $c_{\#23}$, $c_{\#23}$, *mod*), ({*f*, *h*}, $c_{\#30}$, $c_{\#30}$, *gen*), ({*c*}, {$c_{\#31}$, $c_{\#27}$}, $c_{\#27}$, *gen*), ({*f*}, $c_{\#25}$, $c_{\#25}$, *gen*), ({*c*, *d*}, $c_{\#32}$, $c_{\#32}$, *gen*). For modified concepts, add t_{i+1} to their respective *extent*, we get $c_{\#22}$ = ({1,3,4,5}, {*g*}), $c_{\#23}$ = ({1,3,7,8,9}, {*d*}) and $c_{\#19}$ = ({1,2,3,4,5,6,7,8,9}, $\varnothing$). Consequently, the new concepts created at the end of the traversal of Q_{classes} are $c_{\#26}$ = ({1,3,4,6}, {*f*, *h*}), $c_{\#21}$ = ({1,2,3,7,8}, {*c*}), $c_{\#20}$ = ({1,2,3,4,6}, {*f*}) and $c_{\#28}$ = ({1,3,7,8}, {*c*, *d*}). After finishing this step, we get the Figure 5-3. Now, the visible concept set is $V^{+}(3)$ = {$c_{\#22}$, $c_{\#23}$, $c_{\#19}$, $c_{\#26}$, $c_{\#21}$, $c_{\#20}$, $c_{\#28}$}.

5.2 Discovery of lower covers

In Figure 5-1, we call the procedure *Find_Lower_Covers* () (Figure 5-4). The purpose of this procedure is to generate the lower covers of visible concepts ($\mathbf{V}^{+}(t_{i+1})$) using the respective *domain*. This algorithm deploys a bottom-up, width-first traversal mechanism of the iceberg $L^{\alpha'}$.

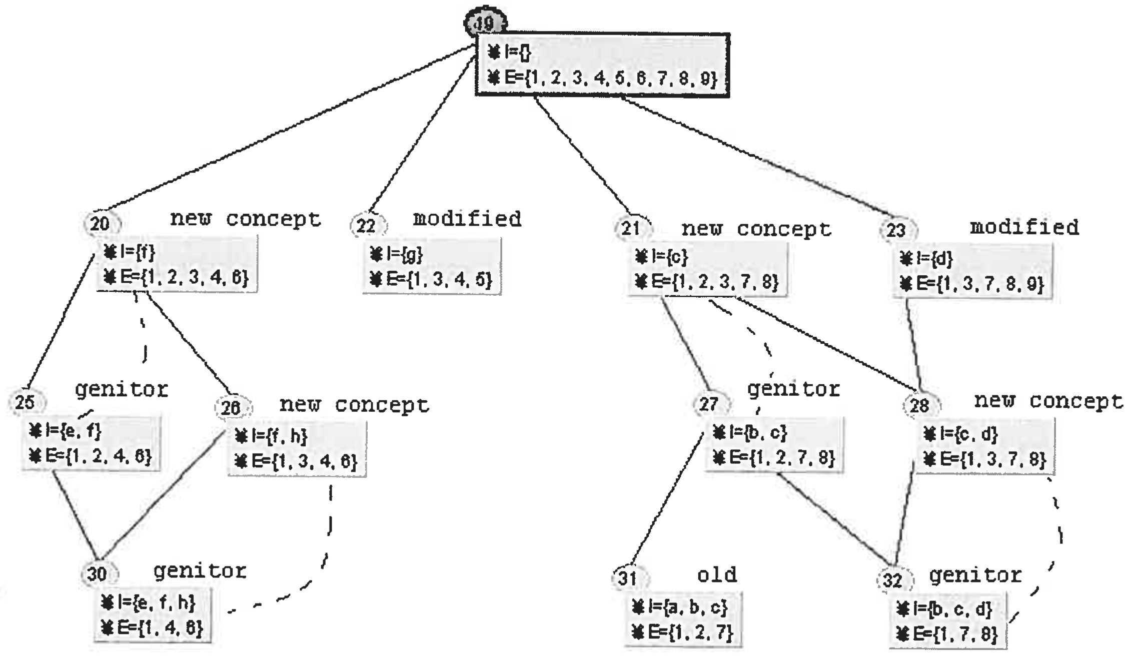

Figure 5-3: The iceberg lattice $L^{0.3}$ from Table 2-1 (α=0.3) before finding hidden concepts

The first step sorts the concepts in $\mathbf{V}^{+}(t_{i+1})$ according to the descending order of intent size (line 9) and it ensures that the concepts are tested in an order that represents a linear extension of the lattice order. Secondly, in the main process, each concept *c* in $\mathbf{V}^{+}(t_{i+1})$ is checked to generate its hidden concepts. This process includes the computation of the frequent extent intersections *Extent* that are stored together with the generating-attributes *Candidates* in the hidden concept candidate set. Finally, at the end of the processing of *c*, (*Extent, Candidates*) in candidate set are used to generate the effective hidden concepts that will be added to the global hidden concept set $\mathbf{H}^{+}(t_{i+1})$.

```
Procedure $Find\_Lower\_Covers$ (In/Out: $V^{+}(t_{i+1})$, $T$ an indexed set of
                                        transactions, $t_{i+1}$ the new transaction)

local: $Hidden$: set of concepts
local: $T^{h}$, $Domain$, $h$: set of attributes
local: $Extent$: set of transactions
local: $Candidate$: set of pair $(X, Y)$

$Hidden \leftarrow \varnothing$
$Sort\ (V^{+}(t_{i+1}))$          {in descending order of the Intent size}
for all $c$ in $V^{+}(t_{i+1})$ do
    $Candidate \leftarrow \varnothing$; $T^{h} \leftarrow Int(c)$; $h \leftarrow \varnothing$
    for all $e \in Cov^{l}(c)$ do
        $T^{h} \leftarrow T^{h} \cup T^{h}(e) \cup h(e)$
    $Domain \leftarrow \{t_{i+1}\}' - T^{h}$
    while not $Domain = \varnothing$ do
        $i \leftarrow extract\_first\ (Domain)$; $Extent \leftarrow Ext(c) \cap \{i\}'$
        if $\alpha * |T| + \alpha \leq |Extent| < \alpha * |T| + 1$ then
            $k \leftarrow Look\_Up\ (Extent, Hidden)$
            if $k \neq$ NULL then
                $Cov^{l}(c) \leftarrow Cov^{l}(c) \cup \{k\}$; $h \leftarrow h \cup Int(k)$; $Domain \leftarrow Domain - Int(k)$
            else
                $can \leftarrow Look\_Up\ (Extent, Candidate)$
                if $can \neq$ NULL then
                    $can.Y \leftarrow can.Y \cup \{i\}$
                else
                    $can \leftarrow (Extent, (Int(c) \cup \{i\}))$; $Candidate \leftarrow Candidate \cup \{can\}$
            $h \leftarrow h \cup \{i\}$
    for all $can \in Candidate$ do
        $e \leftarrow New\_Concept\ (can.X, can.Y)$; $Cov^{l}(c) \leftarrow Cov^{l}(c) \cup \{e\}$
        $Hiddens \leftarrow Hiddens \cup \{e\}$
    $T^{h}(c) \leftarrow T^{h}$; $gen_{h}(c) \leftarrow h$
```

Figure 5-4: The algorithm for generating lower covers

For a visible concept c in $\mathbf{V}^+(t_{i+1})$, now we explain the process of discovering its hidden concepts in detail. The initialization step (line 11) sets the initial value of T^h as *int* (c), then the set T^h is found by applying the propagation mechanism of intents. It makes good use of the T^h values of the predecessors of c (lines 12-13), this step adds all attributes of visible predecessors of c and their hidden concepts to T^h. So far, the T^h represents the maximal set of attributes that can be removed from $\{t_{i+1}\}$' and the *domain* (c) (potential generating attributes) is determined (line 14). The next step is to discover the hidden concepts of c one by one with every attribute i extracted from *domain* (c) if *domain* (c) is not empty (lines 15-27). Once the extent of $\wedge_i c$ is computed (line 16) and it is proven to be frequent (line 17), the algorithm will check whether the extent of $\wedge_i c$ existed in *hidden* set (line 18), if it is found in *hidden* set (line 19), it means that this extent has already been generated by another concept, then the corresponding concept e is simply added to the hidden predecessors set of c (line 20). Furthermore, the intent of e is used to update both the set of generating-attributes $gen_h(c)$ and the *domain*(c) (line 20). When the extent of $\wedge_i c$ is not generated by another concept (i.e., the extent of $\wedge_i c$ is not in *hidden* set), the algorithm will check whether this extent existed in the *candidate* set discovered by c (line 22). If it existed, it means c has generated a candidate hidden concept which has the same extent, we will update the intent of the candidate found by adding i to its intent (line 24), if not, we create a new candidate with the pair of (*ext* ($\wedge_i c$), *int* (c)$\cup\{i\}$) and add it to *candidate* set of c (line 26). On the next step, new hidden concepts are created based on the elements in *candidate* set (line 29). These new hidden concepts are then linked to their upper cover (line 29) and added to the hidden concept set (line 30). At the end of the algorithm, all attributes of the hidden predecessors of c are used to update $T^h(c)$ and $gen_h(c)$ (line 30), these two attribute sets will be used to discover hidden concepts for the successors of c.

In order to illustrate the hidden concept computation, let us observe $c_{\#26}$ and $c_{\#22}$ when insert the transaction 3 = $\{c, d, f, g, h\}$. According to lines 10-14 in Figure 5-3, *domain* ($c_{\#26}$) = $\{c, d, f, g, h\}$-$\{f, h\}$=$\{c, d, g\}$. The intersection between *ext* ($c_{\#26}$) and $\{c\}$' is $\{1,3\}$, it leads to an infrequent extent ($|\{1,3\}|<3$). It is the same situation between *ext* ($c_{\#26}$) and $\{d\}$'. The intersection between *ext* ($c_{\#26}$) and $\{g\}$' is $\{1,3,4\}$ and it is frequent ($|\{1,3,4\}|=3$). However, before dealing with $c_{\#26}$, both the $\mathbf{H}^{+}(3)$ and *candidate* set are empty, the intersection {1,3,4} is neither in $\mathbf{H}^{+}(3)$ nor in *candidate* (line 25), ({1,3,4}, *{f, g, h}*) is then added to *Candidate* set (line 26). Now, the *domain* ($c_{\#26}$) is empty, a new hidden concept $c_{\#29}$ is created from the candidate element ({1,3,4}, *{f, g, h}*) (line 29) and add it to $\mathbf{H}^{+}(3)$ (line 30). Now $T^{h}(c_{\#26}) = \{f, h\}$, $gen_h(c_{\#26})=\{g\}$. For $c_{\#22}$, *domain* ($c_{\#22}$) = $\{c, d, f, g, h\}$- $\{g\}$= $\{c, d, f, h\}$. The intersection between *ext* ($c_{\#22}$) and {*c*}' is {1,3} and it leads to an infrequent extent ($|\{1,3\}|<3$). It is the same situation between *ext* ($c_{\#22}$) and {*d*}'. The intersection between *ext* ($c_{\#22}$) and {*f*}' is {1,3,4}, it is frequent ($|\{1,3,4\}| =3$). However, $c_{\#29}$ (generated by $c_{\#26}$) in $\mathbf{H}^{+}(3)$ has the resulting extent. So $c_{\#29}$ is added to the hidden predecessors set of $c_{\#22}$ and $h(c_{\#22}) = int(c_{\#29}) = \{f, g, h\}$(line 20). Now, update *domain* ($c_{\#22}$), since *h* is in both *int* ($c_{\#29}$) and *domain* ($c_{\#22}$), it is eliminated from *domain* ($c_{\#22}$), so the *domain* ($c_{\#22}$) is empty, we have finished the process of $c_{\#22}$.

5.3 Detailed example

In order to demonstrate the iceberg lattice algorithm, let us take a detail example of context given in section 2.2.

The first part of the algorithm in Figure 5-1 is performed (insertion of the transaction #3, see section 5.1). The result showed in Figure 5-3. The second part starts by dropping out infrequent concepts that could be old concepts or genitors. In this example, all old concepts and genitors are still frequent. In the next step, we deal only with the visible concepts in $\mathbf{V}^{+}(3)$= $\{c_{\#22}, c_{\#23}, c_{\#26}, c_{\#21}, c_{\#20}, c_{\#28}, c_{\#19}\}$. They are first sorted according to their intent size and then their frequent immediate predecessors are calculated. Sorting $\mathbf{V}^{+}(3)$ in descending order of the intent size, we get the following concept order $\{c_{\#26}, c_{\#28}, c_{\#21}, c_{\#22}, c_{\#23}, c_{\#20}, c_{\#19}\}$.

Table 5-1 displays the execution of the algorithm in Figure 5-3 on all visible concepts with their corresponding *domain*. For each row, the table should be read as follows: the first column displays each domain attribute *i*, the second column provides the value of the extent of $\wedge_i c$, the third one is the status of the intersection result (*fre* for frequent, *unfre* for infrequent), the fourth column presents the set $gen_h(c)$, the fifth one indicates the evolution of

Attribute (item)	*Extent*	*Status*	*gen*$_h$(*c*)	*Domain*(*c*)	*Candidates*	*Hidden concepts* ($\mathbf{H}^+(3)$)
Process of concept $c_{\#26}$						
{*c*}	{1,3}	*unfre*	∅	{*d*, *g*}	∅	∅
{*d*}	{1,3}	*unfre*	∅	{*g*}	∅	∅
{*g*}	{1,3,4}	*fre*	{g}	∅	({1,3,4}, {*f*, *g*, *h*})	$c_{\#29}$
Process of concept $c_{\#28}$						
{*f*}	{1,3}	*unfre*	∅	{*g*, *h*}	∅	$c_{\#29}$
{*g*}	{1,3}	*unfre*	∅	{*h*}	∅	$c_{\#29}$
{*h*}	{1,3}	*unfre*	∅	∅	∅	$c_{\#29}$
Process of concept $c_{\#21}$						
{*f*}	{1,2,3}	*fre*	{*f*}	{*g*, *h*}	({1,2,3}, {*c*, *f*})	$c_{\#29}$
{*g*}	{1,3}	*unfre*	∅	{h}	({1,2,3}, {*c*, *f*})	$c_{\#29}$
{*h*}	{1,3}	*unfre*	∅	∅	({1,2,3}, {*c*, *f*})	$c_{\#29}$, $c_{\#24}$
Process of concept $c_{\#22}$						
{*c*}	{1,3}	*unfre*	∅	{*d*, *f*, *h*}	∅	$c_{\#29}$, $c_{\#24}$
{*d*}	{1,3}	*unfre*	∅	{*f*, *h*}	∅	$c_{\#29}$, $c_{\#24}$
{*f*}	{1,3,4}	*fre*	{*f*, *h*}	∅	∅	$c_{\#29}$, $c_{\#24}$
{*h*}	Cancelled by line 20 of algorithm 5-3 (*Domain* ← *Domain* –*Int* (*e*))					
Process of concept $c_{\#23}$						
{*f*}	{1,3}	*unfre*	∅	{*g*, *h*}	∅	$c_{\#29}$, $c_{\#24}$
{*g*}	{1,3}	*unfre*	∅	{*h*}	∅	$c_{\#29}$, $c_{\#24}$
{*h*}	{1,3}	*unfre*	∅	∅	∅	$c_{\#29}$, $c_{\#24}$
Process of concept $c_{\#20}$						
{*c*}	{1,2,3}	*fre*	{*f*}	{*d*}	∅	$c_{\#29}$, $c_{\#24}$
{*d*}	{1,3}	*unfre*	{*f*}	∅	∅	$c_{\#29}$, $c_{\#24}$
Since the *Domain* of $c_{\#19}$ is empty, so it does not generate any *hidden concept*.						

Table 5-1: The trace of Algorithm 5-4 when a new transaction is added

the *domain* variable, the sixth column records and updates the *candidate* set, the last column shows the content of the hidden concepts set ($\mathbf{H}^{+}(3)$). In section 5.2, we have described how to deal with $c_{\#26}$ and $c_{\#22}$ in detail.

Figure 5-5 illustrates the main process of the Table 5-1.

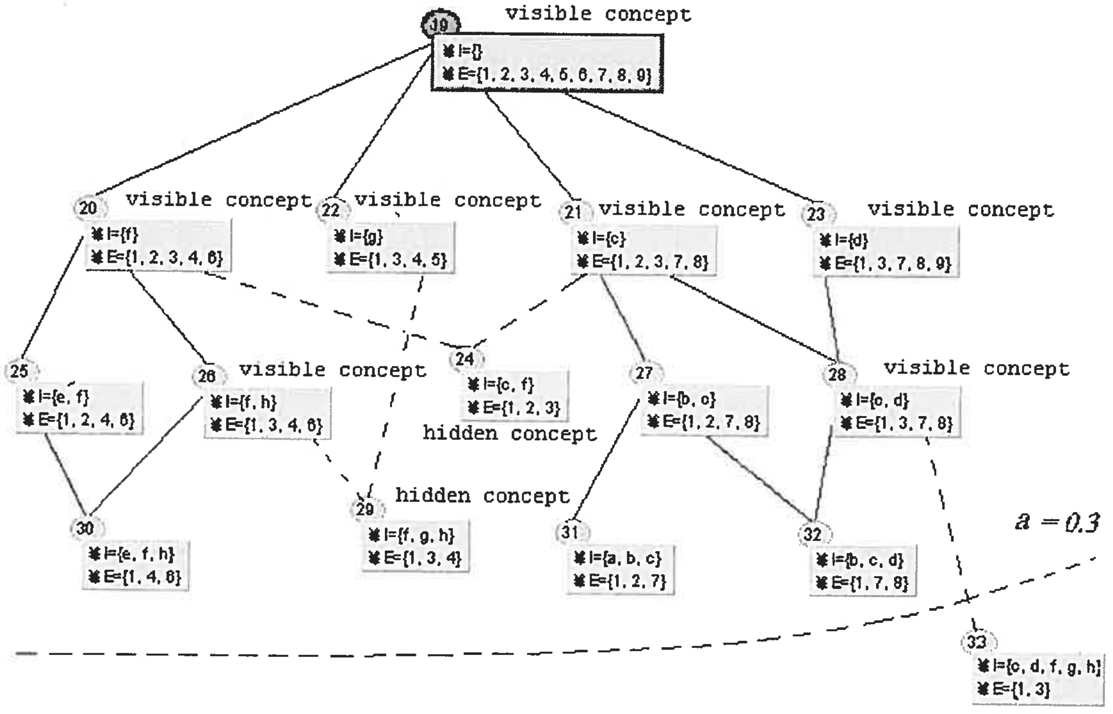

Figure 5-5: Illustration of the discovery of hidden concepts

During the process of concept $c_{\#26}$, we found the hidden concept $c_{\#29} = (\{1, 3, 4\}, \{f, g, h\})$. While dealing with the concept $c_{\#22}$, we found that $c_{\#29}$ was a hidden predecessor of $c_{\#22}$. We found the hidden concept $c_{\#24} = (\{1, 2, 3\}, \{c, f\})$ during the process of concept $c_{\#21}$. Similarly, we found that $c_{\#24}$ was a hidden predecessor of $c_{\#20}$. There was no hidden concept to be found during the process of concepts $c_{\#28}$ and $c_{\#23}$. $c_{\#28}$ only generated a infrequent concept ($c_{\#33}$ in Figure 5-5). The *domain* of $c_{\#19}$ is empty, so it is impossible to generate any hidden concept. Finally, the hidden concepts discovered at the end of the algorithm 5-1 are $\mathbf{H}^{+}(3) = \{c_{\#29}, c_{\#24}\}$.

The new iceberg lattice $L^{\alpha+}$ obtained from L^{α} and the transaction #3 is showed as in Figure 4-2, all concepts are: $c_{\#30}$=({1,4,6}, {e, f, h}), $c_{\#31}$=({1,2,7}, {a, b, c}), $c_{\#25}$=({1,2,4,6}, {e, f}), $c_{\#22}$ = ({1,3,4,5}, {g}), $c_{\#27}$ = ({1,2,7,8}, {b, c}), $c_{\#32}$ = ({1,7,8}, {b, c, d}), $c_{\#23}$ = ({1,3,7,8,9}, {d}), $c_{\#26}$=({1,3,4,6}, {f, h}), $c_{\#21}$=({1,2,3,7,8}, {c}), $c_{\#20}$=({1,2,3,4,6}, {f}), $c_{\#28}$ = ({1,3,7,8}, {c, d}), $c_{\#29}$ = ({1,3,4}, {f, g, h}), $c_{\#24}$ = ({1,2,3}, {c, f}), $c_{\#19}$ = ({1,2,3,4,5,6,7,8,9}, $\varnothing$)}.

5.4 Complexity issues

There are a lot of factors that influence the global complexity of Iceberg Lattice Algorithm. In this section, we will make some explanation.

(1) The number of transactions in a transaction set $|T|$

The number of transactions in a transaction set represents the size of *DB*. This is trivial.

(2) *minsupp*

Given a transaction set, as the *minsupp* value decreases, more closed itemsets become frequent, hence more concepts are in iceberg lattice, obviously, it will increases the global complexity of the algorithm.

(3) The total number of items which represent $|I|$

It is trivial that more items will increase the complexity in computation of *domain* and candidate hidden concepts.

(4) Maximal number of lower covers of a concept in $L^{\alpha+}$

This represents the complexity of discovering the hidden concepts.

The basic notations are summarized in table 5-2.We can assess the complexity of Algorithm 5-4 as follows. First, with respect to the concept intent sizes, we sort the set $\mathbf{V}^{+}(t_{i+1})$ in descending order (line 9), in this step, we just need to compare the intents size, so it can be accomplished in a linear time, i.e. O (l). Now we consider the cost of traversal of the set $V^+(t_{i+1})$ (lines 10). We know that the worst case is when all concepts in $L^{\alpha+}$ are also in $V^+(t_{i+1})$, so it takes O (l) concept examinations to discover hidden concepts.

Variable	Stands for
m	The number of attributes $\mid I \mid$
l	The number of concepts in $L^{\alpha+}$
k	The number of transactions, $\mid T \mid$
$\Delta(l)$	The different number of $L^{\alpha+}$ and L^{α} $\mid L^{\alpha+} - L^{\alpha} \mid$
$d(L^{\alpha+})$	Maximal number of lower covers of a concept in $L^{\alpha+}$, max $(\mid Cov^l(c) \mid) \mid c \in L^{\alpha+}$

Table 5-2: The meaning of variables in complexity issues

The cost for discovering hidden concepts can be further divided into three additional parts: *domain* computation (line 11-14), candidates-related computation (line 15-27) and finding the new hidden concepts (line 28-30).

Maximal number of lower covers of a concept in $L^{\alpha+}$ is $d(L^{\alpha+})$, the algorithm will perform $O(d(L^{\alpha+}))$ concept examinations (line 12) to compute the *domain* for the concept (line 11-14) and each examination computes the union of attribute sets (line 13) which can be done in $O(m)$. In the worst case, every attribute in I will form a lower cover of a concept and we conclude that $O(d(L^{\alpha+})$ has the upper bound $O(m)$, hence the computation of the *domain* costs $O(m^2)$.

The candidates-related computation (line 15-27) has four additional components. The first one is the traversal of *domain* (line 15). In the worst case, we know that all attributes in I are in *domain*, so this step costs $O(m)$. Second, the computation of extent intersection (line 16) will costs $O(k)$, since the maximum values of both extent of concept and $\{t_{i+1}\}'$ are k. Third, when all new concepts (i.e., the difference between $L^{\alpha+}$ and L^{α}) are hidden concepts and the TID of every transaction has been the extent of a hidden concept, then the discovery of hidden concept set ($\mathbf{H}^+(t_{i+1})$) (line 18) can be executed in $O(\Delta(l)k)$. Finally, the discovery of hidden concept candidates (line 22) will also costs $O(mk)$ since in the worst case, the TID of every transaction has been the extent of a candidate hidden concept and every attribute in I will generate a candidate hidden concept. However, we can apply trie-based structure to the sets of extents (hidden concepts and candidate), regardless of the scale of the trie, this structure provides us a very efficient discovery which costs linear in the number of transactions. Thus the cost of the lookup operations go back to $O(k)$ and the candidates-related cost will be $O(mk)$.

Given a concept c, we know that the number of new hidden concepts generated by c is limited by attribute number, so the creation of the hidden concepts (line 29) can be executed in constant time and dealing with new hidden concepts (line 30) costs $O(m)$. In accordance with the analysis above, the total complexity of the Algorithm 5-4 is bounded by $O(m(m+k)l)$.

In order to update the iceberg lattice, the Algorithm 5-1 (see Figure 5-1) performs two traversals. For the first traversal, the algorithm *Add_Transaction_IceBg*() is called, its complexity is $O(\Delta(l)k^2+l(k+m))$ (see [VRM2003]). During the second traversal, to guarantee the integrity of the iceberg lattice, i.e. eliminating infrequent concepts and discovering the hidden concepts, the algorithm performs additional steps. Line 4-6 is the first step that eliminates the infrequent concepts. It can be done in $O(lm)$ since the algorithm checks the support of each concept (maximum number of concepts is l when all concepts in $L^{\alpha+}$ are checked). If needed, infrequent concepts are removed. In the second step, the Algorithm5-4 is used to find the potential hidden concepts in $\mathbf{H}^+(t_{i+1})$. As described earlier, its complexity is $O(m(m + k)l)$. Consequently, the global complexity of reconstructing an iceberg for a single insertion is $O(\Delta(l)k^2+m(k+m)l)$. The time complexity of *ILA* is the same as that of *GALICIA*.

Chapter VI Implementation and experiments

In this chapter, we will describe some issues about implementations and experiments of Iceberg Lattice Algorithm.

6.1 Experimental results

We conducted a set of tests to compare the running performance between *ILA* and *GALICIA* series algorithms. The reasons that we chose *GALICIA* algorithms to compare with *ILA* are: (1) both algorithms are based on concept lattice theory, (2) both algorithms compute frequent closed itemsets, and (3) *ILA* is an enhanced version of the basic *GALICIA* algorithm. The experiments were performed on a 1.3 GHz AMD TB processor with 1.2 GB main memory, running Windows 2000. Both algorithms were tested in JBuilder 5 environment, and a Java implementation is used.

Two synthetic transaction sets, namely Mushroom and T25I10D10 [UCI] were used in the experiments. Mushroom is a strongly correlated transaction set and had 8,124 transactions over 119 items. This transaction set generated 227,699 closed itemsets, 3630 of them had support larger than 0.1. The second transaction set, T25I10D10K, is relatively sparse. It includes 10,000 transactions over 1,000 items where each transaction has 25 items on average, and the average size of maximal potentially frequent itemsets is 10. This transaction set generated 3,530,786 closed itemsets, 23,852 of them were of support larger than 0.005. Table 6-1 and Table 6-2 display the detailed results when increasing subsets of the entire transaction sets respectively. The fourth column of these two tables illustrates approximate multiple between the number of closed itemsets (*CIs*) and the number of frequent closed itemsets (*FCIs*).

6.2 CPU time

In these experiments, we carried out two types of tests to evaluate the performance of *GALICIA-M* and *ILA*. In the first type, we regarded two algorithms as being procedures of a batch process and focused on the CPU time for processing the whole transaction set. In the second type, we focused on the CPU time for processing transactions incrementally. In order to provide a better view about the trends that lay behind each algorithm, we recorded the results for transaction sets of variable size. Thus, both transaction sets have been divided into increments of fixed size, 2,000 transactions for both of T25I10D10K and Mushroom. For each increment, the tests have been carried out with a fixed absolute support threshold for *ILA* (50 for T25I10D10K and Mushroom).

Size of Transaction Set	Number of *CIs*	Number of *FCIs* ($minsupp$ =0.1)	$\frac{NumberofCIs}{NumberofFCIs}$
2,000	57,586	1,519	~ 40
4,000	101,772	2,021	~50
6,000	150,137	2,935	~50
8,124	227,699	3,630	~60

Table 6-1: Mushroom, total *CIs* and *FCIs* with α=0.1

Size of Transaction Set	Number of *CIs*	Number of *FCIs* ($minsupp$ = 0.005)	$\frac{NumberofCIs}{NumberofFCIs}$
2,000	281,209	544	~550
4,000	626,114	2,275	~300
6,000	1,562,211	6,977	~250
8,000	2,479,770	14,701	~200
10,000	3,530,786	23,852	~150

Table 6-2: T25I10D10K, total *CIs* and *FCIs* with α=0.005

Two types of comparisons have been carried out. The first type (see Figure 6-1 and Figure 6-3) focused on comparing the performance of both algorithms as batch procedures. Referring to Table 6-1 and Table 6-2, we know that the number of closed itensets is greater than that of frequent closed itemsets when the *minsupp* is not 0, the results of the tests clearly showed the advantage of *ILA* over *GALICIA-M*, especially when the transaction set is growing large.

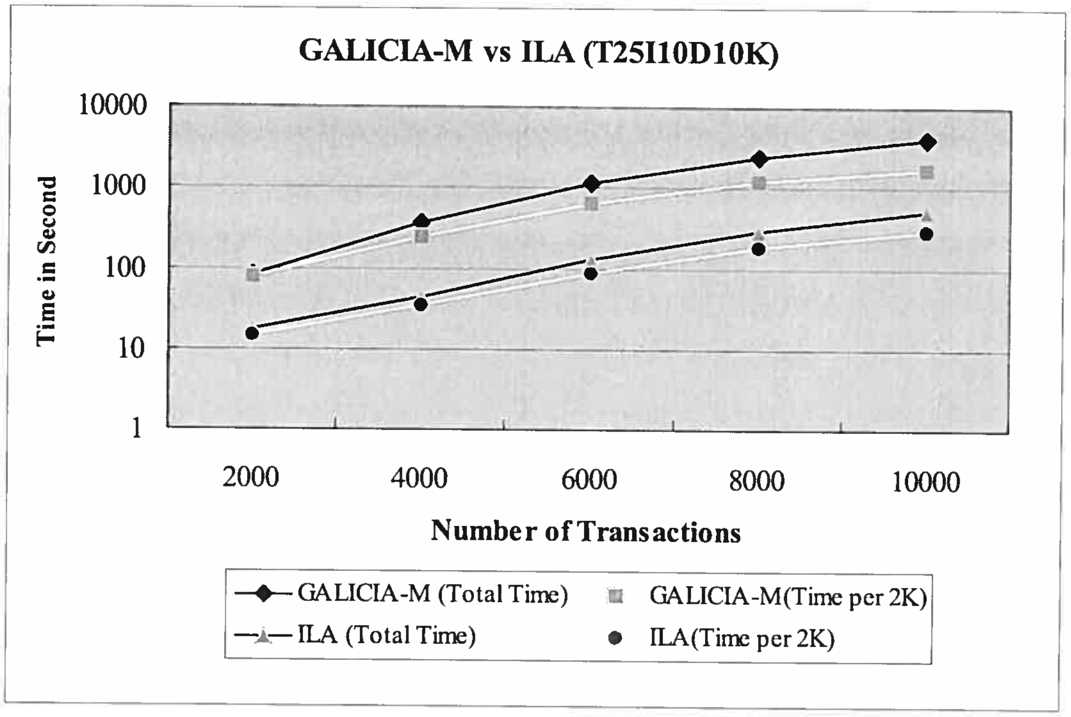

Figure 6-1: CPU time for *GALICIA-M* and *ILA*

(First type of tests - T25I10D10K and *minsupp=50*)

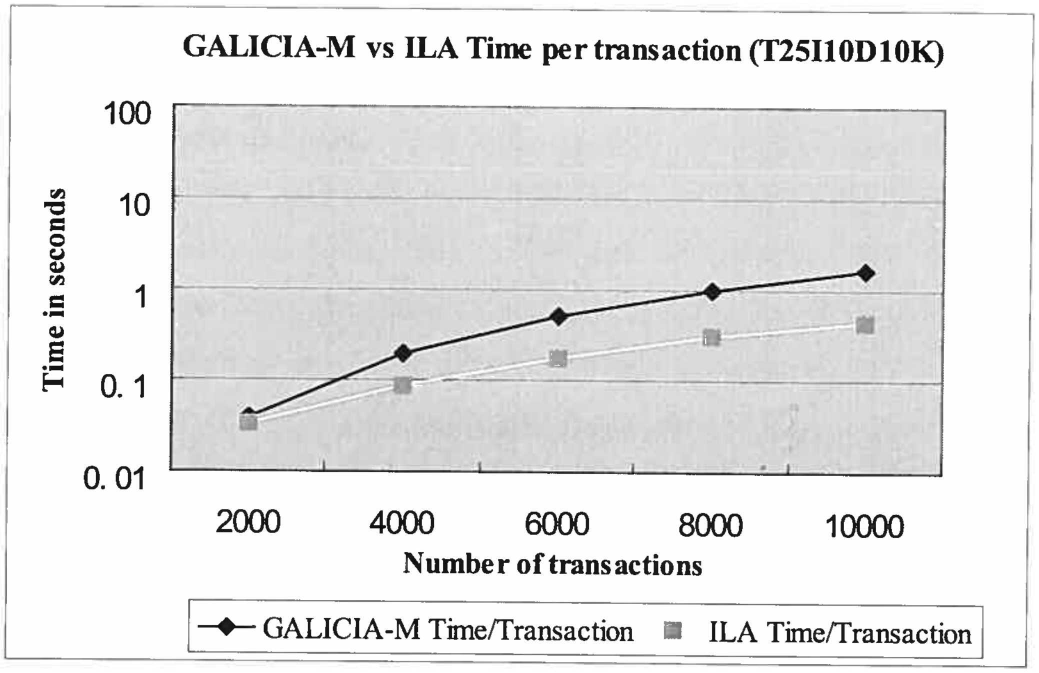

Figure 6-2: Average CPU time for *GALICIA-M* and *ILA*

(Second type of tests - T25I10D10K)

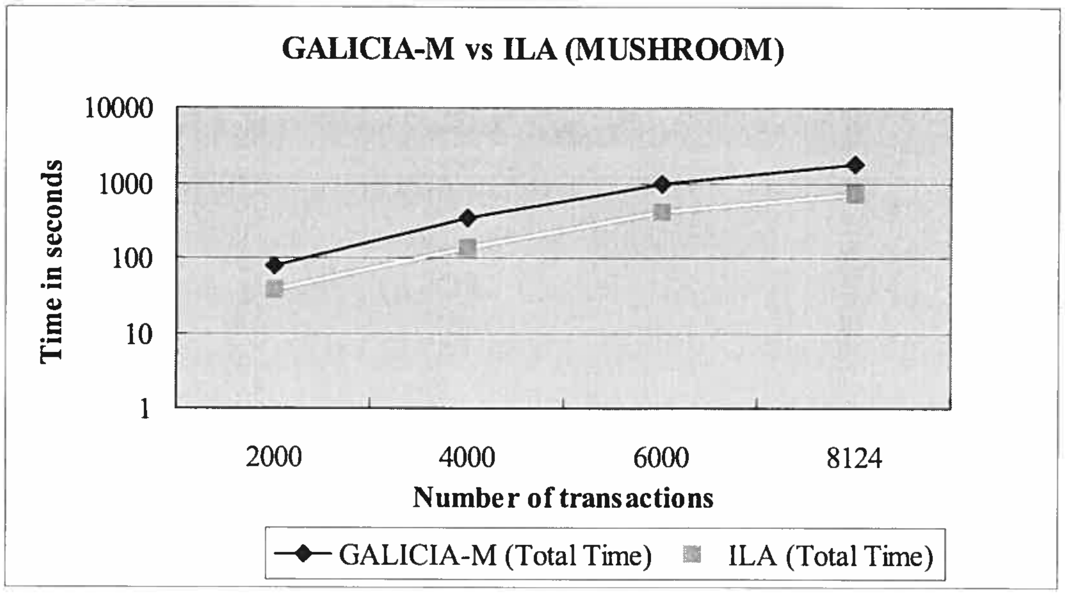

Figure 6-3: CPU time for *GALICIA-M* and *ILA*

(First type of tests - Mushroom and *minsupp=50*)

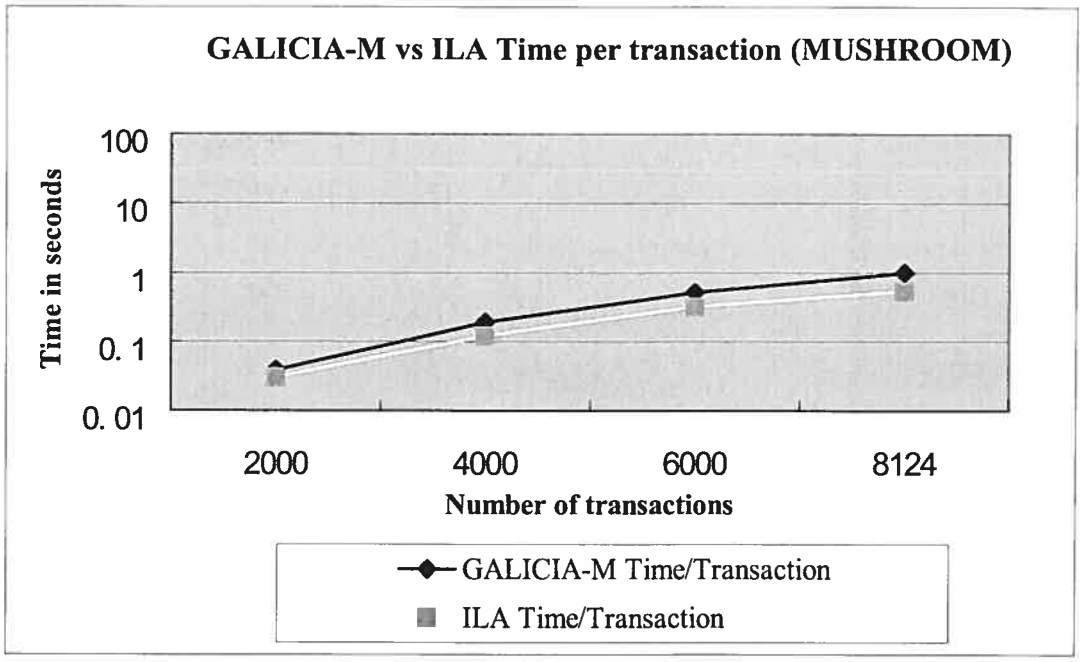

Figure 6-4: Average CPU time for *GALICIA-M* and *ILA*

(Second type of tests – Mushroom)

Figure 6-2 and Figure 6-4 showed the response time of the *GALICIA-M* and *ILA* for the second type of tests. In these tests, we took into account the average CPU time for integrating a new transaction each time. These diagrams show that the average CPU time of *ILA* is lower than that of *GALICIA-M*. As we have mentioned, the number of closed itemsets is always greater than that of frequent closed itemsets when the *minsupp* is not 0. When adding a new transaction, the execution time spent to traverse the complete lattice (includind all closed itemsets) is longer than the time to traverse an iceberg lattice. Especially on T25I10D10K due to the fact that it is a sparse dataset, the same number of transactions and the same *minsupp* will produce less frequent closed itemsets. Taken as a whole, these experimental results indicate that the benefits of techniques in *ILA* are more obvious with sparse transaction sets than with dense ones.

However, the improvement of the CPU time showed from Figure 6-1 to Figure 6-4 is not as dramatic as the proportion between the number of closed itemsets and frequent closed itemsets. The key reason is that we must find the lower covers for visible concepts. This is an expensive task. For example, in T25I10D10K, when the transaction number is increased from 6000 to 8000, the number of frequent closed itemsets increases from 6,977 to 14,701, i.e. 7,724 new frequent closed itemsets must be found. Assume that 90% of them are generated by genitors, so 10% of them (about 700) are hidden concepts; furthermore, in order to get these hidden concepts, we estimate that we need to traverse 500 visible concepts for inserting one transaction and 1,000,000 for 2000 transactions. In T25I10D10K, each transaction has on average 25 items, assume 15 items can be removed from the *domain* of each visible concept, hence at least 10,000,000 intersection calculations are required to find these hidden concepts.

In Table 6-2, after processing 10,000 transactions, the total number of frequent closed itemsets is 23,852. Suppose we add a new transaction (10001st transaction), according to our estimation, it will generate six new frequent closed itemsets, genitors generate four of them and two of them are hidden concepts. In order to get these two hidden concepts, we should check all visible concepts. In this case, we estimate that 10% of frequent closed itemsets are visible concepts, namely, about 2400 concepts must be traversed to discover these two hidden concepts.

The efficiency of the *GALICIA* series approach is clearly obstructed by the necessity of maintaing the whole set of frequent closed itemsets [VMGM2002]. In *ILA*, although traversing iceberg can help save execution time and improve the performance, it does not make our algorithm more efficient and scalable since the total execution time still remains high. When mining dynamic transaction sets with *ILA*, the execution time is taken in different mining steps, such as: (*i*) adding a new transaction to the iceberg, (*ii*) removing infrequent closed itemsets from the iceberg and (*iii*) finding frequent lower covers for visible concepts, i.e. discovering the hidden concepts.

Maintaining iceberg lattice indicates that it obviously improves the performance of step (*i*) which add a new transaction to the iceberg. In this step, we update modified concepts and generate new concepts. Unfortunately, step (*ii*) - removing infrequent closed itemsets, and especially step (*iii*) – discovering hidden concepts, are still very expensive. The main issue now is to establish a proper trade-off of the cost between the different steps.

Chapter VII Conclusions and future work

In this chapter, we summarize our thesis and introduce trends for future work on incremental algorithms.

7.1 Conclusions

The objective of this thesis is to propose an efficient, incremental method for mining frequent closed itemsets in dynamic transaction sets. *ILA* (Iceberg Lattice Algorithm) is an enhanced application of the *GALICIA* approach. When given a *minsupp*, it only needs to maintain the upper most part of the closed itemsets, i.e. frequent closed itemsets. Furthermore, it traverses only iceberg when a new transaction is added.

We observed that it is easy to obtain new concepts in a new iceberg lattice that have a counterpart in a previous iceberg. Therefore, the main problem with maintaining an iceberg in lattice-based incremental algorithms is that we should discover the new concepts without having a counterpart in previous iceberg that we called hidden concepts. The main challenge then becomes the question of computing hidden concepts efficiently. After studying the features of hidden concepts, we found that every hidden concept must be covered by one or more concepts (we called them visible concepts) from above, i.e. every hidden concept must be an immediate successor of a visible concept. Furthermore, we presented and proved useful properties for both visible concepts and hidden concepts. Based on these characteristics, we proposed a new method to maintain an iceberg lattice. Namely, by generating all lower covers of visible concepts. In order to improve the efficiency of discovering lower covers, we applied the mechanism of bottom-up and breadth-first traversal of visible concepts. To take advantage of this new approach, we transformed it into an algorithmic procedure, Iceberg Lattice Algorithm (*ILA*). Furthermore, we implemented the algorithm and conducted a set of tests to compare its performance with *GALICIA* algorithm.

ILA is an effective and useful tool intended for mining transaction sets. It overcomes the weaknesses that existed in non-incremental algorithms, such as level-wise algorithms and those that are based on concept connection. These algorithms either traverse the transaction set repeatedly in order to obtain the frequent closed itemsets, or find closed itemsets from transaction sets first, then calculate the frequent closed itemsets based on a defined *minsupp*. The major drawback in these algorithms is that whenever a new transaction is added, they cannot use the previous result and must completely re-establish the result for the new transaction set. Also, in the *GALICIA* approach, since the complete concept lattice contains all closed itemsets, we have to maintain (store and update) all closed itemsets, even if some are not frequent. *ILA* is an incremental algorithm that constructs an iceberg lattice that only includes the frequent closed itemsets, and finds new set of frequent closed itemsets based on the previous frequent closed itemsets, thus successfully avoiding any excessive computation.

7.2 Future work

Our experiments show that some concepts oscillate between the iceberg and the infrequent part of the lattice, because when adding successive transactions, these concepts may become repeatedly frequent then infrequent, then again frequent and so on.

An interesting extension of our work could be the definition of a lattice zone below the iceberg lattice where those concepts lay. The optional size of the zone (in depth of the graph of the corresponding partially ordered set) and the way it will be maintained are two concrete questions to be answered. The first one will be answered by experimental studies and the second one by some theoretical investigation.

Beside further research in the theoretical aspect, we discuss some practical applications. A variation of iceberg lattice is the most lower part of complete lattice (i.e. sub-semi-lattice). This can be a further topic for data mining. This type of lattice can be used in information retrieval in library browsing.

In the application of information retrieval in library browsing, we regard the ID of a book or document as a TID, and the title, author, subject, index term, etc as items. We can establish a binary relationship between TID collection and items. Then we can construct a sub-semi-lattice to retrieve a list of document according to the quarry terms. If more query terms are selected (the intent of concept is greater), i.e. the request is more precise, fewer documents are retrieved (the extent of the concept is lesser).

Appendix

1 Proof of the properties

Property 4-1: $\forall c \in L^{\alpha}$, if $int(c) \subseteq \{t_{i+1}\}'$ then $e = (ext(c) \cup t_{i+1}, int(c)) \in L^{\alpha+}$

Proof: $c \in L^{\alpha} \Rightarrow |ext(c)| \geq minsupp*|T| \Rightarrow |ext(c)|+1 \geq minsupp*|T|+1$,

$int(c) \subseteq \{t_{i+1}\}' \Rightarrow e = (ext(c) \cup t_{i+1}, int(c))$

$\Rightarrow |ext(e)| = |ext(c)|+1$

$\Rightarrow |ext(e)| \geq minsupp*|T|+1$,

$minsupp \leq 1 \Rightarrow |ext(e)| \geq minsupp*|T| + minsupp$

$$\Rightarrow \frac{|ext(e)|}{(|T|+1)} \geq minsupp$$

$\Rightarrow e \in L^{\alpha+}$.

Property 4-2: $\forall c \in L^{\alpha}$, if c is genitor, then $e = (ext(c) \cup t_{i+1}, int(c) \cap \{t_{i+1}\}') \in L^{\alpha+}$

Proof is the same as property 4-1.

Property 4-3: $(X, Y) \in \mathbf{H}^{+}(t_{i+1})$ iff $\alpha*|T|+\alpha \leq |X| \leq \alpha*|T|+1$

Proof is trivial.

Property 4-5: $\forall c \in \mathbf{H}^{+}(t_{i+1})$, $\forall \underline{c} \in L^{+}$, if $c <^{+} \underline{c}$ then $\underline{c} \in \mathbf{V}^{+}(t_{i+1})$

Proof: $c \in \mathbf{H}^{+}(t_{i+1}) \Rightarrow |ext(c)| \geq \alpha*(|T|+1)$,

$c <^{+} \underline{c} \Rightarrow |ext(\underline{c})| \geq |ext(c)|+1 \geq \alpha*(|T|+1)+1 \geq \alpha*|T|+1$

$c \in \mathbf{H}^{+}(t_{i+1})$ & $c <^{+} \underline{c} \Rightarrow t_{i+1} \in ext(\underline{c}) \Rightarrow \underline{c} \in \uparrow\mu(t_{i+1})$

according to the definition of $\mathbf{V}^{+}(t_{i+1})$, we get the property 4-5.

Property 4-6: $\forall c_1, c_2 \in \uparrow\mu(t_{i+1})$, if $c_1 \leq^{+} c_2$, then $\forall\, i \in int(c_1) - int(c_2) \mid c_1 = \nu(i) \wedge c_2$.

Proof: $c_1, c_2 \in \uparrow\mu(t_{i+1})$ & $i \in int(c_1) - int(c_2) \Rightarrow i \in \{t_{i+1}\}'$ & $\nu(i) \in \uparrow\mu(t_{i+1})$

Assume c_1 is not the smallest intent that includes both $int(c_2)$ and i, i.e. $int(c_1) \neq (int(c_2) \cup \{i\})''$, then c_1 is not closed concept, which is a contradiction.

Property 4-7: $\forall c_1, c_2 \in \mathbf{V}^{+}(t_{i+1})$, s. t $c_1 < c_2$, $int(c_1) - int(c_2) = \{ a \in \{t_{i+1}\}' \mid \wedge_i c_2 = c_1 \}$

From Property 4-6, we know $int(c_1) - int(c_2) \subseteq \{ i \in \{t_{i+1}\}' \mid \wedge_i c_2 = c_1$. now, we just need to prove $\wedge_i c_2 = c_1 \Rightarrow i \in int(c_1) - int(c_2)$.

Proof: $c_1 < c_2 \Rightarrow ext(c_1) \subset ext(c_2)$

If $\wedge_i c_2 = c_1 \Rightarrow int(c_1) = int(c_2) \cup int(v(i))$ & $ext(c_1) = ext(c_2) \cap ext(v(i))$,

Assume $i \notin int(c_1) - int(c_2) \Rightarrow a \in int(c_1)$ & $i \in int(c_2)$

$\Rightarrow ext(v(i) \supseteq ext(c_2)$

$\Rightarrow ext(c_1) = ext(c_2)$, a contradiction.

2 System architecture

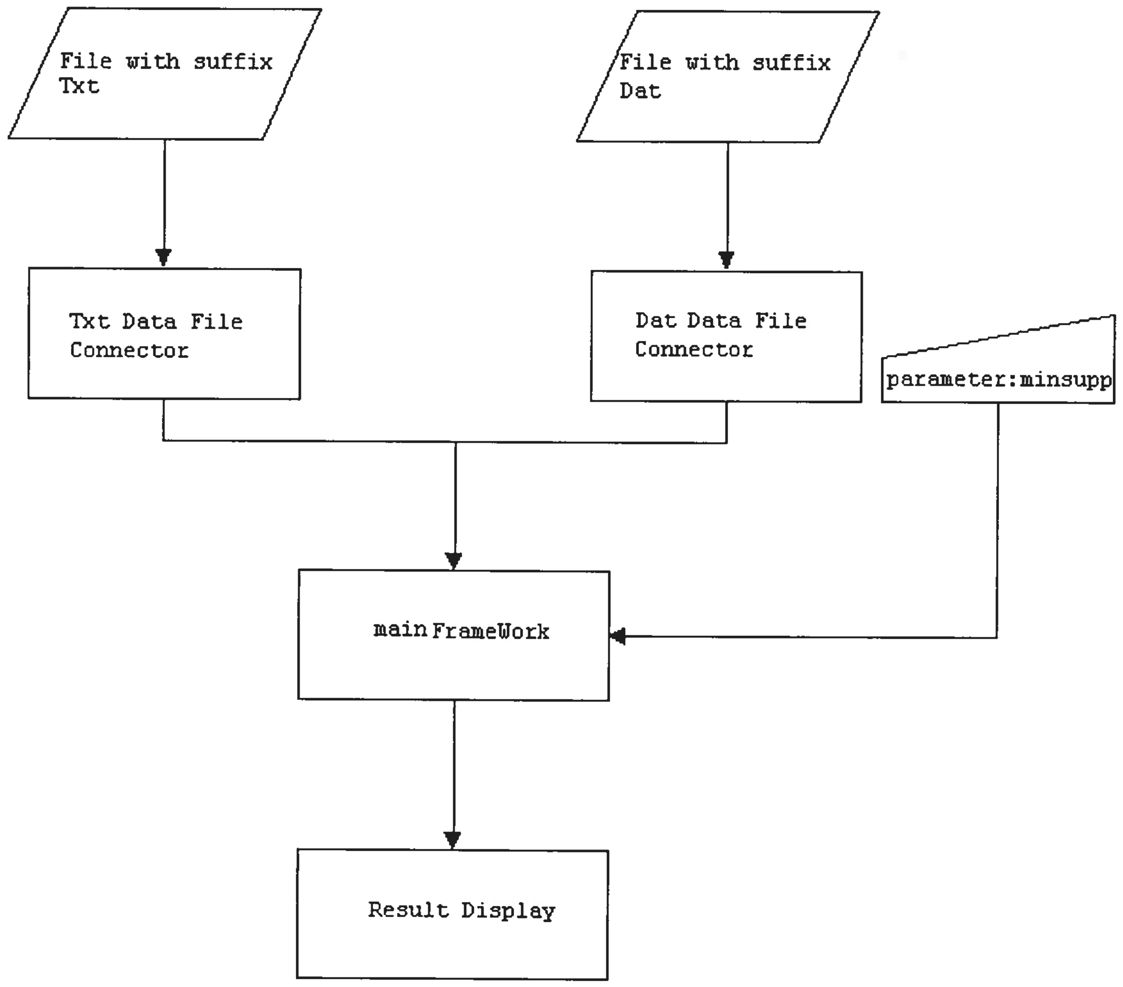

Figure A-1: System architecture

The system consists of three main parts. The first is a connector that connects the transaction set to the corresponding processor according to the transaction set formation (i.e., the suffix name of the data file). The second part is the core of the processor. It reads and processes the transaction one by one, according to the parameter *minsupp* to incrementally maintain the iceberg lattice. The third part is a display tool, which shows the result on the screen.

3 Class diagram

We define the following classes. They are *Concept*, *Transaction*, *VectorQuickSort*, *Common and MainFrame*. Figure A-2 is the Class Diagram.

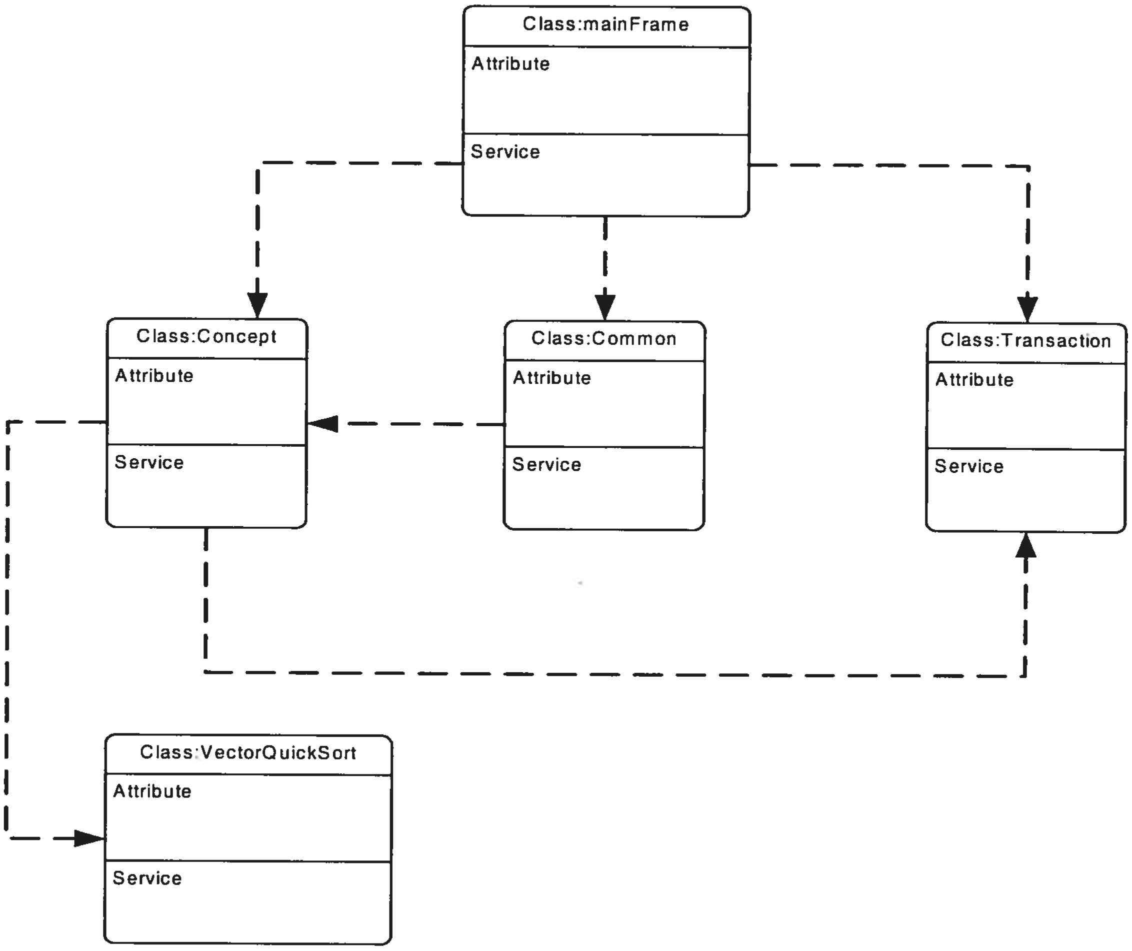

Figure A-2: Class diagram

Class *Concept* has two private data members: *Extent* and *Intent*. They belong to basic data type *String*. This class is used to implement check status of concepts when a new transaction is being inserted; it will also find $gen_h(c)$, $T^h(c)$ and $domain(c)$ as well as finding the lower cover of chosen concept in $\mathbf{V}^+(t_{i+1})$.

Class *Transaction* operates on the transaction set. We have two file types, *.txt* and *.dat*. Meanwhile we also use it to establish and maintain *ITT* (Item_Transaction_Table) and *TIT* (Transaction_Item_Table).

Class *VectorQuickSort* processes the vector, primarily for sorting the elements in vector according a certain field.

Class *Common* updates iceberg lattice with new generated frequent closed itemsets. Furthermore, it will perform vector operations, i.e. combine two vectors or judge if two vectors are the same and so on.

Class *MainFrame* sorts concepts in iceberg lattice. It also extracts transaction one by one from the transaction set.

4 Class definitions

Class *Concept*

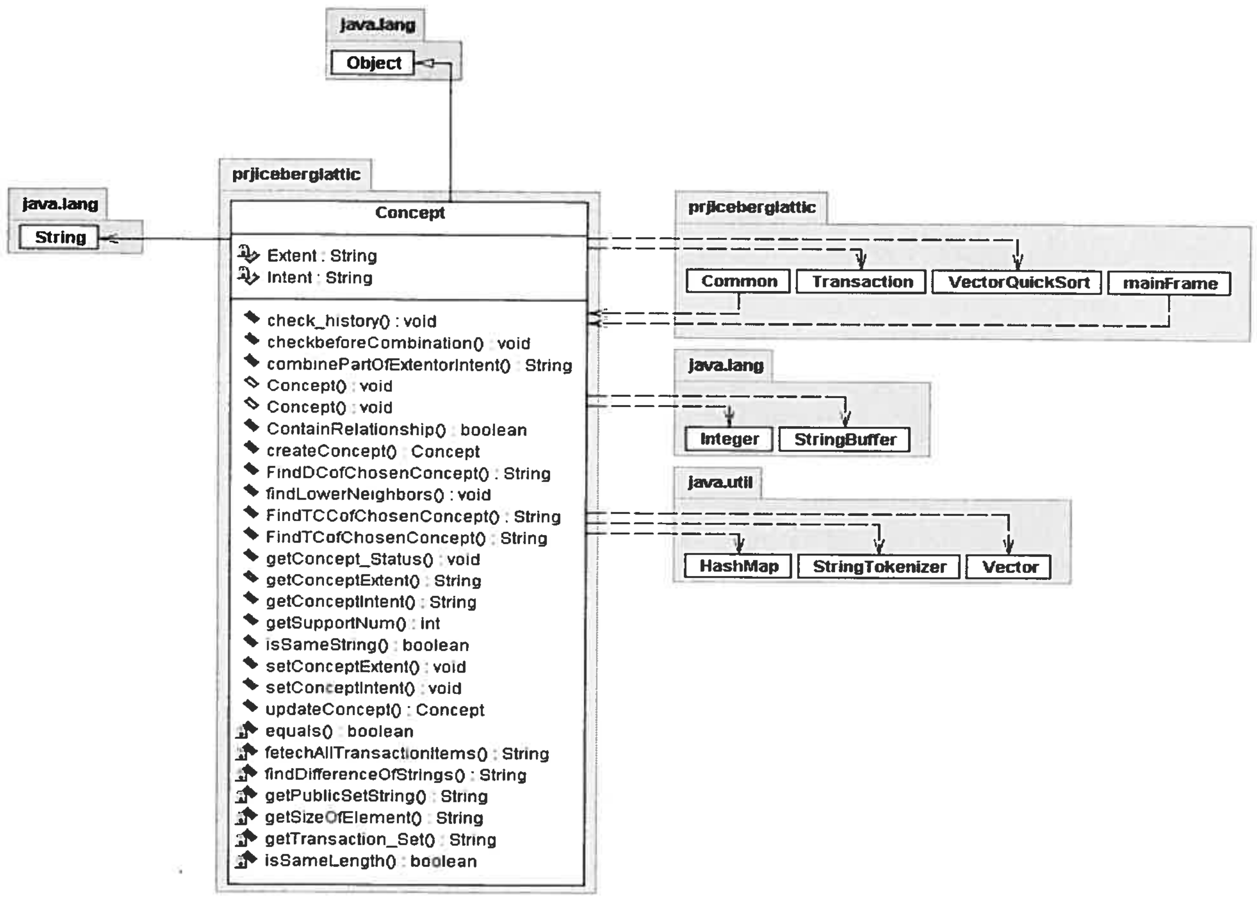
java.lang
Object
java.lang
String
prjicebergiattic
Concept
Extent : String
Intent : String
check_history() : void
checkbeforeCombination() : void
combinePartOfExtentorIntent() : String
Concept() : void
Concept() : void
ContainRelationship() : boolean
createConcept() : Concept
FindDCofChosenConcept() : String
findLowerNeighbors() : void
FindTCCofChosenConcept() : String
FindTCofChosenConcept() : String
getConcept_Status() : void
getConceptExtent() : String
getConceptIntent() : String
getSupportNum() : int
isSameString() : boolean
setConceptExtent() : void
setConceptIntent() : void
updateConcept() : Concept
equals() : boolean
fetechAllTransactionItems() : String
findDifferenceOfStrings() : String
getPublicSetString() : String
getSizeOfElement() : String
getTransaction_Set() : String
isSameLength() : boolean
prjicebergiattic
Common
Transaction
VectorQuickSort
mainFrame
java.lang
Integer
StringBuffer
java.util
HashMap
StringTokenizer
Vector

Figure A-3: Class diagram for class Concept

Attributes: *Extent*: String *Intent*: String

Methods Description

public Concept(String Extent, String Intent)	Constructor
public String combinePartOfExtentorIntent(String Str_1, String Str_2)	Combine str_1 to str_2, no repeat
public boolean ContainRelationship(String str)	Judge if contain all elements in str

public prjiceberglattic.Concept createConcept(String TID_Set,String Item_Set)	Create a new concept with TID_Set and Item_Set as concept's Extent and Intent
private boolean equals(Concept CompareObj)	Judge if two concepts are the same, if they have the same Extent and Intent, they are equal
private String findDifferenceOfStrings(String Str_1,String Str_2)	Calculate the difference between str_1 and str_2
public void findLowerCovers(Concept tmpConceptOfNewTransaction, Vector Vplus, Transaction TransObj, Vector Hplus, double mini_Support)	Find the lower covers of a concept
public void getConcept_Status(Concept ConObj, Concept newConcept, Vector iceBergLatticObj,Vector NewFCIObj,Transaction TransObj, double mini_Support,HashMap newTransaction,Vector Vplus)	Judge the status of a concept when add a new transaction and execute different operations according to their different status
public String getConceptExtent(Concept ConObj)	Get the Extent of a concept
public String getConceptIntent(Concept ConObj)	Get the Intent of a concept
private String getPublicSetString(String Str1,String Str2)	Calculate the intersection of Str_1 and Str_2
private String getTransaction_Set(Transaction TransObj,String ItemID)	Get a collection of TID from Item_Transaction_Table those transaction include this item
public Concept updateConcept(Concept Obj,String NewExtent)	Combine NewExtent to a concept's Extent, no repeat.

public int getSupportNum(Concept conObj)	Calculate the length of extent of a concept
public void setConceptExtent(String newExtent)	Update the Extent of concept with newExtent
public void setConceptIntent(String newIntent)	Update the Intent of concept with newIntent
private boolean isSameLength(String Str1,String Str2)	Judge if the length of two strings are the same
public boolean isSameString(String Str_1,String Str_2)	Judge if two strings are the same
public void checkbeforeCombination(Vector tartgetVector, Concept insertConcept)	If the intent or extent of insertConcept is not exist in tartgetVector, then add insertConcept to insertConcept, otherwise combine their intent or extent
public String FindTHCofChosenConcept(Concept elementOfVplus, Vector Hplus)	Find T^h (c) for concept c
public String FindDCofChosenConcept(Concept masterConcept, Concept c,Vector Vplus,Vector Hplus)	Find domain(c) for concept c
private String getSizeOfElement(String Element)	Calculate the length of a string

Table A-1: Methods description for class Concept

Class *Transaction*

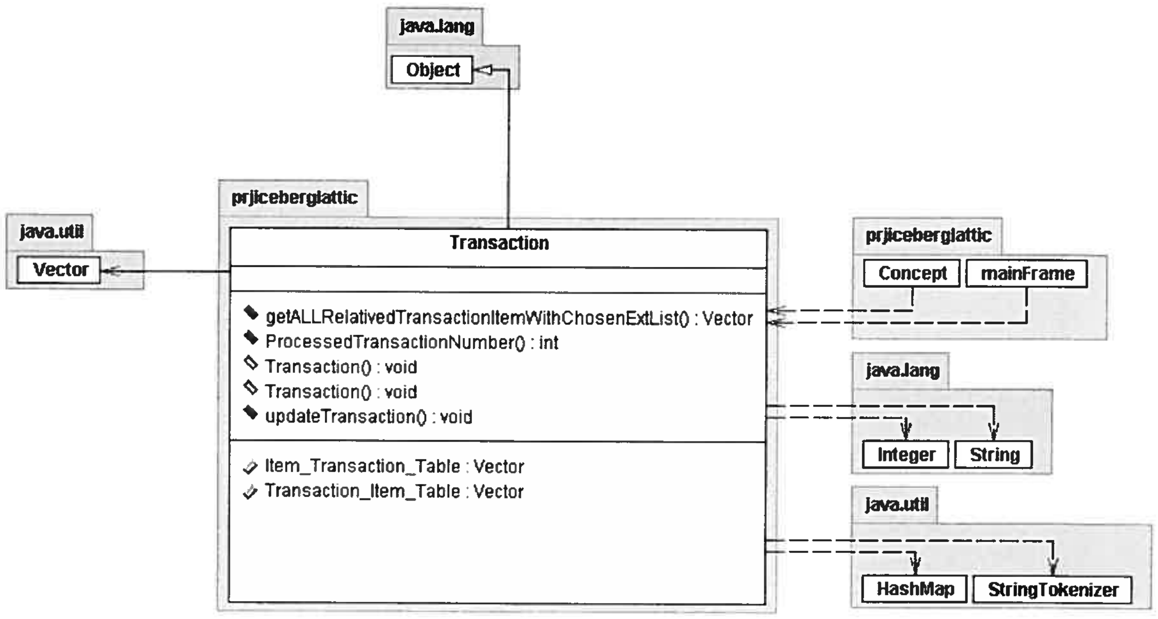

Figure A-4 Class diagram for class Transaction

Attributes: *Transaction_Item_Table*: Vector *Item_Transaction_Table*: Vector
Methods Description

public Transaction();	Constructor
public Transaction(java.lang.String Transaction_Line);	Constructor
public java.util.Vector getItem_Transaction_Table();	Return current item transaction table
public java.util.Vector getTransaction_Item_Table();	Return current transaction item table
public int ProcessedTransactionNumber();	Record the number of transaction which have been processed
public Vector getALLRelativedTransactionItemWithChosenExtList (String ExtList);	find the intersection of intent for several transactions
public void updateTransaction(String Transaction_Line)	Update Transaction_Item_Table and Item_Transaction_Table

Table A-2: Methods description for class Transaction

Class *VectorQuickSort*

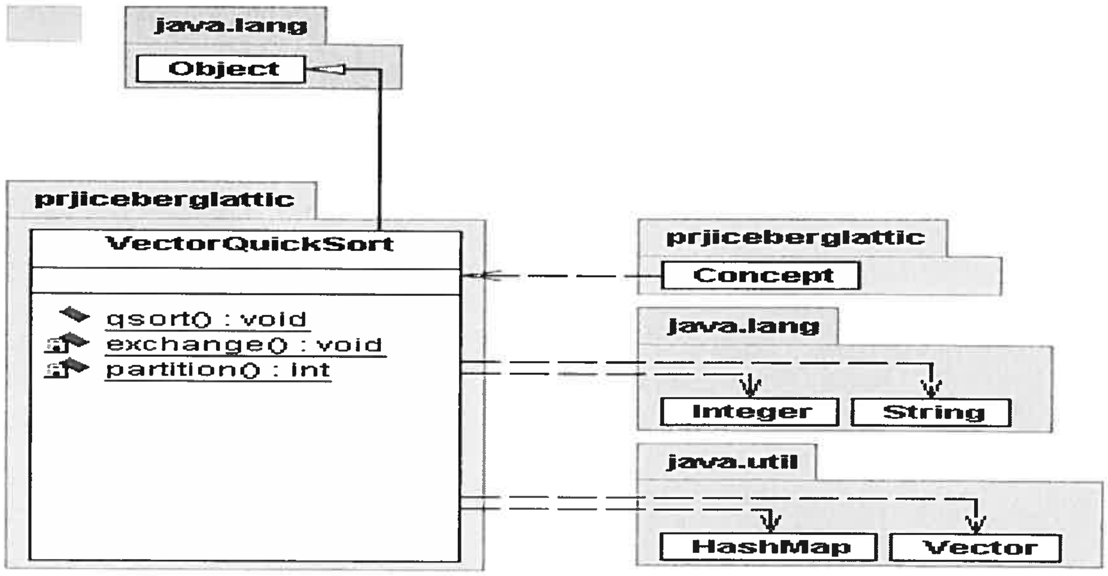

Figure A-5 Class diagram for class VectorQuickSort

Attributes: *a* Vector

Methods Description

private static void exchange(Vector a, int m, int n);	exchange the position of m and n
public static void qsort(Vector a, int l, int r, String compareField);	Quick Sort the vector according to a field

Table A-3: Methods description for class VectorQuickSort

Class *Common*

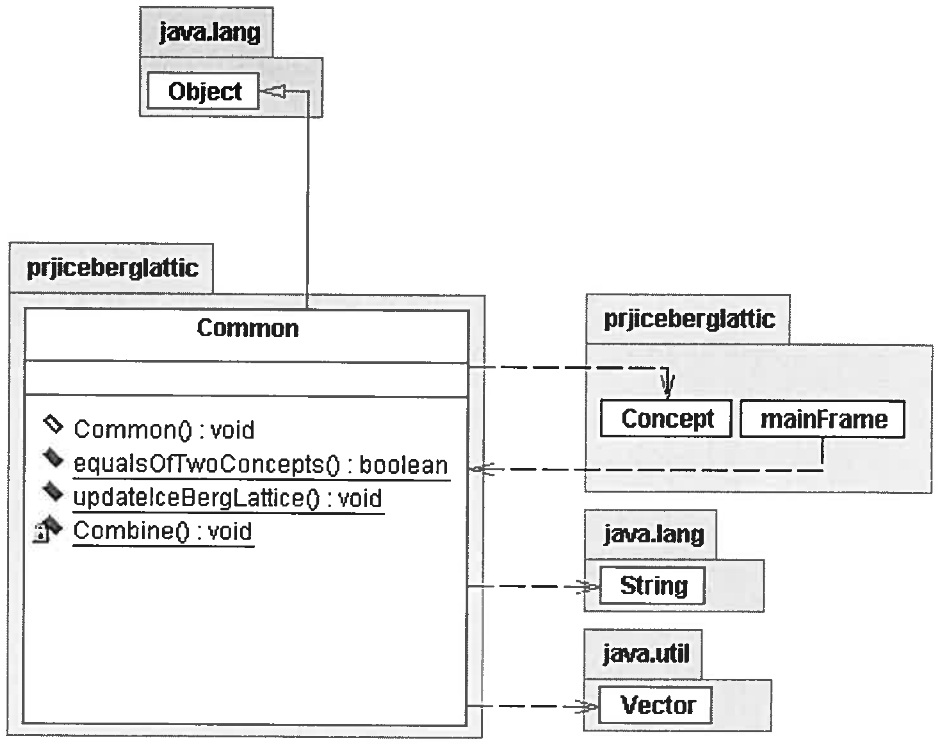

Figure A- 6: Class diagram for class Common

Attributes:

Methods Description

public Common();	Constructor
public static void updateIceBergLattice(Vector iceBergLatticObj,Vector NewFCIObj);	Update Iceberg Lattice with new Frequent Closed Itemsets
private static void Combine(Vector vObj1, Vector vObj2);	Combine two vectors, no repeat
public static boolean equalsOfTwoConcepts(Concept a, Concept b);	Judge if two concepts are the same

Table A-4: Methods description for class Common

Class *MainFrame*

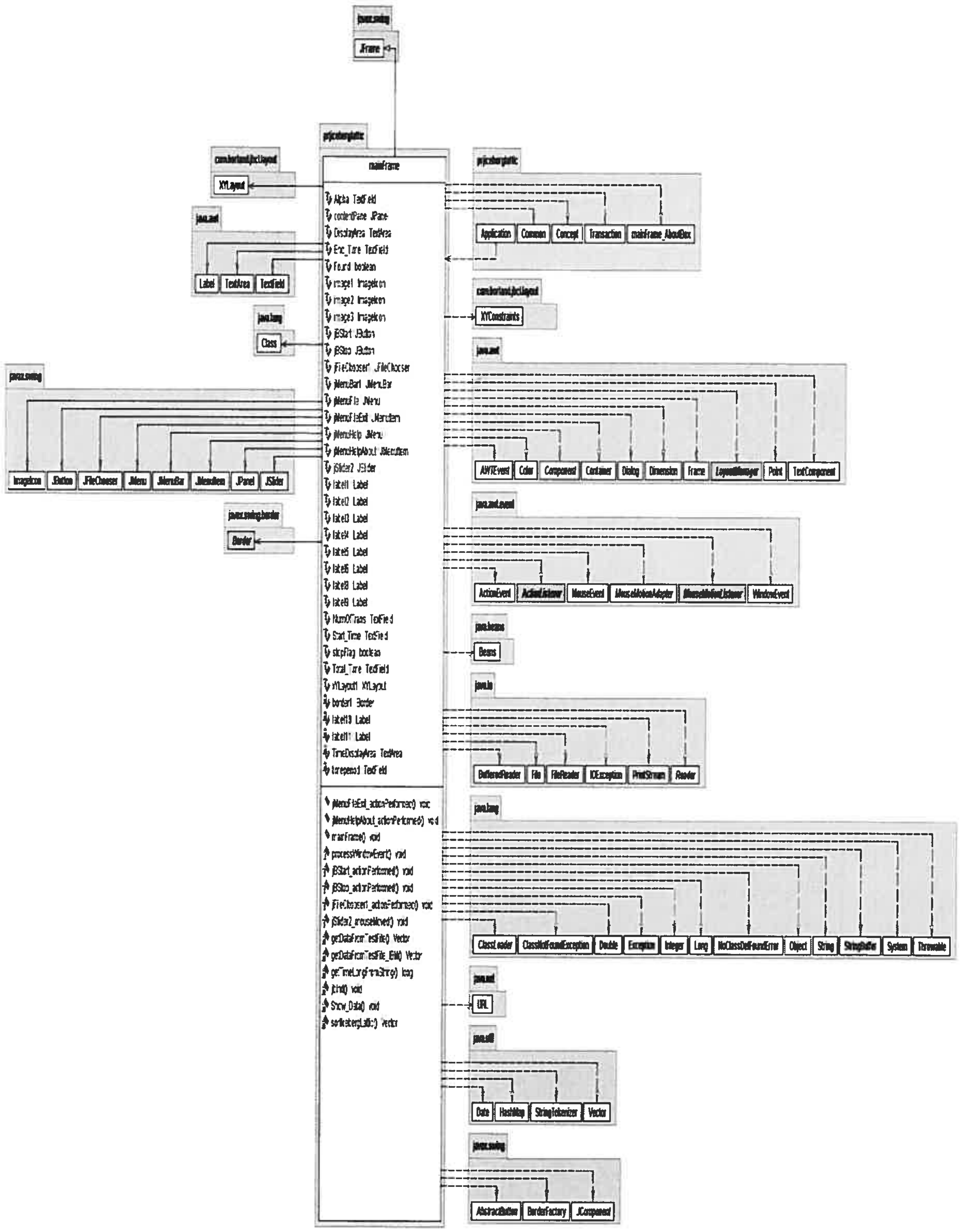

Figure A-7: Class diagram for class MainFrame

Attributes:

Methods Description

void jBStart_actionPerformed(ActionEvent e)	Action Event
public Vector getDataFromTestFile(String filePath)	get transaction one by one from data set (. Txt file)
public Vector getDataFromTestFile_IBM(String filePath)	get transaction one by one from IBM data set (.dat file)
public Vector sortIcebergLattic(Vector iceBergLattic)	sort Iceberg Lattice

Table A-5: Methods description for class MainFrame

5 Validation

After implementing the Iceberg Lattice Algorithm, we validated it by comparing the result with other validated algorithm. During this process, we adopt the *CHARM* algorithm – another algorithm for calculating the frequent closed itemsets from a transaction set.

Validation was conducted on one transaction set (T25I10D10K-300) using a different *minsupp*. T25I10D10K-300 has 300 transactions. It was extracted from data set T25I10D10K [UCI]. The formation of transaction is as following:

the number of items in this transaction (n)	$item_1$	$item_2$	…	$item_n$

For example, following is a transaction:

7 *267 348 389 456 523 624 657*

7 means that there are 7 items in this transaction,

267, 348, 389, 456, 523, 624 and *657* are 7 items in this transaction.

Table A-6 shows the comparing result:

Algorithm	Transaction set	*Minsupp*	Number of *FCIs*	note
ILA	T25I10D10K-300	0.01	3781	
		0.03	454	
		0.05	122	
		0.07	20	
		0.10	0	
CHARM	T25I10D10K-300	Support	Number of *FCIs*	In *CHARM*: Support = minsupp *transaction number
		3	3781	
		9	454	
		15	122	
		21	20	
		30	0	

Table A-6: The results of comparison with *CHARM* algorithm

In addition to comparing the number of frequent closed itemsets in the results, we also compared the extent and intent for every concept.

In order to explain the validation result visually, we give the results obtained from two algorithms under the same *minsupp* (in *ILA minsupp* = 0.07, in *CHARM* algorithm, Support=21).

Result from *ILA* (*Minsupp* =0.07, the total number of frequent closed itemsets is 20)

No.	Extent	Intent	Support
1	0 42 73 95 106 113 119 128 130 133 104 5 36 76 2 69 55 12 21 148 163 182 228 255 265 270 292 296	172	28
2	11 27 42 71 77 95 110 129 131 136 14 25 17 141 151 156 172 179 216 229 237 261 281 286 294	549	25
3	5 36 46 95 97 104 121 129 135 153 156 92 62 3 60 28 169 189 196 210 227 233 285	751	23
4	5 42 71 151 14 25 33 61 95 93 156 150 110 99 88 60 163 180 197 224 229 264 290	60	23
5	9 41 65 80 120 138 148 180 190 192 202 210 220 228 235 238 262 265 277 287 297 127 102 95 67 49 31	322	27
6	7 23 14 48 61 75 82 83 88 97 98 145 155 157 166 173 183 185 205 238 239 243 257 260 299	233	25
7	6 26 24 49 76 89 93 96 121 123 131 135 142 145 146 170 171 186 244 245 274	799	21
8	6 45 73 78 79 98 101 106 153 155 165 167 168 172 177 197 204 221 240 250 263 274 282 288	432	24
9	44 56 60 65 72 78 87 105 123 156 157 165 170	152	24

	190 194 198 200 231 255 256 267 275 277 285		
10	0 42 73 95 125 138 144 159 182 199 201 207 212 222 224 228 242 252 101 28 38 60 258 274 296	578	25
11	14 20 60 101 105 110 117 130 131 142 151 152 167 194 206 208 227 233 247 249 251 256 272 273 279	852	25
12	12 14 20 46 67 81 86 106 112 117 132 135 147 173 191 222 228 234 252 257 285 290	500	22
13	11 51 70 82 86 89 92 110 116 139 153 157 163 180 187 205 230 235 244 250 270 286	27	22
14	35 48 54 55 56 60 105 117 127 137 156 171 192 196 211 216 225 248 260 271 279 283 287 298	785	24
15	18 21 45 69 78 81 85 91 109 132 142 153 177 226 235 243 252 259 263 269 280 285 295	113	23
16	33 63 76 77 99 140 151 157 159 161 167 185 197 198 203 218 241 244 249 273 282 292 293	866	23
17	8 23 25 87 88 97 124 136 143 165 167 174 178 199 210 217 231 246 267 275 285	293	21
18	11 20 27 59 89 90 96 103 141 149 175 183 213 219 238 240 243 246 261 282 284 286	745	22
19	8 17 28 50 57 72 74 91 94 115 145 161 163 181 217 236 247 249 259 263 288 292	974	22
20	4 13 32 60 78 94 102 112 3 17 133 134 139 152 162 179 187 223 235 282 289 297	80	22

Table A-7: Result from *ILA* (m*insupp*=0.07)

Result from *CHARM* algorithm

(support=21, the total number of frequent closed itemsets is 2 0)

```
- <listeFCIs>
- <fci>
  <intent>799</intent>
  <extent>7 25 27 50 77 90 94 97 122 124 132 136 143 146 147 171 172 187 245 246  275</extent>
  <support>21</support>
  </fci>
- <fci>
  <intent>60</intent>
  <extent>6 15 26 34 43 61 62 72 89 94 96 100 111 151 152 157 164 181 198 225 230 265 291</extent>
  <support>23</support>
  </fci>
- <fci>
  <intent>751</intent>
  <extent>4 6 29 37 47 61 63 93 96 98 105 122 130 136 154 157 170 190 197 211 228 234 286</extent>
  <support>23</support>
  </fci>
- <fci>
  <intent>80</intent>
  <extent>4 5 14 18 33 61 79 95 103 113 134 135 140 153 163 180 188 224 236 283 290 298</extent>
  <support>22</support>
  </fci>
- <fci>
  <intent>500</intent>
  <extent>13 15 21 47 68 82 87 107 113 118 133 136 148 174 192 223 229 235 253 258 286 291</extent>
  <support>22</support>
  </fci>
- <fci>
  <intent>974</intent>
  <extent>9 18 29 51 58 73 75 92 95 116 146 162 164 182 218 237 248 250 260 264 289 293</extent>
  <support>22</support>
  </fci>
- <fci>
  <intent>293</intent>
  <extent>9 24 26 88 89 98 125 137 144 166 168 175 179 200 211 218 232 247 268 276 286</extent>
  <support>21</support>
  </fci>
- <fci>
  <intent>27</intent>
  <extent>12 52 71 83 87 90 93 111 117 140 154 158 164 181 188 206 231 236 245 251 271 287</extent>
  <support>22</support>
  </fci>
- <fci>
  <intent>172</intent>
  <extent>1 3 6 13 22 37 43 56 70 74 77 96 105 107 114 120 129 131 134 149 164 183 229 256 266 271 293 297</extent>
  <support>28</support>
  </fci>
- <fci>
  <intent>549</intent>
  <extent>12 15 18 26 28 43 72 78 96 111 130 132 137 142 152 157 173 180 217 230 238 262 282 287 295</extent>
  <support>25</support>
  </fci>
- <fci>
  <intent>233</intent>
  <extent>8 15 24 49 62 76 83 84 89 98 99 146 156 158 167 174 184 186 206 239 240 244 258 261 300</extent>
  <support>25</support>
  </fci>
- <fci>
  <intent>745</intent>
  <extent>12 21 28 60 90 91 97 104 142 150 176 184 214 220 239 241 244 247 262 283 285 287</extent>
  <support>22</support>
  </fci>
- <fci>
  <intent>113</intent>
  <extent>19 22 46 70 79 82 86 92 110 133 143 154 178 227 236 244 253 260 264 270 281 286 296</extent>
  <support>23</support>
  </fci>
- <fci>
  <intent>432</intent>
  <extent>7 46 74 79 80 99 102 107 154 156 166 168 169 173 178 198 205 222 241 251 264 275 283 289</extent>
  <support>24</support>
  </fci>
- <fci>
  <intent>152</intent>
  <extent>45 57 61 66 73 79 88 106 124 157 158 166 171 191 195 199 201 232 256 257 268 276 278 286</extent>
  <support>24</support>
  </fci>
- <fci>
  <intent>578</intent>
  <extent>1 29 39 43 61 74 96 102 126 139 145 160 183 200 202 208 213 223 225 229 243 253 259 275 297</extent>
```

```
 <support>25</support>
 </fci>
- <fci>
 <intent>785</intent>
 <extent>36 49 55 56 57 61 106 118 128 138 157 172 193 197 212 217 226 249 261 272 280 284 288 299</extent>
 <support>24</support>
 </fci>
- <fci>
 <intent>866</intent>
 <extent>34 64 77 78 100 141 152 158 160 162 168 186 198 199 204 219 242 245 250 274 283 293 294</extent>
 <support>23</support>
 </fci>
- <fci>
 <intent>852</intent>
 <extent>15 21 61 102 106 111 118 131 132 143 152 153 168 195 207 209 228 234 248 250 252 257 273 274 280</extent>
 <support>25</support>
 </fci>
- <fci>
 <intent>322</intent>
 <extent>10 32 42 50 66 68 81 96 103 121 128 139 149 181 191 193 203 211 221 229 236 239 263 266 278 288
         298</extent>
 <support>27</support>
 </fci>
 <nombreFCIs>20</nombreFCIs>
 </listeFCIs>
```

Reference

[AIS1993] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami

Mining association rules between sets of items in large databases. 1993.

In Proceedings of the 1993 International Conference on Management of Data (SIGMOD 93), pages 207-216.

[AIS1994] Rakesh Agrawal, Tomasz Imielinski and Ramakrishnan Srikant

Fast algorithms for mining association rules in large databases. 1994.

In Proceedings of the 20th International Conference on Very Large Databases (VLDB'94), Santiago, Chile, pages 487-499.

[AS1995]Rakesh Agrawal and Ramakrishnan Srikant

Mining sequential patterns. 1995

In Philip S. Yu and Arbee L. P. Chen, editors,

In Proceeding of the 11th International conference on Data Engineering, *ICDE* pages 3–14. IEEE Press, Pages: 6–10.

[ATA1999] N.Ayan, A.Tansel and M.Arkun

An efficient algorithm to update large itemsets with early pruning. 1999.

In Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 99), San Diego, CA, pages 287-291.

[B1998] R.J. Bayardo

Efficient mining long patterns from databases. 1998.

In Proceedings of ACM International Conference on Management of Data (SIGMOD), Seattle, Washington, USA, pages 85-93.

[BA1999] R.J. Bayardo and R. Agrawal

Mining the most interesting rules. 1999.

In Proceedings of the 5th International Conference on Knowledge Discovery and Data Ming (KDD'99)

[BCG2001] D. Burdick, M. Calimlim and J. Gehrke
MAFIA:a maximal frequent itemsets algorithm for transactional database. 2001. 17th International. Conference On Data Engineering, Heidelberg, Germany, page 0443.

[BM1970] M. Barbut and B. Monjardet
Ordre et Classification: Algèbre et combinatoire 1970
Paris. Hachette.

[CHNW1996] DW.Cheung, J.Han, VT.Ng and CY.Wong
Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique.1996.
In Proceedings, ICDE-96, New Orleans, LA, USA, pages 106-114.

[DP1990] B.A.Davey and H.A. Priestley
Introduction to lattices and order.1990.
Cambridge Mathematical Textbooks. Published by Cambridge University Press, Cambridge, viii+248 pages ISBN: 0-521-36584-8; 0-521-36766-2.

[G1997] Robert Groth
Data Mining – A Hands-On Approach for Business Professionals.1997.
Data Warehousing Institute Series, Paperback Prentice Hall PTR.

[GMA1995] R.Godin, R.Missaoui, and H.Alaoui
Incremental concept formation algorithms based on Galois (concept) lattices. 1995. Computational Intelligence (1995), 11(2), pages 246-267.

[GW1999] B. Ganter and R. Wille

Formal Concept Analysis, Mathematical Foundations. 1999

Published by Springer Verlag, Berlin.

[GZ2001] K.Gouda and MJ.Zaki

Efficiently Mining Maximal Frequent Itemsets.2001.

In Proceedings of the 2001 IEEE International Conference on Data Mining.

Pages 163-170.

[HGN2000] J.Hipp, U.Guntzer, and G.Nakharizadeh

Algorithms for Association Rule Mining- A general Survey and Comparison.2000.

ACM SIGKDD Explorations. Volume 2, Issue 1,page 58-64.

[HPY2000] J. Han, J. Pei and Y.Yin

Mining frequent patterns without candidate generation.2000.

In proceedings of the ACM SIGMOD International Conference.

Management of Data (SIGMOD'00), Dallas TX, USA, pages 21-30.

[HSH1998] M. Harries,C. Sammut and K. Horn

Extracting hidden context. 1998.

Machine Learning, 36(2), 1998, pages:101-126.

[K1998] D.E.Knuth

The Art of Computer Programming, 1998.

Vol.3, Sorting and Searching.

[M1997] H.Mannila

Methods and problems in data mining. 1997.

In proceedings of the 6th International Conference on Database Theory, Delphi, Greece, Pages 41-55.

[MT1996] H.Mannila and H.Toivonen

On an algorithm for finding all interesting sentences.1996.

In Cybernetics and Systems, Volume II, The 13Th European Meeting on Cybernetics and Systems Research, Vienna, Austria, pages 973-978.

[PBTL1998] N.Pasquier Y.Bastide, R.Taouil and L.Lakhal

Pruning closed itemsets lattices for association rules.1998.

Proceedings of the BDA French Conference on Advanced Databases, pages 177-196.

[PBTL1999-1] N.Pasquier Y.Bastide, R.Taouil and L.Lakhal

Efficient Mining of Association Rules Using Closed Itemsets lattices.1999.

Information systems, 24(1): pages 25-46.

[PBTL1999-2] N.Pasquier, Y.Bastide, R.Taouil, and L.Lakhal.

Discovering frequent closed itemsets for association rules.1999.

In Proceedings of the 7th International Conference on Database Theory, pages 398-416.

[PHM2000] J. Pei, J. Han and R. Mao

Closet: An efficient algorithm for mining frequent closed itemsets.2000.

In ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, pages 21-30.

[PS1991] G. Piatetsky-Shapiro

"Discovery of strong rules in databases,"

In G. Piatetsky-Shapiro and W. J. Frawley, eds,

“Knowledge Discovery in Databases”.

Menlo Park, CA: AAAI/MIT, 1991, pages: 229-238.

[SS1998] Ingo Schmitt and Gunter Saake
Merging inheritance hierarchies for database integration. 1998
In Proceedings of the 3rd International Conference on Cooperative Information Systems, New York City, New York, USA, pages 122-131.

[STBPL2000] Gerd Stumme, Rafic Taouil, Yves Bastide, Nicolas Pasquier, and Lotfi Lakhal
Fast computation of concept lattices using data mining technics. 2000.
In Proceedings of the 7th International Workshop on Knowledge Representation Meets Databases, Berlin, pages 21-22.

[SW2000] G. Stumme and R. Wille
Terminologische Merkmalslogik in der Formalen Begriffsanalyse 2000
Begriffliche Wissensverarbeitung. Methoden und Anwendungen, Springer, Berlin-Heidelberg-New York, pages 99-124.

[T1996] H.Toivonen
Sampling large database for association rules.1996.
In Proceedings of the 22nd international Conference on Very Large Data Bases Mumbai (Bombay), India, Pages: 134 – 145.

[TBAR1997] S.Thomas, S.Bodagala, K.Alsabti, and S.Ranka
An efficient algorithm for the incremental updation of association rules in large database. 1997.
In Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining (KDD 97), New Port Beach, CA, pages 263-266.

[UCI] S. D. Bay. The UCI KDD Archive [http://kdd.icn.uci .edu].
Irvine, CA: University of California,
Department of Information and Computer Science.

1:http://www.almaden.ibm.com/cs/quest/syndata.html

2:ftp://ftp.ics.uci.edu/pub/machine-learning-databases/mushroom/agaricus-lepiota.data

[VM2001] P.Valtchev and R.Missaoui

Building concept (Galois) lattices from parts: generalizing the incremental methods. 2001. In Proceedings, ICCS 2001, Stanford (CA), volume 2120 of Lecture Notes in Computer Science, pages 290-303.

[VMG2002] P.Valtchev, R.Missaoui and R.Godin

A Framework for Incremental Generation of Frequent Closed Itemsets. 2002. In Proceedings of the 1st International Workshop on Data Mining and Discrete Mathematics, Washington (DC), pages 75-86.

[VMGM2002] P.Valtchev, R.Missaoui, R.Godin, and M.Meridji

Generating Frequent Itemsets Incrementally: Two Novel Approaches Based on Galois Lattice Theory. 2002.
Journal of Experimental & Theoretical Artificial Intelligence, 134 (2-3): pages 115-142.

[VML2002] P.Valtchev, R.Missaoui and P.Lebrun

A partition-based approach towards constructing Galois (concept) lattices. 2002.
Discrete Mathematics, 256 (3): pages 801-829.

[VRM2003] P.Valtchev, M.Hacene.Rouane and R.Missaoui

A generic scheme for the design of efficient on-line algorithm for lattice. 2003. In Proceedings of the 11th Intl. Conference on Conceptual Structures (ICCS'03), Dresde (DE), Springer Verlag (LNAI), pages 282-295.

[W1982] R.Wille

Restructuring lattice theory: an approach based on hierarchies of concepts. 1982
In: I.Rival (ed.). Ordered sets. Reidel, Dordrecht-Boston, pages 445-470

[WTL1997] K.Waiyamai, R.. Taouil and L.Lakhal

Towards an object database approach for managing concept lattices. 1997.
In Proceedings of the 16th International Conference on Conceptual Modeling.
Springer, Heidelberg, pages 299-312.

[Z2000] MJ.Zaki

Generating Non-Redundant Association Rules. 2000.
In Proceedings, KDD-00, Boston, MA, USA, pages 34-43.

[ZH2002] Mohammed J. Zaki and Ching-Jui Hsiao

CHARM: An Efficient Algorithm for Closed Itemsets Mining. 2002.
In Proceedings of the 2nd SIAM International Conference on Data Mining
(ICDM'02), Arlinton, VA, pages 457-473.