Declarative Data Mining Using SQL3

Hasan M. Jamil

Department of Computer Science and Engineering
Mississippi State University, USA
jamil@cse.msstate.edu

Abstract. Researchers convincingly argue that the ability to declaratively mine and analyze relational databases using SQL for decision support is a critical requirement for the success of the acclaimed data mining technology. Although there have been several encouraging attempts at developing methods for data mining using SQL, simplicity and efficiency still remain significant impediments for further development. In this article, we propose a significantly new approach and show that any object relational database can be mined for association rules without any restructuring or preprocessing using only basic SQL3 constructs and functions, and hence no additional machineries are necessary. In particular, we show that the cost of computing association rules for a given database does not depend on support and confidence thresholds. More precisely, the set of large items can be computed using one simple join query and an aggregation once the set of all possible meets (least fixpoint) of item set patterns in the input table is known. We believe that this is an encouraging discovery especially compared to the well known SQL based methods in the literature. Finally, we capture the functionality of our proposed mining method in a mine by SQL3 operator for general use in any relational database.

1 Introduction

In recent years, mining association rules has been a popular way of discovering hidden knowledge from large databases. Most efforts have focused on developing novel algorithms and data structures to aid efficient computation of such rules. Despite major efforts, the complexity of the best known methods remain high. While several efficient algorithms have been reported [1,4,10,22,19,12,21,18,23], overall efficiency continues to be a major issue. In particular, in paradigms other than association rules such as ratio rules [11], chi square method [3], and so on, efficiency remains one of the biggest challenges.

The motivation, importance, and the need for integrating data mining with relational databases has been addressed in several articles such as [16,17]. They convincingly argue that without such integration, data mining technology may not find itself in a viable position in the years to come. To be a successful and feasible tool for the analysis of business data in relational databases, such technology must be made available as part of database engines as well as part of its declarative query language.

R. Meo et al.(Eds.): Database Support for Data Mining Applications,LNAI 2682, pp. 52–75, 2004.

While research into procedural computation of association rules has been extensive, fewer attempts have been made to use relational machinery or SQL for *declarative* rule discovery barring a few exceptions such as [13,25,21,8,20,14]. Most of these works follow an apriori like approach by mimicking its functionality and rely on generating candidate sets and consequently suffer from high computational overhead. While it is certainly possible to adapt any of the various procedural algorithms for rule mining as a special mining operator, the opportunity for using existing technology and constructs is preferable if it proves to be more beneficial. Some of the benefits of using existing relational machinery may include opportunity for query optimization, declarative language support, selective mining, mining from non-transactional databases, and so on. From these standpoints, it appears that research into data mining using SQL or SQL-like languages bear merit and warrant attention. But before we proceed any further, we would like to briefly summarize the concept of association rules as follows for the readers unfamiliar with the subject.

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of item identifiers. Let $\mathcal{T}$ be a transaction table such that every tuple in $\mathcal{T}$ is a pair, called the *transaction*, of the form $\langle t_{id}, X\rangle$ such that t_{id} is a unique transaction ID and $X \subseteq \mathcal{I}$ is a set of item identifiers (or items). A transaction is usually identified by its transaction ID t_{id}, and said to contain the item set X. An *association rule* is an implication of the form $X \rightarrow Y$, where $X, Y \subseteq \mathcal{I}$, and $X \cap Y = \emptyset$. Association rules are assigned a *support* (written as δ) and *confidence* (written as η) measure, and denoted $X \rightarrow Y\langle\delta, \eta\rangle$. The rule $X \rightarrow Y$ has a support δ, denoted $sup(X \rightarrow Y)$, in the transaction table $\mathcal{T}$ if $\delta\%$ of the transactions in $\mathcal{T}$ contain $X \cup Y$. In other words, $sup(X \rightarrow Y) \equiv sup(X \cup Y) = \delta = \frac{|\{t|t \in \mathcal{T} \wedge X \cup Y \subseteq t[I]\}|}{|\mathcal{T}|}$, where $I \subseteq \mathcal{I}$ is a set of items. On the other hand, the rule $X \rightarrow Y$ is said to have a confidence η, denoted $con(X \rightarrow Y)$, in the transaction table $\mathcal{T}$ if $\eta\%$ of the transactions in $\mathcal{T}$ that contain X also contains Y. So, the confidence of a rule is given by $con(X \rightarrow Y) = \eta = \frac{sup(X \cup Y)}{sup(X)}$.

Given a transaction table $\mathcal{T}$, the problem of mining association rules is to generate a set of quadruples $\mathcal{R}$ (a table) of the form $\langle X, Y, \delta, \eta\rangle$ such that $X, Y \subseteq \mathcal{I}$, $X \cap Y = \emptyset$, $\delta \geq \delta_m$, and $\eta \geq \eta_m$, where δ_m and η_m are user supplied minimum support and confidence *thresholds*, respectively. The clarity of the definitions and the simplicity of the problem is actually deceptive. As mentioned before, to be able to compute the rules $\mathcal{R}$, we must first compute the frequent item sets. A set of items X is called a frequent item set if its support δ is greater than the minimum support δ_m.

1.1 Related Research

Declarative computation of association rules were investigated in works such as [13,25,9,21,8,20,14,7,5].Meo et al. [14] proposes an SQL like declarative query language for association rule mining. The language proposed appears to be too oriented towards transaction databases, and may not be suitable for general association rule mining. It is worth mentioning that association rules may be

computed for virtually any type of database, transaction or not. In their extended language, they blend a rule mine operator with SQL and other additional features. The series of research reported in [25,21,20] led by IBM researchers, mostly addressed the mining issue itself. They attempted to compute the large item sets by generating candidate sets testing for their admissibility based on their MC model, combination, and GatherJoin operators. Essentially, these works proposed a method for implementing apriori using SQL. In our opinion, by trying to faithfully copy a procedural concept into a declarative representation they retain the drawbacks and inefficiencies of apriori in the model.

The mine rule operator proposed in [13] is perhaps the closest idea to ours. The operator has significant strengths in terms of expressive power. But it also requires a whole suit of new algebraic operators. These operators basically simulate the counting process using a set of predefined functions such as *CountAllGroups, MakeClusterPairs, ExtractBodies,* and *ExtractRules*. These functions use a fairly good number of new operators proposed by the authors, some of which use looping constructs. Unfortunately, no optimization techniques for these operators are available, resulting in doubts, in our opinion, about the computational viability of this approach.

In this article, we will demonstrate that there is a simpler SQL3 expression for association rule mining that does not require candidate generation such as in [25,21,20] or any implementation of new specialized operators such as in [13,14]. We also show that we can simply add an operator similar to cube by operator proposed for data warehousing applications with an optional having clause to facilitate filtering of unwanted derivations. The striking feature of our proposal is that we can exploit the vast array of optimization techniques that already exists and possibly develop newer ones for better performance. These are some of the advantages of our proposal over previous research in addition to its simplicity and intuitive appeal.

1.2 Contributions of this Article and Plan for the Presentation

We summarize the contributions of this article as follows to give the reader an idea in advance. We present a different view of transaction databases and identify several properties that they satisfy in general in section 2. We exploit these properties to develop a purely SQL3 based solution for association rule mining that uses the idea of least fix point computation. We rely on SQL3 standard as it supports complex structures such as sets and complex operations such as nesting and unnesting. We also anticipate the availability of several set processing functions such as **intersect** ($\cap$) and **setminus** ($\setminus$), set relational operators such as **subset** ($\subset$) and **superset** ($\supset$), nested relational operations such as nest by, etc. Finally, we also exploit SQL3's create view recursive and with constructs to implement our least fixpoint operator for association rule mining.

We follow the tradition of separating the large item set counting from actual mining and propose two operators – one to compute the large item sets from a source table, another one to compute the rules from the large item sets. We provide optional mechanisms to specify support and confidence thresholds and a

few additional constraints that the user may wish the mining process to satisfy. We also define a single operator version of mine by operator to demonstrate that it is possible to do so even within our current framework, even though we prefer the two stage approach.

The other implicit contribution of our proposal is that it opens up the opportunity for query optimization, something that was not practically possible until now in mining applications. Finally, it is now possible to use any relational database for mining in which one need not satisfy input restrictions similar to the ones that various mining algorithms require. Consequently, the developments in this article eliminates the need for any traditional preprocessing of input data.

In section 3, we present a discussion on a set theoretic perspective of data mining problem. This discussion builds upon the general properties of transaction tables presented in section 2. In this section, we demonstrate through illustrative examples that we can solve the mining problem just using set, lattice and aggregate operations if we adopt the idea of the so called *non-redundant* large item sets. Once the problem is understood on intuitive grounds, the rest of the development follows in fairly straightforward ways. In section 4 we present a series of SQL3 expressions that capture the spirit of the procedure presented in section 3. One can also verify that these expressions really produce the solution we develop in the illustrative example in this section. We then discuss the key idea we have exploited in developing the solution in section 5. The mining operator is presented in section 6 that is an abstraction of the series of SQL3 expressions in section 4. Several optimization opportunities and related details are discussed in section 7. Before we conclude in section 9, we present a comparative analysis of our method with other representative proposals in section 8.

2 Properties of Transaction Tables

In this section, we identify some of the basic properties shared by all transaction tables. We explain these properties using a synthetic transaction table in relational data model [24]. In the next section, we will introduce the relational solution to the association rule mining problem.

Let $\mathcal{I}$ be a set of items, $\mathcal{P}(\mathcal{I})$ be all possible item sets, T be a set of identifiers, and δ_m be a threshold. Then an *item set* table $\mathcal{S}$ with scheme {Tid, Items} is given by

$$\mathcal{S} \subseteq T \times \mathcal{P}(\mathcal{I})$$

such that $m = |\mathcal{S}|$. An item set table $\mathcal{S}$ is *admissible* if for every tuple $t \in \mathcal{S}$, and for every subset $s \in \mathcal{P}(t[Items])$, there exists a tuple $t' \in \mathcal{S}$ such that $s = t'[Items]$. In other words, every possible subset of items in a tuple is also a member of $\mathcal{S}$. The *frequency* table of an admissible item set table $\mathcal{S}$ can be obtained as

$$F =_{Items} \mathcal{G}_{\mathbf{count}(*)}({}_{Tid}\mathcal{G}(\mathcal{S}))$$

which has the scheme {Items, Count}. A *frequent item set* table F_f is a set of tuples that satisfies the count threshold property as follows.

$$F_f = \sigma_{Count/m \geq \delta_m}(F)$$

F_f satisfies some additional interesting properties. Suppose $I = t[Items]$ is an item set for any tuple $t \in F_f$. Then, for any $X, Y \subset I$, there exists t_1 and t_2 in F_f such that $t_1[Items] = X$, $t_2[Items] = Y$, $t_1[Count] \geq t[Count]$, $t_2[Count] \geq t[Count]$, and $t[Count] \leq min(t_1[Count], t_2[Count])$. The converse, however, is not true. That is, for any two tuples t_1 and t_2 in F_f, it is not necessarily true that there exists a tuple $t \in F_f$ such that $t[Items] = t_1[Items] \cup t_2[Items]$. But if such a t exists then the relation $t[Count] \leq min(t_1[Count], t_2[Count])$ is always true. Such a relationship is called *anti-transitive*.

The goal of the first stage of apriori like algorithms has been to generate the frequent item set table described above from a transaction table $\mathcal{T}$. Note that a transaction table, as defined above, is, in reality, not admissible. But the first stage of apriori mimics admissibility by constructing the k item sets at every kth iteration step.

Once the frequent item set table is available, the association rule table $\mathcal{R}$ can be computed as[1]

$$\mathcal{R} = \Pi_{F_{f_1}.Items, F_{f_2}.Items \setminus F_{f_1}.Items, \frac{F_{f_1}.Count}{m}, \frac{F_{f_2}.Count}{F_{f_1}.Count}} \times (\sigma_{F_{f_1}.Items \subset F_{f_2}.Items}(F_{f_1} \times F_{f_2}))$$

This expression, however, produces all possible rules, some of which are even redundant. For example, let $a \rightarrow b\langle \frac{s_{ab}}{m}, \frac{s_{ab}}{s_a} \rangle$ and $ab \rightarrow c\langle \frac{s_{abc}}{m}, \frac{s_{abc}}{s_{ab}} \rangle$ be two rules discovered from F_f, where s_X and m represent respectively the frequency of an item set X in the item set table (i.e., $t \in F_f$, and $t[Count] = \frac{s_X}{m}$), and number of transactions in the item set table $\mathcal{S}$. Then it is also the case that $\mathcal{R}$ contains another rule (transitive implication) $a \rightarrow bc\langle \frac{s_{abc}}{m}, \frac{s_{abc}}{s_a} \rangle$. Notice that this last rule is a logical consequence of the first two rules that can be derived using the following inference rule, where $X, Y, Z \subset \mathcal{I}$ are item sets.

$$\frac{X \rightarrow Y\langle \frac{s_{X \cup Y}}{m}, \frac{s_{X \cup Y}}{s_X} \rangle \qquad X \cup Y \rightarrow Z\langle \frac{s_{X \cup Y \cup Z}}{m}, \frac{s_{X \cup Y \cup Z}}{s_{X \cup Y}} \rangle}{X \rightarrow Y \cup Z\langle \frac{s_{X \cup Y \cup Z}}{m}, \frac{s_{X \cup Y \cup Z}}{s_X} \rangle}$$

Written differently, using only symbols for support (δ) and confidence (η), the inference rule reads as follows.

$$\frac{X \rightarrow Y\langle \delta_1, \eta_1 \rangle \qquad X \cup Y \rightarrow Z\langle \delta_2, \eta_2 \rangle}{X \rightarrow Y \cup Z\langle \delta_2, \eta_1 * \eta_2 \rangle}$$

Formally, if $X, Y, Z \subseteq \mathcal{I}$ be sets of items, and $X \rightarrow Y\langle \delta_1, \eta_1 \rangle$, $X \cup Y \rightarrow Z\langle \delta_2, \eta_2 \rangle$ and $X \rightarrow Y \cup Z\langle \delta_3, \eta_3 \rangle$ hold, then we say that $X \rightarrow Y \cup Z\langle \delta_3, \eta_3 \rangle$ is an *anti-transitive* rule. Also, if $r = X \rightarrow Y\langle \delta, \eta \rangle$ be a rule, and δ_m and η_m respectively be the minimum support and confidence requirements, then r is *redundant* if it is derivable from other rules, or if $\delta < \delta_m$ or $\eta < \eta_m$.

[1] Assuming that two copies of F_f are available as F_{f_1} and F_{f_2}.

It is possible to show that for any minimum support and confidence thresholds δ_m and η_m respectively, if the rules $X \rightarrow Y\langle\delta_1, \eta_1\rangle$, and $X \cup Y \rightarrow Z\langle\delta_2, \eta_2\rangle$ hold, then the rule $X \rightarrow Y \cup Z\langle\delta_3, \eta_3\rangle$ also holds such that $\delta_3 = \delta_2 \geq \delta_m$, and $\eta_3 = \eta_1 * \eta_2 \leq min(\eta_2, \eta_1)$. Notice that $\eta_3 = \eta_1 * \eta_2$ could be less then the confidence threshold η_m, even though $\eta_1 \geq \eta_m$ and $\eta_2 \geq \eta_m$. In other words, $\eta_1 \geq \eta_m \wedge \eta_2 \geq \eta_m \not\Rightarrow \eta_1 * \eta_2 \geq \eta_m$. Furthermore, we can show that any rule $r = X \rightarrow Y\langle\delta, \eta\rangle$ is redundant if it is anti-transitive. It is interesting to observe that the redundancy of rules is a side effect of redundancy of large item sets. Intuitively, for any given pair of large item sets l_1 and l_2, l_1 is *redundant* if $l_1 \subseteq l_2$, and the support s_{l_1} of l_1 is equal to the support s_{l_2} of l_2. Otherwise, it is *non-redundant*. Intuitively, l_1 is redundant because its support s_{l_1} can be computed from s_{l_2} just by copying. A more formal treatment of the concept of large itemsets and redundant large itemsets may be found in section 5.1.

Since in the frequent item set table, every item set is a member of a chain that differs by only one element, the following modification for $\mathcal{R}$ will compute the rules that satisfies given support and confidence thresholds and avoids generating all such redundant rules[2].

$$\begin{aligned}\mathcal{R} = \sigma_{Conf \geq \eta_m}(&\rho_{r(Ant,Cons,Sup,Conf)} \\ &\times(\Pi_{F_{f_1}.Items, F_{f_2}.Items \setminus F_{f_1}.Items, \frac{F_{f_1}.Count}{m}, \frac{F_{f_2}.Count}{F_{f_1}.Count}} \\ &\times(\sigma_{F_{f_1}.Items \subset F_{f_2}.Items \wedge (|F_{f_2}.Items| - |F_{f_1}.Items| = 1)}(F_{f_1} \times F_{f_2}))))\end{aligned}$$

2.1 The Challenge

The preceding discussion was aimed to demonstrate that a relational computation of association rules is possible. However, we used an explicit generation of the power set of the items in $\mathcal{I}$ to be able to compute the frequent item set table F_f from the item set table $\mathcal{S}$. This is a huge space overhead, and consequently, imposes a substantial computational burden on the method. Furthermore, we required that the item set table $\mathcal{S}$ be admissible, another significant restriction on the input transaction table. These are just some of the difficulties faced when a set theoretic or relational characterization of data mining is considered. The procedurality involved acts as a major bottleneck. So, the challenge we undertake is to admit any arbitrary transaction table, yet be able to compute the association rules "without" explicit generation of candidate item sets from a relational database, and compute the relation $\mathcal{R}$ as introduced before using existing SQL3 constructs and machineries.

3 A Set Theoretic Perspective of Data Mining

In this section, we present our idea of a SQL mine operator on intuitive grounds using a detailed example. The expectation here is that once we intuitively understand the issues related to the operator, it should be relatively easier to follow

[2] ρ is a relation renaming operator defined in [24].

the technical developments in the later sections. Also, this simple explanation will serve as the basis for a more general relational mining operator we plan to present at the end of this article.

Consider a database, called the transaction table, **T** as shown in figure 1. Following the traditional understanding of association rule mining, and also from the discussion in section 2, from the source table **T** we expect to obtain the large item set table (l_table) and the rules table (r_table) shown in figure 1 below once we set the support threshold at 25%. The reasoning process of reaching to the large item set and rules tables can be explained as follows.

t_table

Tranid	Items
t_1	a
t_1	b
t_1	c
t_2	b
t_2	c
t_2	f
t_3	b
t_3	f
t_4	a
t_4	b
t_4	c
t_5	b
t_5	e
t_6	d
t_6	f
t_7	d

transaction table

l_table

Items	Support
{a, b, c}	.29
{b, f}	.29
{b, c}	.38
{f}	.43
{d}	.29
{b}	.71

large item set table

r_table

Ant	Cons	Support	Conf
{b}	{c}	0.38	0.60
{f}	{b}	0.29	0.66
{b}	{f}	0.29	0.40
{b,c}	{a}	0.29	0.66

association rules table

Fig. 1. Source transaction database **T** is shown as t_table, large item set table as l_table, and finally the association rules as r_table

We can think of **T** as the set of complex tuples shown in nested table (n_table) in figure 2 once we nest the items on transaction numbers. If we use a group by on the Items column and count the transactions, we will compute the frequency table (f_table) in figure 2 that will show how many times a single item set pattern appears in the transaction table (t_table) in figure 1. Then, let us assume that we took a cross product of the frequency table with itself, and selected the rows for which

- the Items column in the first table is a proper subset of the Items column in the second table, and finally projected out the Items column of the first table and Support column of the second table[3], or
- the Items columns are not subset of one another, and we took the intersection of the Items of both the tables, created a new relation (int_table, called the

[3] This will give us $< \{b, f\}, 1 >$ and $< \{d\}, 1 >$.

n_table

Tranid	Items
t_1	{a,b,c}
t_2	{b,c,f}
t_3	{b,f}
t_4	{a,b,c}
t_5	{b,e}
t_6	{d,f}
t_7	{d}

nested table

i_table

Items	Support
{b,c}	3
{b,f}	1
{b}	5
{f}	3
{d}	1

inheritance table

f_table

Items	Support
{a, b, c}	2
{b,c,f}	1
{b, e}	1
{b, f}	1
{d}	1
{d,f}	1

frequency table

c_table

Items	Support
{a, b, c}	2
{b,c,f}	1
{b,c}	3
{b, f}	2
{b, e}	1
{d, f}	1
{b}	5
{f}	3
{d}	2

count table

Fig. 2. n_table: t_table after nesting on Tranid, f_table: n_table after grouping on Items and counting, i_table: generated from f_table, and c_table: grouping on Items and sum on Support on the union of i_table and f_table

intersection table) with distinct tuples of such Items with Support 0, and then finally computed the support counts as explained in step 1 now with the frequency table and intersection table[4].

This will give us the inheritance table (i_table) as shown in figure 2. Finally, if we took a union of the frequency table and the inheritance table, and then do a group by on the Items column and sum the Support column, we would obtain the count table (c_table) of figure 2.

The entire process of large item set and association rule generation can be conveniently explained using the so called item set lattice found in the literature once we enhance it with some additional information. Intuitively, consider placing the transactions with item set u appearing in the frequency table with their support count t as a node in the lattice $\mathcal{L}$ as shown in figure 3. Notice that in the lattice, each node is represented as u_c^t, where it denotes the fact that u appears in exactly t transactions in the source table, and that u also appears as a subset of other transactions n number of times such that $c = n + t$. t is called

[4] The result of this will be tuple $< \{b, c\}, 3 >$, $< \{b\}, 5 >$, and $< \{f\}, 3 >$ in this example. Note that the intersection table will contain the tuples $< \{b, c\}, 0 >$, $< \{b\}, 0 >$, and $< \{f\}, 0 >$, and that these patterns are not part of the frequency table in figure 2. The union of step 1, and step 2 processed with the intersection table will now produce the inheritance table in figure 2.

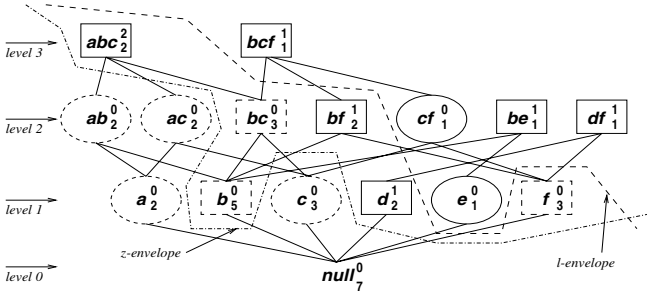

Fig. 3. Lattice representation of item set database **T**

the *transaction count*, or *frequency count*, and c is called the *total count* of item set u. The elements or nodes in $\mathcal{L}$ also satisfy additional interesting properties. A node v at level l differs from its child u at level $l-1$ by exactly 1 element, and that $u \subset v$. For any two children u and w of a node v, $v = u \cup w$. For any two nodes[5] $u_{c_u}^{t_u}$ and $v_{c_c}^{t_v}$ at any level l, their *join* is defined as $(u \cap v)_{c_j}^{t_j}$, and the *meet* as $(u \cup v)_{c_m}^{t_m}$, such that $c_j \leq min(c_u, c_v)$ and $c_m \geq max(c_u, c_v)$.

Note that in figure 3, the nodes marked with a solid rectangle are the nodes (or the item sets) in **T**, nodes identified with dotted rectangles are the *intersection*[6] nodes or the *virtual*[7] nodes, and the nodes marked with ellipses (dotted or solid) are *redundant*. The nodes below the dotted line, called the large item set envelope, or *l-envelope*, are the large item sets. Notice that the node *bc* is a large item set but is not a member of **T**, while *bcf*, *df* and *be* are in **T**, yet they are not included in the set of large item sets of **T**. We are assuming here a support threshold of 25%. So, basically, we would like to compute only the nodes *abc*, *bc*, *bf*, *b*, *d* and *f* from **T**. This set is identified by the *sandwich* formed by the l-envelope and the zero-envelope, or the *z-envelope*, that marks the lowest level nodes in the lattice. If we remove the non-essential, or redundant, nodes from the lattice in figure 3, we are left with the lattice shown in figure 4. It is possible to show that the lattice shown in figure 4 is the set of non-redundant large item sets of **T** at a support threshold 25%. The issue now is how to read this lattice. In other words, can we infer all the large item sets that an apriori like algorithm will yield on **T**? The answer is yes, but in a somewhat different manner. This is demonstrated in the following way.

Notice that there are five large 1-items – namely a, b, c, d and f. But only three, b, d and f, are listed in the lattice. The reason for not listing the other large 1-items is that they are implied by one of the nodes in the lattice. For

[5] For any node u, the notations t_u and c_u mean respectively the transaction count and total count of u.

[6] Nodes that share items in multiple upper level nodes and have a total count higher than any of the upper level nodes.

[7] Nodes with itemsets that do not appear in **T**, and also have total count equal to all the nodes above them.

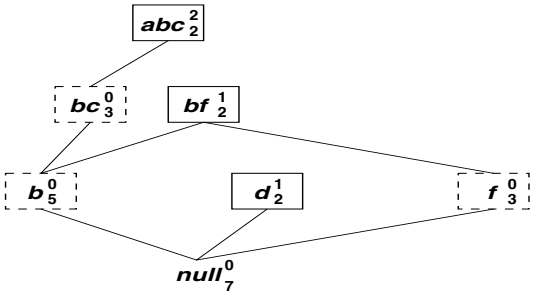

Fig. 4. Non-redundant large item sets of **T** when $\delta_m = 0.25$

example, c is implied by bc for which the count is 3. The nodes b and bc should be read as follows – b alone appears in **T** 5 times, whereas bc appears 3 times. Since c appears a maximum of 3 times with b (2 times in abc and 1 time in bcf actually), its total count can be derived from bc's count. Similarly, a's count can be derived from abc – 2. Hence, the lattice in figure 4 does not include them and considers them as redundant information. This view point has another important implication. We are now able to remove (or prune) redundant association rules too. We will list $b \rightarrow c\langle .38, .60\rangle$ and $bc \rightarrow a\langle .29, .66\rangle$, among several others, as two association rules that satisfy the 25% support and 40% confidence thresholds. Notice that we do not derive the rule $b \rightarrow ac\langle .29, .40\rangle$ in particular. The reason is simple – it is redundant for it can be derived from the rules $b \rightarrow c\langle .38, .60\rangle$ and $bc \rightarrow a\langle .29, .66\rangle$ using the following inference rule. Notice that if we accept the concept of redundancy we propose for rules, computing $b \rightarrow ac\langle .29, .40\rangle$ does not strengthen the information content of the discovery in any way.

$$\frac{X \rightarrow Y\langle \delta_1, \eta_1\rangle \qquad X \cup Y \rightarrow Z\langle \delta_2, \eta_2\rangle}{X \rightarrow Y \cup Z\langle \delta_2, \eta_1 * \eta_2\rangle} = \frac{b \rightarrow c\langle .38, .60\rangle \qquad bc \rightarrow a\langle .29, .66\rangle}{b \rightarrow ac\langle .29, .40\rangle}$$

Finally, we would like to point to an important observation. Take the case of the rules $b \rightarrow f\langle .29, .40\rangle$ and $f \rightarrow b\langle .29, .66\rangle$. These two rules serve as an important reminder that $X \rightarrow Y\langle s_1, c_1\rangle$, and $Y \rightarrow X\langle s_2, c_2\rangle \not\Rightarrow c_1 = c_2$, and that $X \rightarrow Y\langle s_1, c_1\rangle$, and $Y \rightarrow X\langle s_2, c_2\rangle \Rightarrow s_1 = s_2$. But in systems such as apriori, where all the large item sets are generated without considering redundancy, it would be difficult to prune rules based on this observation as we do not know which one to prune. For example, for the set of large item sets $\{bc_3, b_3, c_3\}$, we must derive rules $b \rightarrow c\langle \frac{3}{m}, 1\rangle$ and $c \rightarrow b\langle \frac{3}{m}, 1\rangle$[8] and cannot prune them without any additional processing. Instead, we just do not generate them at all.

[8] Assuming there are m number of transactions.

4 Computing Item Set Lattice Using SQL3

Now that we have explained what non-redundant large item sets and association rules mean in our framework, we are ready to discuss computing them using SQL. The reader may recall from our discussion in the previous section that we have already given this problem a relational face by presenting them in terms of (nested) tables. We will now present a set of SQL3 sentences to compute the tables we have discussed earlier. We must mention here that it is possible to evaluate the final table in figure 1 by mimicking the process using a lesser number of expressions than what we present below. But we prefer to include them all separately for the sake of clarity. In a later section, we will discuss how these series of SQL sentences can be replaced by an operator, the actual subject of this article.

For the purpose of this discussion, we will assume that several functions that we are going to use in our expressions are available in some SQL3 implementation, such as Oracle, DB2 or Informix. Recall that SQL3 standard requires or implies that, in some form or other, these functions are supported[9]. In particular, we have used a nest by clause that functions like a group by on the listed attributes, but returns a nested relation as opposed to a first normal form relation returned by group by. We have also assumed that SQL3 can perform group by on nested columns (columns with set values). Finally, we have also used set comparators in where clause, and set functions such as **intersect** and **setminus** in the select clause, which we think are natural additions to SQL3 once nested tuples are supported. As we have mentioned before, we have, for now, used user defined functions (UDFs) by treating set of items as a string of labels to implement these features in Oracle.

The following two view definitions prepare any first normal form transaction table for the mining process. Note that these view definitions act as idempotent functions on their source. So, redoing them does not harm the process if the source table is already in one of these forms. These two views compute the n_table and the f_table of figure 2.

```
create view n_table as
   (select Tranid, Items
   from t_table
   nest by Tranid)

create view f_table as
   (select Items, count(*) as Support
   from n_table
   group by Items)
```

Before we can compute the i_table, we need to know what nodes in the imaginary lattice will inherit transaction counts from some of the transaction

[9] Although some of these functions are not supported right now, once they are, we will be in a better shape. Until then, we can use PL/SQL codes to realize these functions.

nodes in the lattice – Support value of Items in the f_table. Recall that nodes that are subset of another node in the lattice, inherit the transaction count of the superset node towards its total count. We also know that only those (non-redundant) nodes which appear in the f_table, or are in the least fixpoint of the nodes in f_table will inherit them. So, we compute first the set of intersection nodes implied by f_table using the newly proposed SQL3 create view recursive statement as follows.

```
create view recursive int_table as
   ((select distinct intersect(t.Items, p.Items), 0
   from f_table as t, f_table as p
   where t.Items ⊄ p.Items and p.Items ⊄ t.Items
     and not exists
       (select *
       from f_table as f
       where f.Items = intersect(t.Items, p.Items)))
   union
   (select distinct intersect(t.Items, p.Items), 0
   from int_table as t, int_table as p
   where t.Items ⊄ p.Items and p.Items ⊄ t.Items
     and not exists
       (select *
       from f_table as f
       where f.Items = intersect(t.Items, p.Items))))
```

We would like to mention here again that we have implemented this feature using PL/SQL in Oracle. Notice that we did not list the int_table we create below in figure 1 or 2 because it is regarded as a transient table needed for the computation of i_table.

It is really important that we create only distinct set of intersection items and only those ones that do not appear in the f_table for the purpose of accuracy in support counting. Take for example three transactions in a new frequency table, f′_table, represented as $\{abc_0^1, bcd_0^1, bcf_0^1, bc_0^1\}$. Assume that we compute the set of intersections of the entries in this table. If we do not guard against the cautions we have mentioned, we will produce the set $\{bc_0^0, bc_0^0, bc_0^0\}$ using the view expression for int_table – which is not desirable. Because, these three will inherit Support from $\{abc_0^1, bcd_0^1, bcf_0^1\}$ giving a total count of 10, i.e., bc_{10}^1. The correct total count should have been bc_4^1. If we just ensure the uniqueness of a newly generated item set (but not its absence in the f_table) through meet computation, we still derive $\{bc_0^0\}$ instead of an empty set, which is also incorrect. This means that not including the following condition in the above SQL expression will be a serious mistake.

```
not exists (select *
   from f_table as f
   where f.Items = intersect(t.Items, p.Items)
```

Once we have computed the int_table, the rest of the task is pretty simple. The i_table view is computed by copying the Support of a tuple in f_table for any tuple in the collection of f_table and int_table which is a subset of the tuple in the f_table. Intuitively, these are the nodes that need to inherit the transaction counts of their ancestors (in f_table).

```
create view i_table as
   (select t.Items, p.Support
   from f_table as p,
        ((select *
        from f_table)
        union
        (select *
        from int_table)) as t,
   where t.Items ⊂ p.Items)
```

From the i_table, a simple grouping and sum operation as shown below will give us the count table, or the c_table, of figure 2.

```
create view c_table as
   (select t.Items, sum(t.Support) as Support
   from ((select *
        from f_table)
        union
        (select *
        from i_table)) as t
   group by t.Items)
```

The large item sets of l_table in figure 1 can now be generated by just selecting on the c_table tuples as shown next. Notice that we could have combined this step with the c_table expression above with the help of a having clause.

```
create view l_table as
   (select Items, Support
   from c_table
   where Support ≥ δ_m)
```

Finally, the (non-redundant) association rules of figure 1 are computed using the r_table view below. The functionality of this view can be explained as follows. Two item sets $u[Items]$ and $v[Items]$ in a pair of tuple u and v in the l_table implies an association rule of the form $u[Items] \rightarrow v[Items] \setminus u[Items]\langle v[Support], \frac{v[Support]}{u[Support]}\rangle$ only if $u[Items] \subset v[Items]$ and there does not exist any intervening item set x in the l_table such that x is a superset of $u[Items]$ and is a subset of $v[Items]$ as well. In other words, in the lattice, $v[Items]$ is one of the immediate ancestors of $u[Items]$. In addition, the ratio of the Supports, for example, $\frac{v[Support]}{u[Support]}$ must be at least equal to the minimum confidence threshold η_m.

```
create view r_table as
  (select a.Items, c.Items\a.Items, c.Support, c.Support/a.Support
  from l_table as a, l_table as c
  where a.Items ⊂ c.Items and c.Items/a.Items ≥ η_m and not exists
    (select Items
    from l_table as i
    where a.Items ⊂ i.Items and i.Items ⊂ c.Items))
```

The readers may verify that these are the only "generic" SQL3 expressions (or their equivalent) that are needed to mine any relational database (assuming proper name adaptations for tables and columns). The essence of this relational interpretation of the problem of mining, as demonstrated by the SQL3 expressions above, is that we do not need to think in terms of iterations, candidate generation, space time overhead, and so on. Instead, we can now express our mining problems on any relational database in declarative ways, and leave the optimization issues with the system and let the system process the query using the best available method to it, recognizing the fact that depending on the instance of the database, the choice of best methods may now vary widely.

5 An Enabling Observation

Level wise algorithms such as apriori essentially have three distinct steps at each pass k: (i) scan the database and count length k candidate item sets against the database, (ii) test and discard the ones that are not large item sets, and (iii) generate candidate item sets of length $k+1$ from the length k large items sets just generated and continue to next iteration level. The purpose of the second step is to prune potential candidates that are not going to generate any large item sets. This heuristic is called the *anti-monotonicity* property of large item sets. While this heuristic saves an enormous amount of space and time in large item set computation and virtually makes association rule mining feasible, further improvements are possible. We make an observation that apriori fails to remove *redundant* large item sets that really do not contribute anything new, and not generating the redundant large item sets do not cause any adverse effect on the discovery of the set of association rules implied by the database. In other words, apriori fails to potentially recognize another very important optimization opportunity. Perhaps the most significant and striking contribution of this new optimization opportunity is its side effect on the declarative computation of association rules using languages such as SQL which is the subject of this article. This observation of optimization opportunity helps us avoid thinking level wise and allows us to break free from the expensive idea of candidate generation and testing even in SQL like set up such as in [25,21,20]. As mentioned earlier, methods such as [4,27] have already achieved significant performance improvements over apriori by not requiring to generate candidate item sets.

To explain the idea we have on intuitive grounds, let us consider the simple transaction table t_table in figure 1. Apriori will produce the database in

figure 5(a) in three iterations when the given support threshold is ≈ 25%, or 2 out of 7 transactions.

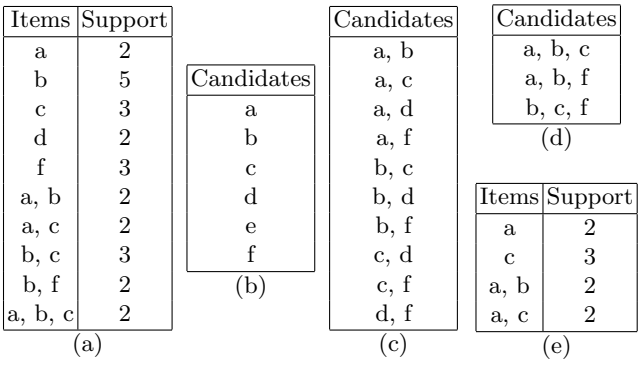

Fig. 5. (a) Frequent item set table generated by apriori and other major algorithms such as FP-tree. Candidate sets generated by a very *smart* apriori at iterations (b) $k = 1$, (c) $k = 2$, and at (d) $k = 3$

Although apriori generates the table in figure 5(a), it needs to generate the candidate sets in figure 5(b) through 5(d) to test. Notice that although the candidates $\{a, d\}$, $\{a, f\}$, $\{b, d\}$, $\{c, d\}$, $\{c, f\}$, $\{d, f\}$, $\{a, b, f\}$ and $\{b, c, f\}$ were generated (figures 5(c) and 5(d)), they did not meet the minimum support threshold and were never included in the frequent item set table in figure 5(a). Also notice that these are the candidate item sets generated by a very smart apriori algorithm that scans the large items sets generated at pass k in order to guess the possible set of large item sets it could find in pass $k + 1$, and selects the guessed sets as the candidate item sets. A naive algorithm on the other hand would generate all the candidate sets from the large item sets generated at pass k exhaustively by assuming that they are all possible. Depending on the instances, they both have advantages and disadvantages. But no matter which technique is used, apriori must generate a set of candidates, store them in some structures to be able to access them conveniently and check them against the transaction table to see if they become large item sets at the next iteration step. Even by conservatively generating a set of candidate item sets as shown in figures 5(b) through 5(d) for the database in figure 1(t_table), it wastes (for the candidate sets that never made it to the large item sets) time and space for some of the candidate sets. Depending on the transaction databases, the wastage could be significant. The question now is, could we generate candidate sets that will have a better chance to become a large item set? In other words, could we generate the absolute minimum set of candidates that are surely a large item set? In some way, we think the answer is in the positive as we explain in the next section.

5.1 Implications

We take the position and claim that the table l_table shown in figure 1 is an information equivalent table of figure 5(a) which essentially means that

these tables faithfully imply one another (assuming identical support thresholds, $\approx 25\%$). Let us now examine what this means in the context of association rule mining on intuitive grounds.

First of all, notice that the tuples (i.e., large item sets) missing in table of figure 1 are listed in table of figure 5(e), i.e., the union of these two tables gives us the table in figure 5(a). Now the question is why do we separate them and why do we deem the tuples in figure 5(e) redundant? Before we present the reasoning, we would like to define the notion of redundancy of large item sets in order to keep our discussion in perspective.

Definition 5.1. Let $\mathcal{T}$ be a transaction table over item sets $\mathcal{I}$, $I \subseteq \mathcal{I}$ be an item set, and n be a positive integer. Also let n represent the frequency of the item set I with which it appears in $\mathcal{T}$. Then the pair $\langle I, n\rangle$ is called a *frequent* item set, and the pair is called a *large* item set if $n \geq \delta_m$, where δ_m is the minimum support threshold.

We define redundancy of large item sets as follows. If for any large item set I, its frequency n can be determined from other large item sets, then I is redundant. Formally,

Definition 5.2 (Redundancy of Large Item Sets). Let L be a set of large item sets of tuples of the form $\langle I_u, n_u\rangle$ such that $\forall x, y(x = \langle I_x, n_x\rangle, y = \langle I_y, n_y\rangle \in L \wedge I_x = I_y \Rightarrow n_x = n_y)$, and let $u = \langle I_u, n_u\rangle$ be such a tuple. Then u is *redundant* in L if $\exists v(v \in L, v = \langle I_v, n_v\rangle, I_u \subseteq I_v \Rightarrow n_u = n_v)$.

The importance of the definition 5.1 may be highlighted as follows. For any given set of large item sets L, and an element $l = \langle I_l, n_l\rangle \in L$, I_l is unique in L. The implication of anti-monotonicity is that for any other $v = \langle I_v, n_v\rangle \in L$ such that $I_l \subset I_v$ holds, $n_v \leq n_l$ because an item set cannot appear in a transaction database less number of times than any of its supersets. But the important case is when $n_v = n_l$ yet $I_l \subset I_v$. This implies that I_l never appears in a transaction alone, i.e., it always appeared with other items. It also implies for all large item sets $s = \langle I_s, n_s\rangle \in L$ of I_v such that $I_s \supset I_v$, if it exists, $n_v = n_s$ too. As if not, n_l should be different than n_v, which it is not, according to our assumption. It also implies that I_l is not involved in any other sub-superset relationship chains other that I_v. There are several other formal and interesting properties that the large item sets satisfy some of which we will present in a later section.

The importance of the equality of frequency counts of large item sets that are related via sub-superset relationships is significant. This observation offers us another opportunity to optimize the computation process of large item sets. It tells us that there is no need to compute the large item sets for which there exists another large item set which is a superset and has identical frequency count. For example, for an item set $S = \{a, b, c, d, e, f, g, h\}$, apriori will iterate eight times if S is a large item set and generate $|\mathcal{P}(S)|$ subsets of S with identical frequency counts when, say, S is the only distinct item set in a transaction table. A hypothetical smart algorithm armed with definition 5.1 will only iterate once and stop. Now, if needed, the other large item sets computed by apriori can be

computed from S just by generating all possible subsets of S and copying the frequency count of S. If S is not a large item set, so cannot be any subset of S. Apriori will discover it during the first iteration and stop, only if S is not a large item set, and so will the smart algorithm.

Going back to our example table in figure 5(a), and its equivalent table l_table in figure 1, using definition 5.1 we can easily conclude that the set of large item sets in table 5(e) are redundant. For example, the item set $\{a\}$ is a subset of $\{a, b, c\}$ and both have frequency or support count 2. This implies that there exists no other transactions that contribute to the count of a. And hence, it is redundant. On the other hand, $\{b\}$ is not redundant because conditions of definition 5.1 does not apply. And indeed we can see that $\{b\}$ is a required and non-redundant large item set because if we delete it, we cannot hope to infer its frequency count from any other large item sets in table l_table of figure 1. Similar arguments hold for other tuples in l_table of figure 1.

The non-redundant large item set table in figure 1 unearth two striking and significant facts. All the item sets that are found to be large either appear as a transaction in the t_table in figure 1, e.g., $\{b, f\}$ and $\{a, b, c\}$, or are intersections of two or more item sets of the source table not related via subsuperset relationships, e.g., $\{b\}$, which is an intersection of $\{b, e\}$ and $\{b, c, f\}$, and $\{b, e\} \not\subset \{b, c, f\}$ and $\{b, e\} \not\supset \{b, c, f\}$.

We would like to point out here that depending on the database instances it is possible that apriori will generate an optimal set of candidate sets and no amount of optimization is possible. Because in that situation, all the candidate sets that were generated would contribute towards other large item sets and hence, were required. This implies that the candidate sets were themselves large item sets by way of anti-monotonicity property of large item sets. This will happen if there are no redundant large item sets. But the issue here is that when there are redundant large item sets, apriori will fail to recognize that. In fact, FP-tree [4] and CHARM [27] gains performance advantage over apriori when there are long chains and low support threshold due to this fact. Apriori must generate the redundant set of large item sets to actually compute the non-redundant ones while the others don't.

This observation is important because it sends the following messages:

- Only the item sets that appear in a source table can be a large item set, or their meets with any other item set in the source table can be a large item set, if ever.
- There is no need to consider any other item set that is not in the source table or can be generated from the source table by computing the least fixpoint of the meets of the source item sets, as the others are invariably redundant, even if they are large item sets.
- The support count for any large item set can be obtained by adding the frequency counts of its ancestors (superset item sets) in the source table with its own frequency count.
- No item set in the item set lattice will ever contribute to the support count of any item set other than the source item sets (transaction nodes/records).

These observations readily suggest the approach we adopted here in developing a relational solution to the mining problem. All we needed was to apply a least fixpoint computation of the source items sets to find their meets. Then we applied the idea of inheritance of frequency counts of ancestors (source item sets) to other source item sets as well as to the newly generated meet item sets. It is evident that the least fixpoint of the meets we need to compute is only for the set of items that are not related by a subset superset relationship in the item set lattice.

6 An SQL3 Mining Operator

We are now ready to discuss our proposal for a mining operator for SQL3. We already know that the (non-redundant) large item sets and the (non-redundant) rules can be computed using SQL3 for which we have discussed a series of examples and expressions. We also know from our previous discussion that the method we have adopted is sound. We further know that the set of rules computed by our method is identical to the set computed by non-redundant apriori, or are equivalent to rules computed by naive apriori. So, it is perfectly all right to abstract the idea into an operator for generic use.

The mine by operator shown below will generate the l_table in figure 1. Basically, its semantics translates to the set of view definitions (or their equivalents) for n_table, f_table, int_table, i_table, c_table and l_table. However, only l_table view is returned to the user as a response of the mining query, and all the other tables remain hidden (used by the system and discarded). Notice that we have supplied two column names to the mine by operator – *Tranid* and *Items*. The *Tranid* column name instructs the system that the nesting should be done on this column and thus the support count comes from the count of *Tranid*s for any given set of *Items*. The *Items* column name suggests that the equivalent of the l_table shown in figure 1 should be constructed for the *Items* column. Essentially, this mine by expression will produce the l_table of figure 1 once we set $\delta_m = 0.25$.

select *Items*, **sup**(*Tranid*) as *Support*
from *t_table*
mine by *Tranid* for *Items*
having **sup**(*Tranid*) $\geq \delta_m$

We have also used a having clause for the mine by operator in a way similar to the having clause in SQL group by operator. It uses a function called **sup**. This function, for every tuple in the c_table, generates the ratio of the *Support* to the total number of distinct transactions in the t_table. Consequently, the having option with the condition as shown filters unwanted tuples (large item sets). The select clause allows only a subset of the column names listed in the mine by clause along with any aggregate/mine operations on them. In this case, we are computing support for every item set using the **sup** function just discussed.

For the purpose of generating the association rules, we propose the so called *extract rules using* operator. This operator requires a list of column names, for example *Items*, using which it derives the rules. Basically the expression below

produces the r_table of figure 1 for $\eta_m = 0.40$. Notice that we have used a having clause and a mine function called **conf** that computes the confidence of the rule. Recall that the confidence of a rule can be computed from the support values in the l_table – it is the (appropriately taken) ratio of the two supports.

select **ant**(*t.Items*) as *Ant*, **cons**(*t.Items*) as *Conseq*, *t.Support*, **conf**(*t.Support*)
from (select *Items*, **sup**(*Tranid*) as *Support*
from *t_table*
mine by *Tranid* for *Items*
having **sup**(*Tranid*) $\geq \delta_m$) as *t*
extract rules using *t.Items* on *t.Support*
having **conf**(*t.Support*) $\geq \eta_m$

Notice that this query is equivalent to the view definition for r_table in section 4. Consequently here is what this syntax entails. The extract rules using clause forces a Cartesian product of the relation (or the list of relations) named in the from clause. Naturally, the two attribute names mentioned in the extract rules using clause will have two copies in two columns. As explained as part of the r_table view discussion in section 4, from these four attributes all the necessary attributes of r_table can be computed even though we mention only two of the four attributes without any confusion (see r_table view definition). All this clause needs to know is which two attributes it must use from the relation in the from clause, and among them which one has the support values. The rest is trivial.

The mine functions **ant** and **cons** generates the antecedent and consequent of a rule from the source column included as the argument. Recall that rule extraction is done on a source relation by pairing its tuples (Cartesian product) and checking for conditions of a valid rule. It must be mentioned here that the **ant** and **cons** functions can also be used in the having clause. For example if we were interested in finding all the rules for which *ab* is in the consequent, we would then rewrite the above rule as follows:

select **ant**(*t.Items*) as *Ant*, **cons**(*t.Items*) as *Conseq*, *t.Support*, **conf**(*t.Support*)
from (select *Items*, **sup**(*Tranid*) as *Support*
from *t_table*
mine by *Tranid* for *Items*
having **sup**(*Tranid*) $\geq \delta_m$) as *t*
extract rules using *t.Items* on *t.Support*
having **conf**(*t.Support*) $\geq \eta_m$ and **cons**(*t.Items*) $= \{a, b\}$

It is however possible to adopt a single semantics for mine by operator. To this end we propose a variant of the mine by operator to make a syntactic distinction between the the two, called the mine with operator, as follows. In this approach, the mine with operator computes the rules directly and does not produce the intermediate large item set table l_table. In this case, however, we need to change the syntax a bit as shown below. Notice the change is essentially in the argument of **conf** function. Previously we have used the *Support* column of the l_table, but now we use the *Tranid* column instead. The reason for this choice makes sense since support is computed from this column and that support column is still

hidden inside the process and is not known yet, which was not the case for the extract rules using operator. In that case we knew which column to use as an argument for **conf** function. But more so, the *Tranid* column was not even available in the l_table as it was not needed.

select **ant**(*Items*) as *Ant*, **cons**(*Items*) as *Conseq*,
sup(*Tranid*) as *Support*, **conf**(*Support*) as *Confidence*
from *t_table*
mine with *Tranid* for *Items*
having **sup**(*Tranid*) $\geq \delta_m$ and **conf**(*Tranid*) $\geq \eta_m$

Here too, one can think of appropriately using any of the mine functions in the having clause to filter unwanted derivations. We believe, this modular syntax and customizable semantics brings in strength and agility in our system.

We would like to point out here that while both the approaches are appealing, depending on the situations, we prefer the first approach – breaking the process in two steps. The first approach may make it possible to use large item sets for other kind of computations that were not identified yet. Conversely speaking, the single semantics approach makes it difficult to construct the large item sets for any sort of analysis which we believe has applications in other system of rule mining.

7 Optimization Issues

While it was intellectually challenging and satisfying to develop a declarative expression for association rule mining from relational databases using only existing (or standard) object relational machinery, we did not address the issue related to query optimization in this article. We address this issue in a separate article [6] for the want of space and also because it falls outside the scope of the current article. We would like to point out here that several non-trivial optimization opportunities exist for our mining operator and set value based queries we have exploited. Fortunately though, there has been a vast body of research in optimizing relational databases, and hence, the new questions and research challenges that this proposal raises for declarative mining may exploit some of these advances.

There are several open issues with some hopes for resolution. In the worst case, the least fixpoint needs to generate n^2 tuples in the first pass alone when the database size is n - which is quite high. Theoretically, this can happen only when each transaction in the database produces an intersection node, and when they are not related by subset-superset relationship. In the second pass, we need to do n^4 computations, and so on. The question now is, can we avoid generating, and perhaps scanning, some of these combinations as they will not lead to useful intersections? For example, the node c_3^0 in figure 3 is redundant. In other words, can we only generate the nodes within the sandwich and never generate any node that we would not need? A significant difference with apriori like systems is that our system generates all the item sets top down (in the lattice) without taking

their candidacy as a large item set into consideration. Apriori, on the other hand, does not generate any node if their subsets are not large item sets themselves, and thereby prunes a large set of nodes. Optimization techniques that exploit this so called "anti-monotonicity" property of item set lattices similar to apriori could make all the difference in our setup. The key issue would be how we push the selection threshold (minimum support) inside the top down computation of the nodes in the lattice in our method. Technically, we should be able to combine view int_table with i_table, c_table and l_table and somehow not generate a virtual node that is outside the sandwich (below the z-envelope in figure 3). This will require pushing selection condition inside aggregate operations where the condition involves the aggregate operation itself.

For the present, and for the sake of this discussion, let us consider a higher support threshold of 45% (3 out of 7 transactions) for the database **T** of figure 1. Now the l-envelope will be moving even closer to the z-envelop and the nodes bf_1^2 and d_2^1 will be outside this sandwich. This raises the question, is it possible to utilize the support and confidence thresholds provided in the query and prune candidates for intersection any further? Ideas similar to magic sets transformation [2,26] and relational magic sets [15] may be borrowed to address this issue. The only problem is that pruning of any node depends on its support count which may come at a later stage. By then all nodes may already have been computed. Specifically, pushing selection conditions inside aggregate operator may become challenging. Special data structures and indexes may perhaps aid in developing faster methods to compute efficient *intersection joins* that we have utilized in this article. We leave these questions as open issues that should be taken up in the future.

Needless to emphasize, a declarative method, preferably a formal one, is desirable because once we understand the functioning of the system, we will then be able to select appropriate procedures depending on the database instances to compute the relational queries involving mining operators which we know is intended once we establish the equivalence of declarative and procedural semantics of the system. Fortunately, we have numerous procedural methods for computing association rules which complement each other in terms of speed and database instances. In fact, that is what declarative systems (or declarativity) buy us – a choice for the most efficient and accurate processing possible.

8 Comparison with Related Research

We would like to end our discussion by highlighting some of the contrasts between our proposal and the proposals in [25,21,13,14]. It is possible to compute the rules using the mine rule operator of [13,14] as follows.

mine rule *simpleassociation* as
select distinct 1..n *Items* as body, 1..n *Items* as head, support, confidence
from *t_table*
group by *Tranid*
extracting rules with support: δ_m, confidence: η_m

As we mentioned before, this expression is fine, but may be considered rigid and does not offer the flexibilities offered by our syntax. But the main difference is in its implementation which we have already highlighted in section 1.1.

It is relatively difficult to compare our work with the set of works in [25,21,20] as the focus there is somewhat different, we believe. Our goal has been to define a relational operator for rule mining and develop the formal basis of the operator. Theirs, we believe, was to develop implementation strategies of apriori in SQL and present some performance metrics. As long as the process is carried out by a system, it is not too difficult to develop a front end operator that can be implemented using their technique. Even then, the only comparison point with our method would be the execution efficiency. In fact, it might become possible to implement our operator using their technique.

9 Conclusions

It was our goal to demonstrate that association rules can be computed using existing SQL3 machineries, which we believe we have done successfully. We have, of course, used some built-in functions for set operations that current SQL systems do not possibly support, but we believe that future enhancements of SQL will. These functions can be easily implemented using SQL's create function statements as we have done. We have utilized SQL's create view recursive clause to generate the intersection nodes which was implemented in PL/SQL.

If one compares the SQL3 expressions presented in this article with the series of SQL expressions presented in any of the works in [25,21,13,14] that involve multiple new operators and update expressions, the simplicity and strength of our least fixpoint based computation will be apparent. Hence, we believe that the idea proposed in this article is novel because to our knowledge, association rule mining using standard SQL/SQL3 is unprecedented. By that, we mean SQL without any extended set of operators such as combination and GatherJoin [25,21], or the *CountAllGroups, MakeClusterPairs, ExtractBodies,* and *ExtractRules* operators in [13,14].

Our mine by operator should not be confused with the set of operators in [25,21] and [13,14]. These operators are essential for their framework to function whereas the mine by operator in our framework is not necessary for our mining queries to be functional. It is merely an abstraction (and a convenience) of the series of views we need to compute for association rule mining. The method proposed is soundly grounded on formal treatment of the concepts, and its correctness may be established easily. We did not attempt to prove the correctness for the sake of conciseness and for want of space, but we hope that readers may have already observed that these are just a matter of details, and are somewhat intuitive too.

The mine by operator proposed here is simple and modular. The flexibilities offered by it can potentially be exploited in real applications in many ways. The operators proposed can be immediately implemented using the existing object

relational technology and exploit existing optimization techniques by simply mapping the queries containing the operators to equivalent view definitions as discussed in section 4. These are significant in terms of viability of our proposed framework.

As a future extension of the current research, we are developing an efficient algorithm for top-down procedural computation of the non-redundant large item sets and an improved SQL3 expression for computing such a set. We believe that a new technique for computing item set join (join based on subset condition as shown in the view definition for int_table) based on set indexing would be useful and efficient. In this connection, we are also looking into query optimization issues in our framework.

References

1. Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.
2. C. Beeri and R. Ramakrishnan. On the power of magic. In *ACM PODS*, pages 269–283, 1987.
3. Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proc. ACM SIGMOD*, pages 265–276, 1997.
4. Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD*, pages 1–12, 2000.
5. H. M. Jamil. Mining first-order knowledge bases for association rules. In *Proceedings of the 13th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 218–227, Dallas, Texas, 2001. IEEE Press.
6. H. M. Jamil. A new indexing scheme for set-valued keys. Technical report, Department of Computer Science, MSU, USA, June 2001.
7. Hasan M. Jamil. Ad hoc association rule mining as SQL3 queries. In *Proceedings of the IEEE International Conference on Data Mining*, pages 609–612, San Jose, California, 2001. IEEE Press.
8. Hasan M. Jamil. On the equivalence of top-down and bottom-up data mining in relational databases. In *Proc. of the 3rd International Conference on Data Warehousing and Knowledge Discovery (DaWaK 01)*, pages 41–50, Munich, Germany, 2001.
9. Hasan M. Jamil. Bottom-up association rule mining in relational databases. *Journal of Intelligent Information Systems*, 19(2):191–206, 2002.
10. Mika Klemettinen, Heikki Mannila, Pirjo Ronkainen, Hannu Toivonen, and Inkeri Verkamo. Finding interesting rules from large sets of discovered association rules. In *CIKM*, pages 401–407, 1994.
11. Flip Korn, Alexandros Labrinidis, Yannis Kotidis, and Christos Faloutsos. Ratio rules: A new paradigm for fast, quantifiable data mining. In *Proc of 24th VLDB*, pages 582–593, 1998.
12. Brian Lent, Arun N. Swami, and Jennifer Widom. Clustering association rules. In *Proc of the 3th ICDE*, pages 220–231, 1997.
13. Rosa Meo, Giuseppe Psaila, and Stefano Ceri. A new SQL-like operator for mining association rules. In *Proc of 22nd VLDB*, pages 122–133, 1996.

14. Rosa Meo, Giuseppe Psaila, and Stefano Ceri. An extension to SQL for mining association rules. *DMKD*, 2(2):195–224, 1998.
15. Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. In *ACM SIGMOD*, pages 103–114, 1994.
16. Amir Netz, Surajit Chaudhuri, Jeff Bernhardt, and Usama M. Fayyad. Integration of data mining with database technology. In *Proceedings of 26th VLDB*, pages 719–722, 2000.
17. Amir Netz, Surajit Chaudhuri, Usama M. Fayyad, and Jeff Bernhardt. Integrating data mining with SQL databases. In *IEEE ICDE*, 2001.
18. Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained association rules. In *Proc. ACM SIGMOD*, pages 13–24, 1998.
19. Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In *Proc. ACM SIGMOD*, pages 175–186, 1995.
20. Karthick Rajamani, Alan Cox, Bala Iyer, and Atul Chadha. Efficient mining for association rules with relational database systems. In *IDEAS*, pages 148–155, 1999.
21. Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. Integrating mining with relational database systems: Alternatives and implications. In *Proc. ACM SIGMOD*, pages 343–354, 1998.
22. Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc of 21th VLDB*, pages 432–444, 1995.
23. Pradeep Shenoy, Jayant R. Haritsa, S. Sudarshan, Gaurav Bhalotia, Mayank Bawa, and Devavrat Shah. Turbo-charging vertical mining of large databases. In *ACM SIGMOD*, pages 22–33, 2000.
24. Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, third edition, 1996.
25. Shiby Thomas and Sunita Sarawagi. Mining generalized association rules and sequential patterns using SQL queries. In *KDD*, pages 344–348, 1998.
26. J. D. Ullman. *Principles of Database and Knowledge-base Systems, Part I & II*. Computer Science Press, 1988.
27. Mohammed J. Zaki. Generating non-redundant association rules. In *Proc. of the 6th ACM SIGKDD Intl. Conf.*, Boston, MA, August 2000.