Optimizing inductive queries in frequent itemsets mining

Marco Botta, Roberto Esposito, Arianna Gallo, and Rosa Meo

May 4, 2004

1 Minerule optimizer: detailed description

Here we describe in large details how the optimization engine of the minerule system works. We give details about the features currently implemented in the minerule system as well as about features we have not implemented yet, but for which we already investigated the feasibility of the approach.

1.1 Task definition and proposed approaches

Let $Q = \{Q_1, \ldots, Q_n\}$ be a set of past queries and let $R = \{R_1, \ldots, R_n\}$ be their results. Moreover, let Q_0 be a query newly submitted to the system and let R_0 be its result. The task of optimizing the extraction of R_0 using the knowledge provided by Q and R have been faced following two distinct approaches.

In the first approach we search for a query $Q_i \in Q$, such that $R_0 \subseteq R_i$ (in such a situation we say that Q_i dominates Q_0) and try to exploit this result in order to simplify the mining of R_0. We notice that in such a situation the system must face two problems:

1. how to detect the relationship and
2. how to exploit R_i in order to build, efficiently, R_0.

Problem (1) is an optimization problem and will be faced in the present section, problem (2) (as we will see) is a mining problem that requires ad-hoc mining algorithms. We call those algorithms “incremental” and study them in greater details in Section 2.3.

The second approach, consists of combining a number of previous results using set intersections and set unions operations in such a way to build R_0 without performing any further mining operations. More formally we search for a set of queries $Q' = \{Q_{i_1}, \ldots, Q_{i_m}\}$ such that $Q' \subseteq Q$ and $R_0 = R_{i_1}\theta_1 \ldots \theta_{m-1}R_{i_m}$ where each θ_j is either the symbol $\cap$ or the symbol $\cup$. This approach does not need further mining operations since the result set can be built entirely out of previous results. As we showed in [6], it is not always possible to rewrite a query in terms of previous results even when the new predicate is explicitely written

in terms of previous predicates. However, we are able to check for sufficient conditions that guarantee the correctness of the result.

The two approaches face two different facets of the optimization problem and, from a theoretical point of view, can be combined in order to better optimize the query. For instance, if we could find a set of queries Q' such that R_0 is strictly included in the combination of their results, we could employ the techniques and the algorithms developed for the first approach in order to "complete" the optimization.

For the sake of exemplification, let us consider the following examples.

Example 1.

In the current query, the user wants to mine all the rules such that the items in the body of the rule costs more than 10.000 euros, while the items in the head of the rule costs more than 12.000 euros. Then, the mining constraints of query Q_0 specifies: $\text{BODY.price} > 10000 \wedge \text{HEAD.price} > 12000$.

Let us now assume that there exist a candidate rewriting [6] Q_i of Q_0 for which the mining predicate specifies that the rules of interest are those having $\text{BODY.price} > 12000 \wedge \text{HEAD.price} > 14000$.

In this situation most of the information needed by the user is already present in a previous result. The goal of the optimizer and of the incremental algorithms, is to exploit this information in order to provide a faster feedback to the user.

Example 2.

The mining predicate in Q_0 is $\text{price} > 1000 \wedge \text{color} = \text{blue}$. Let us assume that there exists two queries $Q_1$and Q_2 in Q such that the mining predicate in Q_1 is $\text{price} > 1000$ and the mining predicate in Q_2 is $\text{color} = \text{blue}$. In this situation the optimizer should:

- recognize that $Q_0 = Q_1 \wedge Q_2$
- verify that the conditions to build the result query out of previous ones are met [6]
- build the result set R_0 as $R_1 \cap R_2$

In the examples, we provided very clean situations in which to apply the methodologies. This is clearly an oversimplifications of the problems involved. Anyway, even in their simplicity, the examples already emphasize that an important subtask must be solved in both the approaches: we must be able to check two queries for equivalence in an efficient way. Unfortunately it is well known that the problem is $\mathcal{NP}$ complete (in particular the formula satisfability problem, which is equivalent to the equivalence problem, is a canonical example of $\mathcal{NP}$ complete problem) and therefore an asymptotic efficient algorithm is not known. However, as we will show, the size of the queries involved are so small that the problem can be efficiently tackled in most cases.

At the present time, the minerule system fully implements the query equivalence checking system that we present below. Preliminary versions of the incremental algorithms have been also implemented and described in details in

Section 2.3. At the present time the optimizer is not yet able to provide the incremental algorithms with the information they need to work properly, then, in order to exploit them the user still needs to identify which query needs to combine or whether an old query dominates the current one. Nonetheless a number of optimization are already done by the system, part of them are throughtfully explained in example reported in Section 1.2.5.

1.2 The query equivalence checking system

In order to carry on any kind of optimization, the optimizer must be able to check whether two boolean expressions are equivalent ones. The check is performed in two steps, in the first the two formulae are rewritten in an alternative form in order to maximize the chances of finding an equivalence, in the second step they are compared using a compact representation of their thruth tables.

1.2.1 Query Normalization

Let us introduce the need of a query normalization step by means of an example. Let us consider a normalized relational database, and let us assume that the database schema implies that the columns item and price of the mining table are two candidate keys for that table. Exploiting this information it is easily seen that, provided that the item bread costs 1 euro, the constraints item = bread and price = 1 are equivalent. The normalization step, makes this kind of domain knowledge exploitation systematic. More precisely, it uses the information provided by the database schema and some additional information in order to rewrite the two queries so that their equivalence become apparent, in particular, in our example we would rewrite the second constraint as item = bread.

More formally, let T be the mining table; we assume that someone provide us with a set of *substitution rules* each one in the following form:

$$A \rightarrow a, \zeta$$

where A is a set of columns of table T, a is a single column of the same table, and $\zeta \in \{\updownarrow, \Box\}$. The meaining of each rule is the following: "A and a are equivalent keys for table T; among them a is the reference attribute, the ordering relation between the two keys is ζ". More precisely, if $\zeta = \updownarrow$ then the rule specifies that, for each attribute in A, increasing values of the attribute are associated to either increasing values of a or to decreasing values of a; if $\zeta = \Box$ then neither the two relations can be stated. We refer to the left part of the rule as the rule head, to the attribute a as the "reference" attribute and to ζ as the ordering relation.

Without loss of generality we consider here only formulae in disjunctive normal form, that is:

$$P = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} C_{ij}$$

where $C_{ij} = b\theta v$, b is a column of T, $\theta \in \{<, \leq, >, \geq, =, \neq\}$, and v is either a column of T or a value in the domain of b. In the following, we refer to C_i as a short for $\bigwedge_{j=1}^{m_i} C_{ij}$ and call each C_{ij} an atomic predicate. Let us consider a substitution rule $A \rightarrow a, \zeta$, an atomic predicate $C_{ij} = b\theta v$ is a *canonization candidate* if $b \in A$ and v is *not* a column identifier. In such a situation we call a the canonization attribute of C_{ij} w.r.t. $A \rightarrow a, \zeta$. We say that θ is consistent with ζ whenever $\theta \in \{=, \neq\}$ or $\zeta = \updownarrow$.

The normalization of P is a two step process. The steps are:

canonization

begin
  foreach $C_i \in P$ **do**
    let S_A be the set of all canonization attributes of the constraints C_i w.r.t. $A \rightarrow a, \zeta$;
    if $S_A = A$ *and the comparisons in the canonization candidates are consistent with* ζ **then**
      set $C_i^{\mathsf{rules}} = C_i^{\mathsf{rules}} \cup A \rightarrow a, \zeta$;
  to each C_i apply the found substitution C_i^{rules} (see Section 1.2.2);
end

minimization remove from P every C_i which form a contraddiction, remove P if any C_i forms a tautology, and remove any redundant C_{ij} (see Section'1.2.3).

1.2.2 Canonization

In brief, the canonization step is needed to augment the chances that equivalences are found. It works by substituting (when it is possible) queries made on generic attributes with ones made on reference keys. Basically, the algorithm works in two steps, in the first it finds all possible substitutions and in the second it applies them. This two step process is needed since, as we shall see, the substitution is made applying all the applicable rules at once.

After the canonization step, it is often the case that some redundance, contraddiction or tautology can be discovered in the mining query. The minimization step is needed in order to clean up the predicate before the rest of the equivalence checking system tries to find equivalences. It also exposes some equivalences that would not be noticed otherwise.

Let us now briefly review how the substitution works. The substitution algorithm takes in input the set C_i^{rules} containing all the rules that can be applied to the conjunct C_i, the result of the algorithm is to replace C_i with a new conjunct of atomic predicates which is semantically equivalent to the former, but is built using different columns of T. The semantical equivalence is guaranteed by the functional equivalence of the attributes in the head of the rule and the attribute in the tail.

Let us consider a rule $A \rightarrow a, \zeta \in C_i^{\text{rules}}$ and let $C_{iA} = a_1\theta_1 v_1 \wedge \ldots \wedge a_k\theta_k v_k$ be the subexpression of C_i formed by all the atomic predicates in C_i for which the left hand side belongs to A. Moreover let us consider an atomic predicate $a_l\theta_l v_l$ in C_{iA} (here and in the following $l \in \{1, \ldots, k\}$). Since θ_l is consistent with ζ, there exists a range $[m_l, M_l]$ of values belonging to the domain of a such that if $a_l\theta_l v_l$ is satisfied implies $a \in [m_l, M_l]$. As it is showed in Figure 1, this implies that C_{iA} is true iff a belongs to the range $[\max_l(m_l), \min(M_l)]$. Then we aim at substituting each C_{iA} with the conjunctive formula $a \geq \max_l(m_l) \wedge \min_l(M_l)$. Before doing that we have still to answer to two questions: how we perform all the substitution at once as claimed earlier, and how we get the $\max_l(m_l)$ and $\min_l(M_l)$ values? Both questions are easily answered. In order to make all the substitution at once we first eliminate all the canonization candidates from C_i, then we add to the resulting conjunction all the conjunctive formulae we built on the attribute a as explained above. The $\max_l(m_l)$ and $\min_l(M_l)$ values are easily obtained from the DBMS by means of a SQL query. Both the running time of the query and the overhead needed for the communication with the DBMS can be neglected. In fact, provided that suitable indexes are present in the database, the query needed to find the maximum (the minimum) of the result of a conjunctive predicate is one of the operations a DBMS is very good at. Moreover, since the result set consists of only two values, the overhead due to the interaction with the DBMS is very small as well.

As a part of the canonization step, we perform a further substitution that we have not mentioned earlier for the sake of simplicity. This last canonization step deals with the following problem. Let us consider two atomic predicates, $a > 1000$ and $a > 1001$, the question is: are they equivalent? Again we have to fill the gap between the syntactical equivalence check we want to make and the semantic involved in the query. Obviously the predicates are not syntactically equivalent, but there is the chance that every time the first atomic predicate is true, the same holds for the second because there is no tuple in the database for which a is exactly 1001. In order to overcome this problem, we modify the predicate given by the user. In this case however, the operation to be performed is quite simple: every time an atomic predicat in the form $a > v$ or $a < v$ is given to the system, we query the database in order to find the first value present in the database such that the predicate holds. We then relax the predicate (i.e., we change $<$ to $\leq$ and $>$to $\geq$) and substitute the found value to v. In case there is no value in the database that satisfy the query, then the whole conjunct is never satisfied. As a result, the conjunct is removed from the query (or it is substituted with the false predicate, in case no more conjuncts are present in the query).

1.2.3 Minimization

Here, we describe the algorithm the optimizer uses in order to remove redundant predicates from the mining condition. The algorithm analyses the conjunctions in predicate P one at a time applying to them the algorithm reported in Figure 1. The algorithm analyses the atomic predicates from left to right comparing each

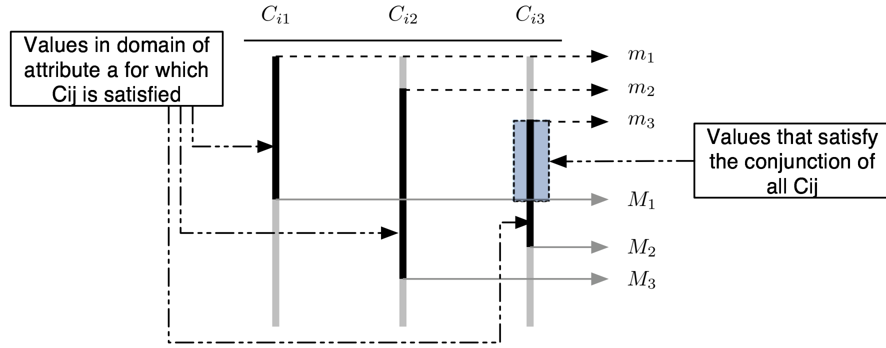

Figure 1: An example, involving three atomic predicates, of how the correct range for the values of attribute a is determined.

one with those on its right. The comparison is made by means of the function getRelation, which basically access a lookup table (reported in Table 1) that specifies the relationship type between the two atomic predicates as a function of the operators that appear inside them. Once the relationship has been estabilished, the algorithm perform the conresponding operations. The meaning of the symbol used in both the lookup table and in the algorithm is the following:

$\rightarrow$ C_{il} implies C_{ik}, then C_{ik} is redundant and we should eliminated it

$\leftarrow$ C_{ik} implies C_{il}, then C_{il} is redundant and we should eliminate it

$\leftrightarrow$ C_{ik} and C_{il} are equivalent, we should eliminate one of the twos

! C_{ik} and C_{il} forms a contraddiction, we should eliminate the whole C_i

$*$ no relationship can be estabilished between the two predicates

$=$ the two predicates implies an equality operator,e.g.,$C_{il} \equiv$ a ≤ 5 and $C_{ik} \equiv a \geq 5$, we should substitute one of the two relational operators with $=$ and discard the other predicate

$<$ similar to $=$, but the operator implied is $<$

$>$ similar to $=$, but the operator implied is $>$

More in details, the getRelation function checks whether C_{ik} and C_{il} are defined over the same attribute. If this is false the function returns $*$, otherwise it uses both the information about the value with which the attribute is compared and the informations provided by Table 1 in order to return a correct answer. In the table, the column “Xop” reports the operator present in C_{il}, column “Yop” the operator in C_{ik} and column “ord” the relative ordering of the two database columns. The meaning of the symbols in column “ord” is the following:

1 the two columns are sorted in the same order

-1 the two columns are sorted in reverse order (i.e., decreasing values of the table column present in C_{ik} are associated to increasing values of the table column present in C_{il})

0 neither of the two sorting relations can be stated

We notice that the definitions above make sense only because the table columns are functionally related by hypothesis.

```
Data : A conjunctinve formula C_i = ⋀_j C_ij
for l ∈ {1..m_j} do
    for k ∈ {l + 1..m_j} do
        switch getRelation(C_il, C_ik) do
            case ∗
                skip;
            case ←
                remove C_ik;
            case →
                swap C_ik and C_il;
                remove C_ik;
            case ↔
                remove C_ik;
            case !
                remove C_i;
            case =
                substitute the operator in C_il, with =;
                discard C_ik;
            case <
                substitute the operator in C_il, with <;
                discard C_ik;
            case >
                substitute the operator in C_il, with >;
                discard C_ik;
        endsw
    end
end
```

Algorithm 1: Minimization algorithm

		Xop					
ord	Yop	<	≤	=	≥	>	≠
−		→	→	→	*	*	*
0	<	↔	←	!	!	!	←
+		←	←	!	!	!	←
−		→	→	→	*	*	*
0	≤	→	↔	→	=	!	<
+		←	←	!	!	!	←
−		!	!	!	←	←	←
0	=	!	←	↔	←	!	!
+		←	←	!	!	!	←
-		!	!	!	←	←	←
0	≥	!	=	→	↔	→	>
+		*	*	→	→	→	*
-		!	!	!	←	←	←
0	>	!	!	!	←	↔	←
+		*	*	→	→	→	*
-		→	→	→	*	*	*
0	≠	→	<	!	>	→	↔
+		*	*	→	→	→	*

Table 1: Lookup table for the minimization algorithm

1.2.4 The equivalence checking algorithm

The equivalence checking algorithm aims at veryfing the logical equivalence of the predicates involved in the queries P_1 and P_2. Since the normalization step already put the atomic predicates in a form which is easily checked for identity, the remaining step needs only to verify whether two syntactically different predicates implies each other, as it would be the case if, for instance, $P_1 = ([\mathrm{a} < 3] \vee [\mathrm{b} < 5]) \wedge [\mathrm{c} = 4]$ and $P_2 = ([a < 3] \wedge [c = 4]) \vee ([b < 5] \wedge [c = 4])$.

The question arise about whether the operation is really necessary and if it would not be better to save implementation and execution time by requiring the user to be "self-consistent" in the queries she writes. The answer is that we cannot pose such a burden over the user for at least two good reasons. The first one, is that the queries submitted to the system may remain inside the system catalogue for a very long time before they become useful again. Clearly in such a situation the "self-consistency" requirement would be too much demanding for the user. If we could not find any other option, it would be better to avoid the normalization step altogether and ask explicitly the user which of the previous queries may be used to mine the new results. The second reason is that the user may not have all the informations needed to satisfy the requirement. In fact, after the normalization step, the query may look somewhat different from the one intended by the user (even though it remains semantically equivalent), and redundancies may become apparent only in that occasion.

Given this, a careful implementation must be carried on to solve the problem. The problem is, in fact, $\mathcal{NP}$ complete. The only hope to build an efficient equivalence checker resides in the fact that usually the size of the queries is modest. Moreover, since the check for equivalence is a really common operation (and their number per session increases as new queries are put into the catalogue), we must double our efforts to obtain an efficient implementation.

Before going to describe the equivalence checking system, we expose here an important detail of the system: it is not true that it suffices to find two "different" atomic predicates in order to conclude that the queries are different. In fact, we checks for equivalence of two predicates even when they are built on a (non totally) disjunct set of atomic predicates. The reason is that we must cope with situations similar to the one in which $P_1 = \mathrm{a} < 4 \vee T$, and $P_2 = \mathrm{b} < 7 \vee T$[1], i.e., we must consider situations in which the predicates which differ in the two predicates does not really contribute to the truth value of the formula (they are then unuseful, but we cannot check it without making the whole equivalence check).

We do not give here the details of the implementation, but expose instead the choices we made to keep the execution time of the algorithm as low as possible.

The first, important choice is to not materialize the whole truth table of the predicates. In fact, we do not need it and it grows exponentially with the size of the query. Instead we build a compact representation of its "output" by means of as a binary string mapped on a vector of 32 bit integers. This choice allows us to check for equivalence by performing $\max(1, 2^{n-5})$ comparisons between

[1]Here T stands for the "True" predicate

integers. Here n is the size of the current two queries in terms of distinct atomic predicates. We notice that for common query sizes this amount to perform only 1 comparison and that the comparison for equality between two integers is usually made in a single clock cycle (we are disregarding here the time needed to fetch the two integers). The binary string representation can be made because we fix ahead of time the ordering of the atomic predicates, then the position of each bit identify uniquely the exact truth values of the variables involved in the checking.

The other important implementation detail concerns how we generate all the possible truth values of the atomic predicates. Again this can be done very efficiently since we decide ahead of time the ordering of the predicates. In fact, we use a single 32 bit unsigned integer to iterate through all the possible truth values. We initialize it to 0 and increase it until we reach $2^n - 1$. At each step we use the bit configuration of the integer to decide which atomic predicate is true and which is false. Again, the machine operations needed to perform the algorithms are very fast ones since they involve only bit operations between integers. As a side effect of this choices, we cannot handle more than 31distinct atomic predicates. We notice, however that this does not limit the optimizer. In fact, such a number of predicates is not manageble by the algorithms anyway due to the huge amount of work needed to check the equivalence.

Those two "tricks" allows us to keep the execution time very low. The experiments show that we could perform the equivalence checking when $n \simeq 18$ in about one tenth of a second. When $n = 21$ the equivalence checker takes one second to complete. We conclude that when n increases beyond this limit, the optimizer should not be used since it would possibly waste too much user time.

1.2.5 A complete example

Let us consider the following setting:

- The database table under investigation is T (reported in Table 2), the columns of the table are $\mathrm{tr}, \mathrm{a}_0, \mathrm{a}_1, \mathrm{a}, \mathrm{b}, \mathrm{c}$ all of them are of type integer. In this example we assume that items are modeled by values in the database column a.
- The system catalogue tells us that the following substitution rule is valid: $a_0, a_1 \to a, \updownarrow$
- Sometime in the past the user issued a query which specified to mine all frequent itemsets, satisfying the constraint $a < 5 \wedge b > 7 \vee c = 3$.
- The actual query specifies to mine all frequent itemsets for which it holds $(a_0 > 90 \wedge a_1 < 19 \wedge a \leq 4 \vee c = 3) \wedge (b > 10 \vee c = 3)$

Needless to say, the example has been handcrafted to show a case in which two very different mining queries are found to be equivalent by the optimizer.

Before explaining how the current query is handled by the system it is necessary to notice that, the queries are stored in the database catalogue in their

tr	a_0	a_1	a	b	c
1	100	1	1	5	3
1	110	7	3	11	12
1	120	19	5	15	50
2	100	2	2	7	10
2	110	11	4	21	30
2	120	21	6	21	60
3	100	2	2	7	10
3	110	11	4	21	30
4	110	7	3	11	12
4	120	19	5	15	50

Table 2: Source table for the example described in Section 1.2.5.

normalized form. As it will become apparent soon, the normalized form of the previous query is $P^* = a \geq 1 \wedge a \leq 4 \wedge b \geq 11 \vee c = 3$, we do not give here the details about how this has been accomplished since the same operations will be repeated soon for the current query.

First we need to put the predicate in disjunctive normal form, let therefore

$$\begin{aligned} P^0 \quad = \quad & [a_0 > 90 \wedge a_1 < 19 \wedge a \leq 4 \wedge b > 10] \vee \\ & [a_0 > 90 \wedge a_1 < 19 \wedge a \leq 4 \wedge c = 3] \vee \\ & [b > 10 \wedge c = 3] \vee \\ & [c = 3] \end{aligned}$$

be the result of this first step. Then for each conjunct, we need to apply all possible substitution rules. The first and the second conjuncts, are suitable for the only rule we are aware of. The substitution is made by asking to the database which are the minimal and maximal values of column "a" associated to tuples which satisfy the constraint $a_0 > 90 \wedge a_1 < 19$. As it is easy to check, such values are 1 and 4 respectively. Moreover , in the rest of the query, the operators $<$ and $>$ are substituted with $\leq$ and $\geq$ respectively and the value they are compared with are substitued to the maximal (respectively minimal) values in the database which satisfy the predicates. At the end of those operations we obtain the following canonical predicate

$$\begin{aligned} P^1 \quad = \quad & [a \geq 1 \wedge a \leq 4 \wedge a \leq 4 \wedge b \geq 11] \vee \\ & [a \geq 1 \wedge a \leq 4 \wedge a \leq 4 \wedge c = 3] \vee \\ & [b \geq 11 \wedge c = 3] \vee \\ & [c = 3] \end{aligned}$$

The canonization step is now completed, we then apply the minimization step obtaining thus the predicate

$$
\begin{aligned}
P^2 \quad = \quad & [a \geq 1 \wedge a \leq 4 \wedge b \geq 11] \vee \\
& [a \geq 1 \wedge a \leq 4 \wedge c = 3] \vee \\
& [b \geq 11 \wedge c = 3] \vee \\
& [c = 3]
\end{aligned}
$$

The last step performed by the optimizer is to check the truth table of the two predicates P^* and P^2 in order to check for logical equivalence. In so doing the optimizer disregard the actual contents of the atomic predicates. Instead, it handles them as they where four different propositional variables only checking whether the connectives included in P^1 and P^* implies logical equivalence or not. In other words, it checks whether $P^* = A \wedge B \wedge C \vee D$ and $P^2 = A \wedge B \wedge C \vee A \wedge B \wedge D \vee C \wedge D \vee D$ are equivalent. As Table 3 makes evident, in fact they are.

A last notice to emphasize, once again, that despite we materialized the truth table for presentation purposes, the system does not. In fact, it only builds the two bitstring contained in the columns labelled $P^{\{*\}}$ and P^2 and then checks for their identity. Since the two strings are shorter than 32 bit, this amount to a single integer comparison.

A	B	C	D	P^*	P^2
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

Table 3: Truth tables for predicates P^* and P^2.

2 Incremental Algorithms for association rules extraction with constraints

2.1 Introduction

In [6] we have presented new classes of constraints, called *contex dependent constraints* (CDC) whose satisfaction for a given pattern depends on the context in the database in which the pattern occurs. In contrast, traditionally studied constraints [3, 4, 2, 7, 8] are satisfied for a given pattern either in all their occurrences in the database or in none of them. We call these constraints *item dependent constraint* (IDC). We present new algorithms to deal with context dependent constraints

2.2 Mining Index

In this Section we describe the intermediate data structures that are used by more general incremental algorithms in order to retrieve from the database only those data that are effectively needed. In traditional databases, an index is used by the system to make efficient access to a tuple searched by the value of some of its attributes. A similar need is present in the case of association rule mining. Given an itemset I, from which a rule r in R can be generated, we need to retrieve all the groups in which I is present. This because we want to evaluate the constraints of the new query *M'* on the rule r in order to compute its support. A mining index is designed as follows. In order to quickly and efficiently evaluate mining conditions, the MINE RULE system uses a set of so called mining indexes. These structures allow to avoid to read us the database. Since the itemset elements aren't stored in consecutive way, without mining index we would have much I/O to level of page. A mining index tells which groups in the source relation contain an item satisfying a mining condition. It is implemented as a B+-tree. A B+-tree is a data structure to store vast amounts of information. Typically, B+-trees are used to store amounts of data that will not fit in main system memory. To do this, secondary storage (usually disk) is used to store the leaf nodes of the tree. Only the internal nodes of the tree are stored in computer memory. In a B+-tree the leaf nodes are the only ones that actually store data items. All other nodes are called index nodes or i-nodes and simply store 'guide' values which allow us to traverse the tree structure from the root down and arrive at the leaf node containing the data item we seek. In the mining index used by MINE RULE, a data item is a pair of values $< GID, item >$, where GID is a group identifier, while nodes are organized according to the values of the mining attribute.

For instance, let us consider the mining condition BODY.price > 100, that says that all items in the body of a rule must have price greater than 100. A mining index is built in MINE RULE on the price attribute in order to quickly access groups and items that satisfy the mining condition. The mining condition BODY.price>100 can be evaluated by using the mining index: indeed, querying the index with this condition will provide all pairs $< GID, item >$ that satisfy

the condition without accessing the source table, ordered by group or item as necessary. Moreover, the result of such a query on the mining index is a set of pairs. Computing more complex conditions can be done by simply querying the corresponding mining indexes and then combining the resulting sets of pairs by simple set operations. In such a way, a usually small set of $< GID, item >$ pairs will be considered by the MINE RULE engine during the subsequent rule extraction phase.

The mining index can also be used to refine a previously computed query result, in order to answer a new related query. For instance, let us suppose the result of a query Q1, whose mining condition is BODY.price > 100 has already been computed and stored in the database. A new query Q2, whose mining condition is *BODY.price* > 200 is to be evaluated. By using the mining index on price the MINE RULE engine can find the items with 100 < *price* <= 200 and the groups in which they occur. Then, the result of query Q1 can be updated by decrementing the support of the rules and/or the confidence of the itemset that do not satisfy the constraints anymore. This issue will be discussed in detail in the following Section.

2.3 Incremental Algorithms

With the above observations, we propose two enhanced incremental algorithms. We call them *destructive algorithm* and *constructive algorithm*. Here, we consider only queries in which the mining conditions are composed by conjunctive predicates. For the sake of clarity, in order to be concrete, we make use of a practical example. Suppose that a MINE RULE query Q_1 with the mining conditions BODY.price<50 has been already executed. After some time, another query is submitted, Q_2, that is identical to Q_1 apart from a tighter mining condition (BODY.price<100). You can see that Q_1 dominates Q_2 because between the mining conditions of Q_1 and Q_2 there is an implication relationship. The intuition tells us that it is not necessary to read the database in order to derive result set of Q_2 from the one of Q_1. As we shall show, it suffices that someone (most likely the optimizer of the system) provides us with some additional information in order to be able of solving the problem. In particular, the algorithms described in this section will make use of few sets of pairs $< GID, item >$. The description of these sets, what they contain and how to build them is reviewed in the following section, the description of the algorithms will follow.

2.3.1 Set of pairs

Let $B_1 \wedge H_1$ be the mining condition of query Q_1, where B_1 and H_1 are the conjunctive constraints that apply to the body and the head of the rule, respectively. Using the same notation, let $B_2 \wedge H_2$ be the mining conditions of query Q_2. If Q_1 dominates Q_2, we can evaluate the result set R_2 of the query Q_2 from the result set R_1 of the query Q_2. In order to do that we provide the constructive and destructive incremental algorithms with four or two sets of pairs respectively. Each pair is in the form $< GID, item >$, where GID is a group

identifier and *item* is an item to be mined. In the following we will denote with S_i those sets with $i \in \{1 \dots 4\}$ or $i \in \{1 \dots 2\}$ depending on which algorithm is being introduced. In the most common case, these sets are the result of a query on the mining index.

If the Optimizer decides to call the Destructive Algorithm, it poses four queries to the mining index, otherwise only two are necessary. The query constraints that must be used to form each sets in both cases is reported in Table 4.

Set	Mining Index Constraint
Destructive algorithm	
S_1	$B_1 \wedge \neg B_2$
S_2	$H_1 \wedge \neg H_2$
S_3	B_1
S_4	S_4
Constructive algorithm	
S_1	B_2
S_2	H_2

Table 4: Mining index queries needed to form the input of the incremental algorithms.

In the following we will need to distinguish among the several different items and gids we read from the mining index result. To this aim we adopt the following notation:

$$S_i = \{< GID_1, item_{1,1} >, ..., < GID_1, item_{1,n} >, \\ < GID_2, item_{2,1} >, ..., < GID_2, item_{2,m} >, ..., \\ < GID_k, item_{k,1} >, ..., < GID_k, item_{k,w} >\}$$

where n, m, w are the number of items in the *gid lists* $1, 2, \dots, k$ respectively.

Example 3.

gid	item	price
1	A	0
	B	7
	C	8
2	A	1
	B	0
	C	6
3	A	1
	B	6
	C	1

Table 5: example of a source table

Let us assume the following setting:

- the source table is the one reported in Table 5
- the constraints present in query Q_1 are `BODY.price`$\geq$`0` $\wedge$ `HEAD.price`$\geq$`1`
- the constraints present in query Q_2 are `BODY.price`$\geq$`1` $\wedge$ `HEAD.price`$>$`5`
- the Optimizer decides to call the destructive algorithm.

here there follow the mining index queries needed to obtain the sets S_1, S_2, S_3 and S_4 along with the resulting sets of gid/item pairs.

1. BODY.price$\geq 0 \wedge \neg$BODY.price≥ 1
   S_1= [1 A] [2 B]
2. HEAD.price$\geq 1 \wedge \neg$HEAD.price> 5
   S_2= [2 A] [3 A] [3 C]
3. BODY.price≥ 0
   S_3= [1 A] [1 B] [1 C] [2 A] [2 B] [2 C] [3 A] [3 B] [3 C]
4. HEAD.price≥ 1
   S_4= [1 B] [1 C] [2 A] [2 C] [3 A] [3 B] [3 C]

If, on the contrary, the optimizer decides to exploit the constructive algorithm, the associated queries and lists would be the following ones.

1. BODY.price≥ 1
   S_1= [1 B] [1 C] [2 A] [2 C] [3 A] [3 B] [3 C]
2. HEAD.price> 5
   S_2= [1 B] [1 C] [2 C] [3 B]

2.4 Description of the algorithms

The goal of the algorithms is to derive the result of a new mining query Q_2 by means of the exploitation of a previous result R_1. Therefore we assume that there exists a query Q_1 stored in the database and such that Q_1 dominates Q_2. The first step made by both algorithms is to read rules from the result set of Q_1 and to build a data structure to keep track of the previous result. In our case we use a hierarchical representation implemented as a tree. We call this data structure *BHT* (Body-Head Tree).

In particular, we maintain two separate structures, one for the itemset that belongs to the body of the rules and one for the itemsets that belong to the head. We need such a distinction since we cast ourselves in a very general setting in which the constraints over the body of the rule may differ from the ones over the head of the rule. This implies that some of the items can only appear in the body part of a rule, while some others can only appear in the head part

of a rule. As a result, the BHT maintains a number of trees linked together. More in particular, we build a tree containing all the items which may appear in the body. A single path in this tree describe an itemset B which may appear in the body part of the rule. At the end of this path we put a link to a tree representing the itemsets which are allowed to appear in the head part of the rule when B forms the body.

As we already explained, both algorithms receive from the Optimizer the results of the queries made on the mining index. This information will be used to locate in the BHT the rules which does not satisfy the constraints anymore.

The mining algorithm implemented in the system at the present time stores in a structure very similar to the BHT a number of lists that resemble very much the lists we obtain from the optimizer [1]. In contrast, by exploiting the fact that the BHT we built from the rules in R_1 already contains a superset of the rules that will be given in output, we do not need to store those lists anymore. As we shall show, in fact, we can simply update a couple of counters. One is needed to keep track of the support for the complete rule and the one to keep track of the support for the body.

In the following we will adopt the following notation. Let r be a rule, let b be the body of r, and denote with G the number of total groups in the source table. We write $r.supp$ to denote the number of groups that contain r and $b.supp$ to denote the number of groups that contain b. As it is usual, we define the confidence of the rule $r.confidence$ to be $\frac{r.supp}{b.supp}$ and its frequency $r.frequency$ to be $\frac{r.supp}{G}$.

2.4.1 Destructive approach

The inputs of the destructive algorithm are the past result R_1 and the four sets S_1, S_2, S_3, and S_4. R_1 contains the set of rules that satisfy Q_1 along with their frequency and confidence. The destructive algorithm builds the BHT out of R_1, and stores for each rule the quantities $r.supp$ and $b.supp$. As it is easy to verify those quantities can be easily computed using $r.confidence$, $r.frequency$, and G.

Once the BHT has been built, the algorithm uses S_1,S_2,S_3, and S_4 to find those rules that need their $r.supp$ and/or their $b.supp$ counters to be decremented. In the following, we report the algorithm implementing the destructive approach and show the use of the S_i.

Before going into the details of the algorithm, let us notice that if S_1 is empty, then all the itemsets in the body part of rules belonging to R_1 satisfy the mining condition of Q_2. If S_2 is empty, then all itemsets in the head part of rules in R_1 satisfy the mining condition of Q_2. Otherwise (if at least one of the two is not empty), there exist in R_1 some rules for which the body and/or the head part satisfy the conditions in Q_1, but not the ones in Q_2. The aim of the algorithm is to find these itemsets in BHT and decrement their support count accordingly.

Let us now define ${I_g}^i = \{s | \langle s, g \rangle \in S_i\}$, i.e., we denote with I_g^i the items in the set S_i that are associated to the group g. Moreover, we define a new operator

$\uplus$ that build a new itemset by drawing one or more items from its left argument and zero or more items from its right argument. Then with the notation $I_1 \uplus I_2$ we denote the set of all those itemsets. For instance if $b \subseteq \{{I_g}^1 \uplus {I_g}^3\}$, then b is an itemset that contains at least one item in ${I_g}^1$ and zero or more items in ${I_g}^3$.

Here it follows the destructive algorithm.

Input: pointers to S_1, S_2, S_3, S_4*; Output:* R_2

1. If $S_1 \wedge S_2$ are non empty:
2. for all GID g $\in S_1 \wedge S_2$:
3. for all rule $r \in \{$ r:b⇒h | $(b \subseteq \{{I_g}^1 \uplus {I_g}^3\} \wedge\ h \subseteq\ {I_g}^4)\}$
4. if $r \in$ *BHT* :
5. (if they are not already marked) mark b and h of r;
6. decrement *r.supp* and *b.supp*;
7. for all rule $r \in \{$ r:b⇒h $|(b\ \subseteq {I_g}^3 \wedge\ h \subseteq \{{I_g}^1 \uplus\ {I_g}^4\})\}$
8. if $r \in$ *BHT* :
9. (if they are not already marked) mark b and h of r;
10. decrement *r.supp*;
11. remove the marks from all the b and h marked in this step;
12. If S_1 is non empty:
13. for all GID $g \in S_1 \wedge \neg \in S_2$:
14. for all rule $r \in \{$ r:b⇒h | $b \subseteq\ \{{I_g}^1 \uplus {I_g}^3\} \wedge h \subseteq\ {I_g}^4\}$
15. if $r \in$ *BHT* :
16. (if they are not already marked) mark b and h of r;
17. decrement *r.supp* and *b.supp*;
18. remove the marks from all the b and h marked in this step;
19. If S_2 is non empty:
20. for all GID $g \in S_2 \wedge \neg \in S_1$:
21. for all rule $r \in \{$ r:b⇒h $|b \subseteq {I_g}^3 \wedge h \subseteq \{{I_g}^1 \uplus\ {I_g}^4\}$
22. if $r \in$ *BHT* :
23. (if they are not already marked) mark b and h of r;
24. decrement *r.supp*;
25. remove the marks from all the b and h marked in this step;
26. For all rule $r \in$ *BHT* do
27. if *r.supp* and *b.supp* are sufficient then
28. insert r in R_2 ;

We notice that we are using the marks (steps 5.,9.,16.,23.) to avoid decrementing two times the same (body or rule) support.

2.4.2 An example of the Destructive Algorithm

In this section we give a complete example of the execution of the destructive algorithm. The example is the one reported in Section 2.3.1. The mining index queries that needs to be made and the sets S_1, S_2, S_3, and S_4 are reported in that section. The mining table has been reported in 5, while the result R_1 is reported in Table 6.

body	head	frequency	confidence
A	B	0.6	0.6
B	A	0.3	0.3
A	C	1	1
C	A	0.6	0.6
B	C	1	1
C	B	0.6	0.6
AB	C	1	1
AC	B	0.6	0.6
BC	A	0.6	0.6
A	BC	0.6	0.6
B	AC	0.6	0.6
C	AB	0.3	0.6

Table 6: The R_1 table

Here it follows what the operations the destructive algorithm performs for each of the groups. We use the symbol $\bullet$ to show which rules are marked in each step and the symbol $\circ$ to show the moment in which the marks are removed.

1. For the group 1 :

   the algorithm finds $\boxed{\text{1 A}}$ only in S_1 (step 13.). It finds in BHT the rules $A \uplus ABC \Rightarrow BC$:

   $A \Rightarrow B$ $\bullet$ if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset;

   $A \Rightarrow C$ $\bullet$ if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset;

   $A \Rightarrow BC$ $\bullet$ if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset;

   $AB \Rightarrow C$ $\bullet$ if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset;

   $AC \Rightarrow B$ $\bullet$ if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset;

   $\circ$ demarked all the marked itemsets;

2. For the group 2: the algorithm find $\boxed{\text{2 B}}$ in S_1 and $\boxed{\text{2 A}}$ in S_2 (step 2.).

It found in BHT the rules $B \uplus ABC \Rightarrow AC$ (step 3.):

$B \Rightarrow A$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

$B \Rightarrow C$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

$B \Rightarrow AC$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

$AB \Rightarrow C$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

$BC \Rightarrow A$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

It found in BHT the rules $ABC \Rightarrow A \uplus AC$ (step 7.):

$B \Rightarrow A$ •if they are not already marked, decrement only its rule support, and mark its head itemset.

$C \Rightarrow A$ •if they are not already marked, decrement only its rule support, and mark its head itemset.

$B \Rightarrow AC$ •if they are not already marked, decrement only its rule support, and mark its head itemset.

$BC \Rightarrow A$ •if they are not already marked, decrement only its rule support, and mark its head itemset.

◦ demarked all the marked itemsets;

3. For the group 3:

   the algorithm find $\boxed{\text{3 A}}$ and $\boxed{\text{3 C}}$ in S_2 (step 20.). It found in BHT the rules $ABC \Rightarrow AC \uplus ABC$ (step 21.):

   $A \Rightarrow C$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

   $B \Rightarrow C$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

   $AB \Rightarrow C$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

   $B \Rightarrow A$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

   $C \Rightarrow A$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

   $BC \Rightarrow A$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

   $B \Rightarrow AC$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

$A \Rightarrow BC$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

$C \Rightarrow AB$ •if they are not already marked, decrement its rule support and body support, and mark its body itemset and head itemset.

◦ demarked all the marked itemsets;

The result set R_2 is the reported in Table 7.

body	head	frequency	confidence
A	B	0.3	0.6
A	C	0.3	0.6
B	C	0.3	0.6
C	B	0.6	1
AC	B	0.3	0.6

Table 7: The R_2 table

2.4.3 Constructive approach

In the following we report the constructive algorithm. The algorithm takes the result set R_1 satisfying query Q_1 and returns the result set containing the rules that satisfy Q_2. We notice that in this algorithm we do not insert the rule support and the body support in the structure. In fact, both are initialized to 0. A complete example of the execution of the algorithm is reported in Section 2.4.4.

Input: pointers to S_1, S_2*; Output:* R_2

```
1. If S1 ∧ S2 are non empty:
2.  for all GID g ∈ S1:
3.    if ∃g ∈ S2:
4.    for all body b ⊆ Ig^1 increment b.supp;
5.    for all rule r ∈ {r : b ⇒ h | b ⊆ Ig^1 ∧ h ⊆ Ig^2}
6.      if r ∈ BHT :
7.        increment r.supp;
8. For all rule r ∈ BHT do
9.     if r.supp and b.supp are sufficient then
10.       insert r in R2 ;
```

2.4.4 An example of the Constructive Algorithm

In this section we give a complete example of the execution of the constructive algorithm. The example is the one reported in Section 2.3.1. The mining index queries that needs to be made and the sets S_1, S_2 are reported in that section. The mining table has been reported in 5, while the result R_1 is reported in Table 6

1. For the group 1: the algorithm find | 1 B | | 1 C | in S_1 and | 1 B | | 1 C | in S_2.

   It found in BHT all the bodies b$\subseteq BC$ (step 4.):

   B,C,BC and increment its *b.supp*

   Then, if found all the rules $BC \Rightarrow BC$ (steps 5.-6.):

   $C \Rightarrow B$ decrement its *r.support*

   $B \Rightarrow C$ decrement its *r.support*

2. For the group 2: the algorithm find | 2 A | | 2 C | in S_1 and | 2 C | in S_2.

   It found in BHT all the bodies b$\subseteq AC$ (step 4.):

   A,C,AC and increment its *b.supp*

   Then, if found all the rules $BC \Rightarrow BC$ (steps 5.-6.):

   $A \Rightarrow C$ decrement its *r.support*

3. For the group 3: the algorithm find | 3 A | | 3 B | | 3 C | in S_1 and | 3 B | in S_2.

   It found in BHT all the bodies b$\subseteq ABC$ (step 4.):

   A,B,AB,AC,BC and increment its *b.supp*

   Then, if found all the rules $ABC \Rightarrow B$ (steps 5.-6.):

   $A \Rightarrow B$ decrement its *r.support*

   $C \Rightarrow B$ decrement its *r.support*

   $AC \Rightarrow B$ decrement its *r.support*

The result set R_2 is reported in Table 7.

3 Conclusions

This document introduced an engine for query optimization in a data mining environment. We notice that the system at large and, hence, also the optimizer, supports a very general form of queries. In fact, we support a generalized form of constraint (context dependent ones [6]) and also admit different constraints to be given for the body part and for the head part of the rules that need to be extracted. We find both these generalizations to be useful for the investigator. Unfortunately, at the best of our knowledge, few systems currently support similar kind of constraints [3, 2].

As it has been explained, the optimization engine implemented so far supports the recognition of query equivalences and poses the bases for the recognition of query containments. Moreover, we introduced two novel mining algorithms (named incremental algorithms) that can exploit the containment relationship in order to provide a solution to the mining query in a faster and less resource demanding fashion. In order to support the operations of these algorithms we have also revised the concept of "mining index" [5]. Briefly stated, a mining index is a persistent data structure modeled by means of B+-tree allowing fast retrieval of the items and of the context for which the mining constraints are satisfied.

References

[1] Marco Botta, Rosa Meo, and Cinzia Malangone. Associaction rules extraction with mine rule operator. Technical Report RT73-2003, Università di Torino, 2003.

[2] J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. DMQL: A data mining query language for relational databases. *In Proceedings of SIGMOD-96 Workshop on Research Issues on Data Mining and Knowledge Discovery*, 1996.

[3] T. Imielinski, A. Virmani, and A. Abdoulghani. Datamine: Application programming interface and query language for database mining. *KDD-96*, pages 256–260, 1996.

[4] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *Proceedings of the 22st VLDB Conference*, Bombay, India, September 1996.

[5] Rosa Meo. Optimization of a language for data mining. In *Proc. of the 2003 ACM Symposium on Applied Computing*, Melbourne, Florida, 2003.

[6] Rosa Meo, Marco Botta, and Roberto Esposito. Query rewriting in itemset mining. In *Proceedings of the 6th International Conference On Flexible Query Answering Systems. LNAI (to appear).* Springer, 2004.

[7] Dick Tsur, Jeffrey D. Ullman, Serge Abiteboul, Chris Clifton, Rajeev Motwani, Svetlozar Nestorov, and Arnon Rosenthal. Query flocks: A generalization of association-rule mining. In *Proceedings of 1998 ACM SIGMOD International Conference Management of Data*, 1998.

[8] Haixun Wang and Carlo Zaniolo. User defined aggregates for logical data languages. In *Proc. of DDLP*, pages 85–97, 1998.