CERIA Research Report 2005-05-15, May 2005

Galois Connections, T-CUBES & Database Mining

Witold Litwin[1]

Abstract

Galois connections are the bread and butter of the formal concept analysis. The universe of discourse there is the collections of objects with properties. A property corresponds typically to a single-valued (.true or null) binary attribute of the object. Probably the most studied connections are the closed sets and Galois lattices. We propose a generalization of these connections to the relational database universe with the multi-valued domains. We show the database mining queries that appear from, hard or impossible to formulate with SQL at present. We further show that the groups produced by the popular CUBE operator form a type of multi-valued closed sets, including the traditional binary closed sets. To deal with more types of closed sets, we generalize CUBE to a new operator we call θ-CUBE, writing it T-CUBE. This one calculates the groups according to all the values of the θ operator popular with the relational joins, namely $\theta = \{=, \leq, <, <>, \geq, >\}$. The $\theta =$ '=' defines the CUBE. The generalization applies naturally also to GROUP BY and to the other popular SQL grouping operators. We further show the utility of the aggregate function LIST and of a new one that we have named T-GROUP. The couple conveniently aggregates the composition of a closed set produced by T-CUBE into a single tuple. We finally discuss the algorithms for the computation of the queries involving the closed sets. Unlike perhaps for the concept analysis, the scalable distributed algorithms in P2P or grid environment appear the most useful for the database mining in practice. Our proposals should benefit to both the database mining and the concept analysis over larger collections of objects.

1 Introduction

The formal concept analysis studies the relationship between objects and the properties in a space of objects and properties. The Galois connection in this universe is a relationship among some objects and some properties. Whether an object has a property is typically indicated by a binary attribute of the object. Probably the most studied Galois connection is that among a set **O** of all the objects sharing some set of properties **P**, such that there is no property in the space beyond **P** that would be also shared by all the objects in **O**. One qualifies such sets as *closed sets*. The closed sets over the subsets of a set of objects sharing a set of properties can be ordered by inclusion over **P** or **O.** A popular result is a *Galois lattice*. Finding a closed set let us conclude about the maximal common set of properties. The set may be then abstracted into a concept. The lattice calculus let us see possible abstractions among the concepts.

As a simple motivating example, the objects may be the students for some diploma, e.g., DESS 220 at Dauphine University. A property may be the final "pass" grade at a course. A closed set would be any couple (*S*, *P*) such that *S* contains all the students who passed all the courses in *P*, and, for any other course, at least one of the students in *S* failed. It could be for instance students s_1,s_2,s_3, who all passed courses c_1,c_3,c_5, while s_1 or s_2 or s_3 failed on any other course. The Galois lattice would show the inclusion connections. It could show for instance for our closed set that students s_1,s_3 form also a close set over more courses which are c_1,c_3,c_5,c_{12}. The set of students passing all the exams (one extremity of the lattice) may or may not be empty. Likewise, there may or may not be the course that all students pass (the other extremity). The dean may obviously be interested in mining the resulting (Galois)

[1] CERIA, Université Paris Dauphine, Pl. du Mal. de Lattre, 75016 Paris, France, mailto:WitoldLitwin@dauphine.fr

connections, i.e., searching for some closed sets or selected parts of the lattice. Obviously if a close set includes many more students and much fewer courses than any other, these courses are perhaps a little too easy.

There were interesting applications of the concept analysis to databases, e.g., to mine for ISA relationship, [H05]. Our example shows however also that the theory of Galois connections over binary attributes is intrinsically of limited value for that field. After all, courses usually have multi-valued grades, e.g., 0 to 20 in Dauphine. It is a truism to say that relational databases deal almost exclusively with multi-valued attributes. If the concept analysis should apply to databases at a larger scale, there is a need for the related generalization of its universe of discourse. The need may obviously concern applications to other domains. The latter triggered already two generalization attempts, [DE97], [G01], and a trend called *scaling* that instead attempt to map the multi-valued attributes into binary ones. All these proposals were not specific to databases and, as we show later, were in fact too limited for our goal. For instance, the scaling would typically require a change to the database schema under consideration. The proposals in [DE97] would be limited to attributes with totally ordered domain, and, even in this case, would present other important limitations. Like [G01] besides, for other reasons.

We therefore propose a different generalization, tailored specially for the database needs. As the result, we let to mine for the Galois connections using SQL. We mine for the closed sets especially through a perhaps surprising relation between this concept and that of a grouping realized by SQL queries through the popular GROUP BY, CUBE etc. operators. As the result, we propose a generalization of these operators and a new aggregate function. The database mining gains then new kinds of queries inherited from the work on the concept analysis, which are hard or impossible to achieve with SQL at present. In turn, the concept analysis inherits in this way the database techniques for mining large collections of data. These should help that domain, whose algorithms are basically limited in practice at present to relatively small collections only, i.e., dozens of objects usually and hundreds at most.

To give an idea of our approach, observe that the notion of sharing a multi-valued property may be given various meanings. Observe further, on this basis, that the basic operation in the relational database universe of discourse, that is an equi-join, is in fact a property value sharing in some sense. The meaning is that the values of the join attributes are equal through the "=" comparison operator. More generally, the sharing may concern any operator in the set $\theta = \{=, \leq, <, <>, \geq, > \}$. This is the idea in the θ-joins. The meaning of "≤" operator for instance is that an object with a property symbolized by a join attribute value v, shares this property with any other object whose value v' of the join attribute is such that $v \leq v'$. Hence, joins seen under this angle are also some Galois connections. Notice in particular, in the light of the comments to [DE97] above, that the use of operators "=" and "<>" does not require a total order on the domain of the join attributes.

Next, observe that the popular GROUP BY operator, also explores the '=' Galois connection among the objects (tuples) that it groups over a given set of (grouping) attributes. More generally, the popular CUBE operator groups over any subset of the set of the grouping attributes, but also along the same '=' connection, [G&al97]. A group T over some of the attributes under the CUBE, together with every other column forming a group over some other attributes of the tuples in T, if there are any such columns, form a closed set in this sense. Hence, CUBE is the basis for some closed sets computation and analysis. Subsequently, one may order the closed sets by inclusion, getting a generalized Galois lattice.

The generalization of the grouping operators we propose below follows this approach. In short, the new operators group for the other θ-operators as well. We may then calculate the

closed sets also over the other θ values. The idea is somehow similar to that in θ-joins with respect to the '='-join (equi-join). The new operator generalizing the CUBE for instance, groups as CUBE, but over the other θ values as well, even different ones combined in a single grouping operation. We term it T-CUBE and read θ-CUBE. It. We act similarly with respect to GROUP BY, ROLLUP and GROUPING SETS, proposing the T-GROUP BY etc. The Galois lattices can be further formed by inclusion over the closed sets.

Like CUBE, T-CUBE does not show explicitly the tuples it has grouped to form a closed set. We combine for this purpose the operator with the LIST aggregate function [L03]. To show the attributes of the closed set, we propose a new aggregate function we call T-GROUP. The function renders an aggregated value at an attribute iff given tuples form a group at that attribute over given θ. Otherwise, the result of T-GROUP is null. As typically for the databases, we can further combine the mining of a closed set with that of the attributes not in the set. We can apply any SQL aggregate function to these attributes, as well as apply aggregate functions other than T-GROUP to the attributes forming the closed sets.

To come back to our motivating example, to use '=' operator only, hence CUBE in fact, let us to find the closed sets of students sharing the same grades. In particular, we can find the usual single-valued closed sets. We can further compute the averages, standard deviations etc on these or other grades for a multi-valued closed set. As we use SQL, we can restrict our attention to only some but not all closed sets, e.g., those with all grades above 10. The restrictions may greatly speed up the query execution, with respect to the search for all the closed sets. If we choose θ = '≥', then we determine the closed sets of students sharing at least some grade. This is in essence, by the way, the idea in the generalization proposed in [diday], and the related algorithms subsequently developed in [_]. If we choose in turn θ = '≤', we get the closed sets of students sharing at most some grade. And so on. Notice that the choice of θ = '<>' requires rather more thoughts about what the concept of closed set may then mean.

One knows well that CUBE operator is hard to evaluate. It should be obviously even harder therefore to evaluate T-CUBE. The proposed concepts can thus be useful in practice, only if one finds the efficient way of executing T-CUBE queries.

A variety of algorithms for CUBE have been proposed. One has to revisit them thus for '=' based closed sets calculus. For the '≥' operator, algorithms for the generalized closed sets computation on a single CPU have been proposed, [ELL97], [BL03]. One, called ELL, appears the fastest for the centralized calculus. The known algorithms, ELL in particular thus, should be revisited to see whether they generalize efficiently to other θ values. They should also be re-engineered to accept the restrictions (selections and projections especially), to avoid the costly production of uselessly many closed sets. Finally, one should similarly revisit the algorithms generating the '≥'-generalized Galois lattices.

While for the concept analysis, the centralized algorithms seem to suffice often, it should not be usually the case for the database mining. It seems rather clear from our motivating example. One way towards the goal should be to distribute the calculus. The distribution should however be possibly such that a closed set can be determined without the communication with other side. Next, as we do not know in advance how many closed sets will result, we need a scalable distributed storage for those.

We show below one general basis for the scalable distributed calculus. That one uses a distributed hash partitioning schemes. Other schemes are possible, especially for specific Θ-values. We mention such schemes, easily derived from the CUBE calculus for '=', and from the ELL for '≥'.

Below we first recall the formal definitions of the closed sets and of Galois lattices. Next we define the syntax and semantics of the T-CUBE. We then show some motivating examples. Finally, we discuss the scalable distributed computation of our closed sets in P2P or grid environment. We conclude with the direction for the future work.

2 Galois Connections

The Galois connections are the mathematical framework for extracting concepts and rules from other concepts. The closed sets and the Galois lattices are the most popular related notions. Formally, all these notions are constructs of the lattice theory [B67]. We now recall the related basics.

A *formal context is* a triple of sets (O,A,I), where O is a set of *objects*, A is a set of *attributes*, and I is a binary relation between A and O ($I \subseteq O \times A$). The notation $o\ I\ a$, $o \in O$, $a \in A$, indicates that $(o,a) \in I$. For $O_1 \subseteq O$, let O_1' be the set of the common attributes to all these objects, i.e., $O_1' = \{a \in A \mid o\ I\ a \text{ for each } o \in O_1\}$. Vice versa, for $A_1 \subseteq A$, let A_1' be all the objects verifying all the attributes of A_1, i.e., $A_1' = \{o \in O \mid o\ I\ a \text{ for each } a \in A_1\}$. A *Galois connection* (GC) is the pair of mappings that we denote (f, g), over P(O) and P(A), where: f: $O_1 \rightarrow O_1'$ and g: $A_1 \rightarrow A_1'$. The couple (O_1, A_1) where $O_1 = A_1'$ and $A_1 = O_1'$ constitutes a *closed set*, (CS), also called *concept*. The set O_1 (vs. A_1) is called *extent* (vs. *intent*) of (O_1, A_1).

We usually represent an object o as a tuple with an OID attribute and the binary attributes a that evaluate to *.true* or '1' iff ($o\ I\ a$). Thus $a = 1$ means that o possesses the property represented by a. Fig.1 is an example of a formal context represented in this way, with O = (1,2,3,4,5,6,7) and A = (a,b,c,d,e,f,g,h).

The (double) inclusion relation we denote as "$\leq$" defines furthermore a partial order of all the concepts over a formal context:

(O_1,A_1) $\leq$ (O_2,A_2) $\Leftrightarrow$ $O_1 \subseteq O_2$ and $A_1 \supseteq A_2$ $\Leftrightarrow$ (O_1, A_1) is a *subconcept* of (O_2,A_2) $\Leftrightarrow$ (O_2, A_2) is a *superconcept* of (O_1,A_1).

O \ A	a	b	c	d	e	f	g	h
1	1	1	1	1	1	1	1	
2	1	1	1	1	1	1		1
3	1	1	1	1	1		1	1
4	1	1	1	1		1		
5	1	1		1	1		1	
6	1	1	1		1			1
7	1		1			1		

Fig. 1 A formal context C

A sub-concept corresponds thus to a concept specialization, with possibly less objects, but with possibly more .true attributes in common (common properties). A super-concept corresponds to the opposite, i.e., realizes an abstraction of its sub-concepts. A *Galois lattice* (GL) is finally the set of all the concepts, ordered by the relation $\leq$. Fig.2 shows the GL over the formal context at Fig. 1. Notice that we represent the sub-concepts above the super-concepts. Several algorithms determine the CSs, or a GL over a formal context [B86] [G84] [GM95]. In practice, these algorithms work only for contexts of a few hundreds of objects at most, i.e., small from the database mining perspective.

2.2 *Multi-Valued Galois Connections*

A relational database contains tuples whose attribute domains are typically multi-valued. This makes the notion of a GC and the related apparatus of the concept analysis theory basically inappropriate for the database management. The starting point of any application of these notions to the databases has to be some generalization to the multi-valued domain. We now propose such a generalization. First, we now talk about *tuples*, to designate our objects of interest. We begin with the definition of what it may mean that two tuples share a property defined by some attribute (column) values. Observe, that the fundamental notion of θ-join already defines some meaning of such sharing for the join attributes. We derive our approach from these premises as follows. We consider for each attribute a *grouping* operator θ, $\theta \in \{=, \leq, <, <>, \geq, >\}$. Let $a(t)$ be an attribute value of tuple t. Then, given θ, a set T of tuples t share the property defined by one or more following attribute values c:

$\theta =$ '=' $\Rightarrow T = \{t : a(t) = c\}$.

$\theta =$ '≤' $\Rightarrow T = \{t : a(t) \leq c\}$.

$\theta =$ '<' $\Rightarrow T = \{t : a(t) < c\}$.

$\theta =$ '<>' $\Rightarrow T = \{t : a(t) <> c\}$.

$\theta =$ '≥' $\Rightarrow T = \{t : a(t) \geq c\}$.

$\theta =$ '>' $\Rightarrow T = \{t : a(t) > c\}$.

Fig. 2 Galois lattice for C

Accordingly, if we choose c at some attribute x, we say that T forms a *θ-group* for c, if for the θ value chosen at x, every t in T shares c value according to the above rules. In the vocabulary of θ- joins, a tuple t belongs to a group formed at x iff $t.x\ \theta\ c$ =.true. We then call c a *common property* of group T. As usual, one can define the grouping at several attributes, as the intersection of the groups at each attribute chosen.

Given for each x, a choice of θ and of some c, present in some tuple, we say that T and some set A' of its attributes form a *θ-closed set* (T, A') if T contains all and only tuples forming the groups for each attribute in A', and there is no group involving all the tuples of T on an attribute not in A'. We abbreviate the term generalized closed set to θ-CS. The notion of a *multi-valued θ-Galois connection* (θ-GC) and of a *multi-valued θ-Galois lattice* (θ-GL) define accordingly. Notice however that, unlike in a GL, we may have several lower extremities in a θ-GL. We may have now several sets of objects sharing each all the attribute values, but different among the sets.

The choice of c for defining a θ-group and a θ-CS is the usual one θ = '='. To deal also with the other θ values, we generalize the usual approach to the following rule. Example 1 below illustrates its rationale and application.

We choose some tuple s, termed *seed*, in the table subject to the grouping. The attributes of s define as follows the values of c for the groups at selected attributes. Let $s.x$ be the value of the attribute x of s. Let T be the group formed using some θ values at some attributes of s. Let $\theta(x)$ denote θ used at attribute x. Then, if $s.x$ defines a group for $\theta(x)$ over tuples in T, we set $c = s.x$. We form A' from all the groups formed in this way over s.

We represent (T, A') for a relational calculus basically as a tuple $t = (T, v(A))$, where for each attribute $x \in A'$, $t.x = c$ otherwise $t.x$ = '_', i.e., is null. We qualify this representation of a θ-CS as *basic*. Observe that our rule for $v(A)$ defines in fact a specific aggregate function, in the common, relational database, sense. We consider therefore furthermore also the *aggregate* representations of a θ-CS, where one forms $t.x$ using a different aggregate function, perhaps more convenient.

We now illustrate our notions of θ- groups and of θ-CS, and their interest for the database mining. We leave the analysis of the θ-GLs as the future work.

Example 1

1. Consider table S of DESS 220 students in Fig. 3, with the key S#, and with the grades[2] per course *a,b…h*. We wish to find any students with the grades of student 1 for course *a*. If so, we wish to mine for the courses where these students also share the respective grade student 1.

S#	a	b	c	d	e	f	g	h
1	12	16	6	10	12	13	12	6
2	12	12	8	12	11	13	11	8
3	10	13	16	14	11	8	12	10
4	13	14	11	10	13	13	12	14
5	17	10	10	14	13	10	14	12
6	0	14	3	8	4	13	11	10

Fig. 3 Table S

To get the answer to our query, we now consider θ = '=' at all the attributes of S. The students {1, 2} form the group at *a* sharing c = 12. These students form also the groups at *f*, for c = 13. The couple ({1, 2}, {a = 12, f = 13}) form a θ-CS, namely an '='-CS, that responds to our query. We aggregate this θ-CS to the tuple compatible with the original table which is tuple ({1, 2},12,_ ,_,_,_,12,_,_}). This is the basic representation of this CS.

As we show in next section, no current SQL dialect let us to express the above query. Notice also that the calculus would apply to the single-valued attributes, e.g., to pass an exam for a student. It would lead to the usual CSs.

2. We now wish to issue the similar query, but for any student and any course. Several other θ-CSs appear in reply to our query. For instance:

({1,2,4,6}, {f = 13}), ({1,3,4}, {g = 12}), ({4, 6}, {b = 14, f = 13}), ({1, 4}, {d = 10, f = 13, g=12})…

[2] Actual ones, in fact.

3. Consider now that we wish to mine every student at least as good as student 1 for course *a*, and perhaps for other courses that we wish to mine therefore as well. We thus naturally assume the ascending order on the grades. We apply θ = '≥' at all the columns, perform the grouping on column *a*, for *a* = 12, and find *c* for the groupings at other columns. We get the following θ-CS:

({1, 2,4,5},12,_ ,6,10,_,_,_,6})

Alternatively, we may wish to know for every course the least grade of these students, i.e., the minimal competence level they share. We may then apply to each column the aggregate function $c = \min_T(x)$, where *T* denotes the students in the group of student 1 over *a*. Our result is now:

({1, 2,4,5},12,10 ,6,10,11,10,11,6}).

This is the aggregated representation of our CS. The result meets the definition of a CS over multi-valued attributes proposed in [DE97], termed a *generalized* CS. The algorithms in [ELL97], compute CSs accordingly. The example illustrates that the definition in [DE97] is a specific case of ours, limited to a '≥'-CS, in our vocabulary. In contrast, [DE97] mentions generalization directions we do not deal with, e.g., towards the fuzzy sets.

Yet alternatively, for our group {1, 2,4,5}, and column *b* for instance, one could mine instead rather for avg (*b*). The result *b* = 13 would give yet another idea of the difference between the grade of student 1 and of the other students for this course.

Like for query (2) above, we may wish to ask furthermore for students who are at least as good as any other student for any set of courses. The set of all θ-CSs will be the answer. In this example its cardinal is in fact 39. The calculus for the actual course, with in fact only about 20 students, showed already over 6.000 θ-CSs. This number alone seems too high for a practical purpose. The related mining seems to make sense only with some additional aggregations or restrictions.

If we were interested by the students at most as good as student 1, or at most as good as any other student etc., we would use θ = '≤' at all columns and, perhaps, $c = \max_T(x)$. Observe that, for both values of θ, no current SQL dialect would again let us to express the discussed queries.

4. Consider now that we are interested in students that are strictly better than student 1, on column *a*, if there are some, and possibly at some others. We may choose θ = '>' for all the columns. There could be no group at column *a*, but in our example we have one that is {4,5}. We now get now the following CS:

({4,5},12,_, _,_,12, _,_,_}).

Notice that the seed, i.e., the tuple of student 1, is here not in the group it leads to.

5. We may wish to mine alternatively in this way the grades of all the students with the grade simply different from that of student 1 for course *a*, and perhaps some others. We choose θ = '<>' for all the columns. We get the group {3,4,5,6} at column *a*, assuming that 0 is not a null grade (otherwise the group would be {3,4,5} only. The values of attributes *b,c,e,h* in the group match our θ. It is not the case for the others. Accordingly, we get the <>-CS:

({3,4,5,6}, 12, 16, 6,_,12,_,_,6).

6. Finally, we may wish to mine again for the students who are as good as student 1 for course *a*, but also for their other courses, where they are at least as good as student 1. We apply θ = '=' to form the group at *a*, and θ = '≥' to all the other columns. The resulting group is {1,2} as for case (1) above. But the θ-CS becomes:

({1, 2},12,_ ,6,10,_,13,_,6}).

If we mined instead for students as good as student 1 for course *a*, but better than for other courses, we would apply θ = '>' to the other columns and get the θ-CS:

({1, 2},12,_ ,8,12,_,_,_,8}).

3 *T-CUBE*

The notion of grouping is popular with the database management since the operator GROUP BY was introduced. It calculates a single group over some columns. It uses Θ = '=' in our vocabulary. Other well-known operators for multiple groupings followed. Among them the CUBE is the most general at present. We recall that CUBE ($a_1 \ldots a_n$) calculates all the groups over all the subsets of $\{a_1 \ldots a_n\}$ for Θ = '=' in our vocabulary. The CUBE appears as the natural basis for the calculus of θ-GCs in general, and of θ-CSs specifically. Notice that the notion of θ-CS appears as a natural 2-d extension of that of a group. It operates indeed not only on the tuples common to the group, but also on the common properties.

For our purpose, one has to first generalize the CUBE operator so to accept the other values of θ as the grouping criteria. Next, one has to accompany it with aggregate functions that show the θ-CS composition, with respect to both the tuples and the attribute values. Notice that CUBE by itself does not provide the group composition at present. Our proposal is as follows.

We consider a new operator we call T-CUBE. The operator allows for any θ at any of the columns it should operate upon. It performs the multiple groupings accordingly, as illustrated by Example 1 above. For θ = '=', T-CUBE reduces to CUBE. Over the groups provided by T-CUBE, we use basically two following aggregate functions to form the θ-CS tuple:

1. The LIST (I) function, [L03], where I indicates the attribute(s) chosen to identify each tuple composing the θ-CS.

2. The T-GROUP (θ, *x*) function. This function tests whether column *x* forms a group, given a seed. If so, it renders *c* as the result, otherwise it evaluates to '_'. In the latter case, it may invoke another aggregate on the column.

For convenience, we consider T-GROUP (*x*) = T-GROUP (=, *x*). We also consider T-GROUP implicit for every attribute without a value expression named in the Select clause and T-CUBE. T-GROUP inherits then for each attribute its θ under T-CUBE. The function is also implicit for attributes without a value expression and not in the T-CUBE, provided T-CUBE uses of only one θ value.

We also consider that T-CUBE allows for a restriction on attributes *X* that is enforced only when *X* are processed as the grouping attributes. For instance, in our motivating example, one may indicate that, for query *Q*, only the values of *a* above 0 are of interest for θ = '≥' groups to form at this attribute. When T-CUBE evaluation in *Q* leads in turn however to the grouping on attribute *b* only, again just for instance, then *Q* concerns potentially all the values of *a*. Hence, we may a get a θ-CS involving student 6 with 0 grade.

Finally, as we allow for $\theta \neq$ '=' for T-CUBE, we implicitly generalize similarly the GROUP BY, ROLLUP and GROUPING SETS clauses. That is, we may invoke in a query, respectively, T-GROUP BY, T- ROLLUP or T- GROUPING SETS clauses.

Example 2

1. The following syntax expresses the intention in T-CUBE "by example", as well as some queries in Example 1 and more.

- T-CUBE (*a,b,c*) means CUBE (*a,b,c*).
- T-CUBE (≥,*a,b,c*) means the groupings using θ = '≥' at all three attributes. Hence, T-CUBE (=,*a,b,c*) means again CUBE (*a,b,c*).
- T-CUBE (≥*a, =b, >c*) means the groupings using the indicated different θ values at each attribute.
- T-CUBE (=,*a* : *a* <> 0,*b,c*) means we are interested in groups formed at *a* only for *a* not 0.
- T-CUBE (=,*a* : select a from S where *a* <> 0,*b,c*) means the same.

2. The following query produces the mining (1) requested in Example 1. The function T-CUBE is implicit at attributes a…h.

```
Select List (s#),a,b,c,d,e,f,g,h
from S
T-CUBE (a)
having a = 12
```

Here the functions T-GROUP (=, a), T-GROUP (=, b)… T-GROUP (=, h) are implicit. We could write the query alternatively as:

```
Select List (s#),a,b,c,d,e,f,g,h
from S
T-GROUP BY (a)
having a = 12
```

We could also write it in any of the following forms:

```
Select List (s#),a,b,c,d,e,f,g,h
from S
CUBE (a)
having a = 12
```

```
Select List (s#), T-GROUP (=,a), T-GROUP (=,b), T-GROUP (=,c), T-GROUP (=,d), T-
GROUP (=,e), T-GROUP (=,f), T-GROUP (=,g), T-GROUP (=,h)
from S
where a = 12
```

```
Select List (s#), T-GROUP (a), T-GROUP (b), T-GROUP (c), T-GROUP (d),
T-GROUP (e), T-GROUP (f), T-GROUP (g), T-GROUP (h)
from S
GROUP BY (a)
having a = 12
```

The last three formulations are the easiest to implement with the current DBMS technology. It suffices that the DBMS lets to define external functions.

3. The following query mines for the students as wished in 2nd part of (2) in Example 1, i.e., for every CS formed by the students at least as good as some student at some course. Besides, it calculates the minimal grades at each column for each group, getting thus the generalized CS of

[DE97]. It also adds the average values for *b* at each group, as wished in Example 1. Finally, it restricts our mining to CSs formed from at least 2 members.

Select List (s#), min (a), min (b), min (c), min (d), min (e), min (f), min (g), min (h), avg(b)
from S
T-CUBE (≥, a,b,c,d,e,f,g,h)
Having count(*) > 1;

Alternatively, we may mine for all CSs of students possibly as good as some other student at selected courses etc., as also wished in Example 1:

Select List (s#), max (a), max (b), max (c), max (d), max (e), max (f), max (g), max (h)
from S
T-CUBE (≥, a,b,c,d,e,f,g,h)
Having count(*) > 1;

4. The following query mines the students as in point (4) in Example 1, i.e., the students strictly better than student 1 at least course *a*:

Select List (s#),a,b,c,d,e,f,g,h
from S
T-CUBE (>, a)
having a = 12

5. The following straightforward queries mine finally for the students wished in (5) and (6) in Example 1:

Select List (s#),a,b,c,d,e,f,g,h
from S
T-CUBE (<>, a)
having a = 12

Select List (s#),a,b,c,d,e,f,g,h
from S
T-CUBE (=a, ≥b,≥c, ≥d, ≥e, ≥f, ≥g, ≥b)
having a = 12

4 T-CUBE Query Evaluation

One knows well that CUBE may generate a large number of groups, perhaps prohibitive in practice. T-CUBE may obviously generate even more groups, and larger ones. We have already shown that even for a small collection of tuples, one can run into the calculus of too many θ-CSs potentially. Efficient algorithms for T-CUBE queries are therefore crucial. One obvious processing rule is that restrictions should be typically processed at earliest. Besides however, it strongly seems that for the database mining as we discussed, involving thus a large number of objects, the scalable distributed (SD), P2P or grid, calculus, should usually be the basic choice, perhaps the only in practice. We thus analyse now some possibilities that open up for T-CUBE and θ-CS calculus more specifically. We recall that for the latter, T-CUBE should be coupled with LIST and T-GROUP functions.

With respect to this aspects, even '='-CS calculus is at present impossible at major DBMSs. We are not aware of any implementation of T-GROUP function as yet. A single-attribute LIST function, limited in this way with respect to the proposal in [L03], is standard at SQL Anywhere Studio, but this DBMS does not offer CUBE. Tentative implementations of single-attribute LIST as user-defined functions were also attempted under Oracle. Summing, at least

T-GROUP has to be added to present DBMSs. To start with, it should be a user-defined function on DBMSs with this capability and CUBE operator. These are, in the commercial word, e.g., Oracle and DB2. SQL Server should get user-defined functions in the next release (Yukon). Fortunately, one may expect the task of creating T-GROUP as a user-defined function rather simple.

With respect to an SD evaluation of T-CUBE, a double global constraint is to (i) reasonably minimize and balance the load of each node, while (ii) reasonably minimizing also the messaging between the nodes. The calculus should possibly satisfy further the following constraints:

- A node should have a reasonably small and about the same per node number of θ-CSs to find.
- Every θ-CS should be calculated at only one node. If some duplicates are unavoidable, they should be removed efficiently (processing and messaging cost).
- A node should calculate "its" θ-CSs and related aggregations without communicating with other nodes.

Fig. 4 shows our gross architecture, aiming at these constraints. We consider the grid or P2P environment. The *dispatcher* node coordinates the query evaluation. It calculates any restrictions, and distributes the calculus of θ-CSs to the application/client nodes (peers). The application component performs the actual calculus. The client component is as an SDDS client. It stores the θ-CSs produced at the node in a file or a relational table, dynamically distributing itself at SDDS servers. The file can be an LH* file, [LNS96], especially kept in the distributed RAM for orders of magnitude faster access than to disks. The table, termed *SD-table* is a relational table able to dynamically partition itself under inserts, transparently for an application, as under SD-SQL Server, [LRS02]. Each θ-CS has a key, unique regardless of, perhaps, its duplicate production by different applications. The SDDS server access is key based. Duplicated θ-CSs can thus be disregarded. Every application reports to the dispatcher once it terminates. The dispatcher performs then any post-processing or returns the control to the user.

To design the algorithms, one may start either from the database heritage or from the formal concept analysis heritage. The former approach implies the reuse of GROUP BY and CUBE algorithms, of some of a general relational query evaluation, and of some of those for an SDDS. The was a great body of work in these domains, [GUW02], [G&al97], [L05]. The latter direction implies the algorithms already proposed for a binary CS, [GWF99], and, more specifically, those proposed for a generalized CS, [DE97], [ELL97]. We now examine both directions.

4.2 *Database Heritage*

We target a general algorithm, applying to every θ. As for joins, we thus consider that the grouping calculus basically uses some nested-loop approach. Algorithms valid for some θ only, θ = '=' especially, are not in our scope here. Notice however that the major DBMSs have already the CUBE operator. Hence, the implementation of '='-CS calculus is there especially simple. Only the implementation of T-GROUP and of LIST functions remains. Note that, for Oracle, only T-GROUP may remain a concern, as limited editions of LIST were already proposed by some users as external functions, [L03].

The nested loop approach means that the grouping calculus (i) visits sequentially all the tuples concerned, and (ii) for each tuple, it examines all the possible subsets of its attribute

values. For each choice, termed the current *seed*, it visits other tuples in *T*, perhaps all, to determine the existing groups.

For instance, if we start with student 1, the successive seeds could be (12,_ ,_,_,_,_,_,_}), (_,16 ,_,_,_,_,_,_}), (12,16 ,_,_,_,_,_,_}), (_,_ ,6 ,_,_,_,_,_}), (12,_,6 ,_,_,_,_,_})… The 1st seed means the grouping calculus over $a = 12$. Next seed leads to the grouping over $a = 12$ and $b = 16$ etc. Different seeds may produce different θ-CSs. They may alternatively produce a duplicate. The latter case results for instance from (12,_ ,_,_,_,_,_,_}) and of (_,_ ,_,_,_,13,_,_).

Our SD calculus replicates *T* on some $N > 1$ nodes, numbered 0,1…*N*-1. Then, the calculus at each node computes θ-CSs only for some but not all seeds. The distribution should be uniform. *N* should be such that each node has neither too many nor too few θ-CSs to compute, with respect to some optimal load per node and number of messages to spread *T*. One may foresee the alternative of a centralized *distribution* calculus and of a distributed one. The former choice means that the dispatcher defines *N* upfront, given the estimate of the number of seeds to process and perhaps of θ-CSs wished to calculate per node. The latter approach means that application nodes determine *N* by themselves. Notice that the storage of θ-CSs is distributed in both cases, as they enter an SDDS.

One approach to a centralized distribution is to consider each seed as a large integer *s*, formed from the bit values of the concatenated attributes. Then, one may use same hash function *h* at every application node, mapping *s* on some number $a = 0,1…N-1$, where *N* is a parameter of *h*. Node *a* calculates then the θ-CS only for *s* such that $h(s) = a$. The dispatcher may choose *N* so to divide enough the interval [0, max(*s*)], where max(*s*) is the actual or a maximal possible *s*, e.g., with all the bits equal to 1. Once the dispatcher chooses *N*, it sends it along with *T* to all the nodes involved. Every node starts the calculus, and sends each θ-CS computed, with its key and all the additional aggregated values, into the SDDS, through its client component. The key can be simply the value of θ-CS also seen as integer.

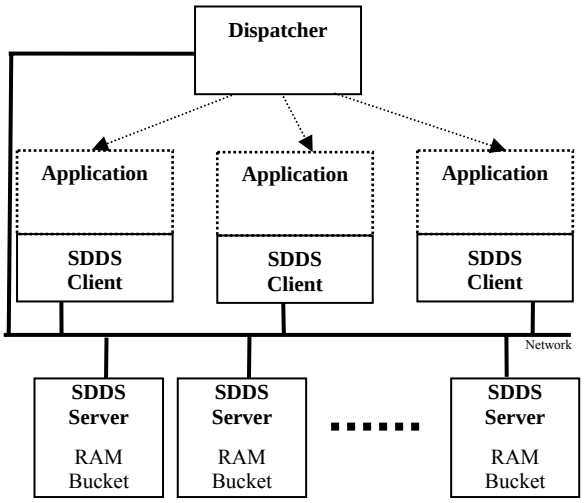

Fig. 4 SD-Architecture for T-CUBE calculus

The strategy for *N* choice may also take to the account the duplicated seeds. If there are many of those, the basic strategy above may end up largely sub-optimal. To define *N* on the basis of the actual seeds only, the dispatcher may first constitute a file of the non-duplicated actual seeds only. Any hash technique, e.g., for hash-joins will do. The technique minimizes

the number of seeds to visit. The resulting file can however end up perhaps very large, hence cumbersome to manage (it can be then itself made an SDDS file). The database heritage points then to the alternatively techniques that only estimate the number of the actual seeds from *T*. Several methods could serve with different trade-offs.

A distributed calculus of *N* can be as follows. We aim at the distribution of the actual seeds only that could above require a large file. Each application node has some *calculus capacity b* measured in the maximal expected number of seeds to process. It also disposes, instead of a unique hash function *h* above, of a family of hash functions h_i ; $i = 0,1\ldots$ that are the LH*-hash functions. The dispatcher starts by sending *T* to node 0. The node examines possible seeds and retains those that match its address 0, according to h_0. This function in fact hashes every seed to 0. If the node retains *b* seeds, it *splits*. This means it applies to the set of stored seeds h_1 function. The well-known result is that about half of the seeds will be remain hashed to 0, the other half being hashed to $0 + 2^{i-1}$ that is thus node 1. More generally function h_i applied at node *m* sends the seeds to node $m + 2^{i-1}$. Node 0 sends then these seeds to node 1, along with *T*, the query and the first seed s_1 it did not process yet.

Node 0 continues then the seed generation till it reaches *b* again. It then splits again, using h_2, sending as the result about *b*/2 seeds to node 2. Next split will send the seeds to node 4 etc. In the meantime, node 1 does similarly, using h_1 and starting with s_1. Its first split will use h_2 and send the seeds from some s_2 for evaluation to node 3. This node will start the similar processing starting with h_2 and s_2. The overall result is the distribution on about most adequate number *N* of nodes, given *b*.

The termination protocol towards the dispatcher with this scheme may be that each node ends up the calculus by sending the acknowledgement to the dispatcher with *i* of it final h_i. The dispatcher may then easily find out what nodes have been finally involved in the calculation and whether they have all terminated. Notice that we implicitly consider here that both the dispatcher and the application nodes have the IP addresses of the peers involved, with the same correspondence to the logical addresses $0,1\ldots N\text{-}1$ of the application nodes.

Once the above distribution phase terminated the local computation can take to the account techniques already known for CUBE, and the processing various relational queries with aggregates and θ-joins more generally. Alternatively, for a query using '='-CUBE only, one may reuse even a different distribution strategy inherited from the popular database techniques specific to equi-joins and CUBE. For instance, one may simply reuse the strategy of distributed hash joins at each attribute, intersecting then the tuple lists at different attributes, produced for join attribute values. All these aspects of T-CUBE processing remain to be analysed in depth.

4.3 *Concept Analysis Heritage*

Several algorithms for CS and GL calculus for binary attributes are known, especially the Ganter's classic, [G84]. As we mentioned, the universe of discourse of these algorithms makes them unfit for our database mining purpose. Most calculate a GL that is the issue we do not address here (yet). Whether some algorithms can be generalized to our goal remains an open issue.

The concept analysis community has of course noticed the existence of multi-valued attributes. However the main direction to formally deal with was apparently simply the conceptual mapping of such attributes into the binary ones, called *scaling* [G01]. This approach also seems unfit for our purpose. An alternative and exclusively formal approach to multi-valued GLs is suggested in [G01]. It starts with a generic concept of *description*. This one assigns to an object set-valued attribute values, in the way somehow similar to the way proposed by the symbolic objects approach, [D0]. It then orders the objects into a GL

according to the inclusion of the descriptions (called *precision* of the descriptions). Our "universe of discourse", of single-valued attributes typical for the relational databases, corresponds entirely to the most restrictive case in [G01]. Our definition of GCs on the basis of the values of the θ-operator is "orthogonal" to the formalism of [G01]. That formalism will be however perhaps of more use for the database mining one day, if the set-oriented attributes become more popular with the databases. Besides, its other conceptually interesting point is the attempt to formally cope with set-valued nulls, called *empty cells* in [G01]. Likewise, the proposal copes with the formalism for sub-GLs, over some descriptions up to the selected precision. This approach will perhaps be also of interest for the databases one day.

Another generalization attempt, we already somehow discussed, apparently unknown to [G01], is that in [DE97], together with the subsequent algorithms in [ELL97]. In our vocabulary, these proposals all limited to the '≥'-CSs and, for some algorithms, to the calculus of '≥'-GLs. Next, the proposed algorithms are not designed for the groupings with restrictions. They generate the full sets of '≥'-CSs. As the analysis shows, even for quite a few tuples and attributes, e.g., in dozens, these are typically very large, e.g., in billions of CSs.

Among these algorithms, one termed ELL, generating the '≥'-CSs only, i.e., without their GL, appears the fastest. This property holds, by the way, for the ≥-CSs with the binary attributes only as well, [BL05]. ELL has two formulations, termed respectively *iterative* and *recursive*. A scalable distributed version of the iterative ELL was proposed in [BL03]. The idea was to partition the set of object or of attributes over the application sites. Unfortunately, the calculus needed then the inter-application node communication, close to gossiping for every CS generated. The messaging cost of the algorithm was consequently prohibitive, already for collections of dozens of objects.

In the wake of the study to overcome this limitation, it appeared that the recursive ELL can possibly constitute a basis for the scalable distributed computation[3]. It indeed recursively divides *T* so that the computation of CSs over successive subsets of objects partitions more and more the set of all the CSs. During the division, the dispatcher calculates some CSs by itself. It then sends out subsets of *T* smaller than some arbitrary threshold, expecting to end up with small enough sets of CSs to generate per application node. The number of the subsets determines *N*. The local calculus of CSs uses the iterative ELL. The CSs are collected in an SDDS, as in Fig. 4. The process generates a partition of all the CSs, hence the algorithm minimizes the number of generated CSs and is thus optimal in the sense.

The SD application of ELL (SD-ELL) appears a potential basis for '≥'-CSs calculus, with respect to both the distribution and the calculus schemes. Provided however that one can re-engineer it for the restrictions. Otherwise, the sheer size of the generated collection could be too large in practice, as we already mentioned. The algorithm has also be studied more in depth, as well as compared more to the "database inherited" methods. In particular, since the first experiments[4] show the differences among the numbers of '≥'-CSs generated per node, for the same size of the input set, from simple to double. They also show that a node getting less objects than another one may still generate more '≥'-CSs. Hence, the decision when to stop the recursive division in practice is not clear yet. Finally, whether SD-ELL can be generalized to other θ values is an open question.

[3] Observation by G. Levy, after the common brainstorming including F. Baklouti.
[4] By F. Baklouti.

5 Conclusion

Galois connections analysis is the heart of the formal concept analysis. The CUBE operator is popular with the database mining. Both domains exhibit an interesting relationship. The conceptually simple T-CUBE operator and new aggregate functions that result from, materialize this relationship for the SQL users. New database mining capabilities result from. In turn, the popularity of CUBE operator, coupled with the LIST and T-GROUP aggregate functions, as well as of θ-joins, shows that θ-CSs may be useful for the formal concept analysis. New perspectives open for the Galois connection analysis in larger collections.

The P2P and grid environment appears in every case the most appropriate support for the underlying calculus. Known algorithms for the CUBE processing as well as some of those known to the concept analysis may apply to T-CUBE for specific θ values, θ = '=' or '≥' especially.

Future work should concern at first deeper analysis of the algorithms described. One should also proceed with the implementation of LIST and of T-GROUP on existing DBMSs, especially those we have mentioned. Next, various open problems we have mentioned should be addressed. Furthermore, one should investigate the processing of the (multi-valued) θ-GLs. The database techniques for the transitive closure queries seem a promising starting point towards this goal. Notice finally, as we have shown, that the operators T-GROUP BY, T-ROLL UP, T-GROUPING SETS make sense as well, hence could be studied on their own.

Acknowledgments

This research was partly supported by a grant from Microsoft Research.

References

[BL03] Baklouti. F, Lévy. G. Parallel algorithms for general Galois lattices building. Proc. WDAS 2003.

[BL05] Baklouti. F, Lévy. G. A fast and general algorithm for Galois lattices building. Submited to the Journal of Symbolic Data Analysis. April 2005, 1st revision.

[B67] Birkhoff. G. Lattice Theory. American Mathematical Society, Providence, RI, 3rd edition. 1967.

[B86] Bordat. J. Calcul Pratique du Treillis de Galois d'une Correspondance. Mathématique, Informatique et Sciences Humaines 24(94), 31. 1986.

[D0] Diday, E. Knowledge discovery from symbolic data and the SODAS software. The 4th Europ. Conf. on Principles and Practice of Knowledge Discovery in Databases, PPKDD-2000. Springer (publ.).

[DE97] Diday. E, Emilion. R. Treillis de Galois maximaux et capacités de Choquet. Cahier de Recherche de l'Acadèmie des Sciences. Paris, t.325, Série I, p.261-266, 1997.

[ELL97] Emilion. R., Lambert. G, Lévy. G, Algorithmes pour les treillis de Galois, Indo-French Workshop., University Paris IX-Dauphine. 1997.

[G84] Ganter. B. Two basic algorithms in concept analysis. Preprint 831, Technische Hochschule Darmstadt 1984.

[G&al97] Gray, J & al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Data Mining and Knowledge Discovery J., 1 , 1, 1997, 29 – 53.

[GM95] Godin. R., Mineau. G. & al. Méthodes de classification conceptuelle bases sur les treillis de Galois et application. Revue d'intelligence artificielle pp 105-137. 1995.

[GUW02] Garcia-Molina, H., Ullman, J., Widom, J. Database Systems: The Complete Book. Prentice Hall, 2002.

[GWF99] Ganter, B. Wille, R., Franzke, C. Formal Concept Analysis: Mathematical Foundations. Springer-Verlag New York, 1999, 294.

[G01] Gugisch, R. Many-valued Context Analysis using Descriptions. H. Delugach and G. Stumme (Eds.): ICCS 2001, LNAI 2120, 157-168, 2001, Springer-Verlag.

[H05] Hainaut, J-L. Recovering ISA Structures. Introduction to Database Reverse Engineering. Chapter 14. (book in preparation), 2005.

[LNS96] Litwin, W., Neimat, M.-A., Schneider, D. LH* : A Scalable Distributed Data Structure. ACM Transactions on Database Systems (ACM-TODS), Dec. 1996.

[L03] Litwin, W. Explicit and Implicit List Aggregate Function for Relational Databases. IASTED Intl. Conf. On Databases and Applications, 2003.

[L05] Scalable Distributed Data Structures. ACM Thirteenth Conf. On Information and Knowledge Management (CIKM-2004), Washington DC. Tutorial.

[LRS02] Litwin, W., Risch, T., & Schwarz, Th. An Architecture for a Scalable Distributed DBS: Application to SQL Server 2000. 2nd Intl. VLDB Workshop on Cooperative Internet Computing (VLDB-CIC 2002), 2002, Hong Kong.