Querying Inductive Databases via Logic-Based User-Defined Aggregates

Fosca Giannotti and Giuseppe Manco

CNUCE - CNR
Via S. Maria 36. 56125 Pisa - Italy
{F.Giannotti,G.Manco}@cnuce.cnr.it

Abstract. We show how a logic-based database language can support the various steps of the KDD process by providing: a high degree of expressiveness, the ability to formalize the overall KDD process and the capability of separating the concerns between the specification level and the mapping to the underlying databases and datamining tools. We generalize the notion of Inductive Data Bases proposed in [4, 12] to the case of Deductive Databases. In our proposal, deductive databases resemble relational databases while user defined aggregates provided by the deductive database language resemble the mining function and results. In the paper we concentrate on association rules and show how the mechanism of user defined aggregates allows to specify the mining evaluation functions and the returned patterns.

1 Introduction

The rapid growth and spread of knowledge discovery techniques has highlighted the need to formalize the notion of knowledge discovery process. While it is clear which are the objectives of the various steps of the knowledge discovery process, little support is provided to reach such objectives, and to manage the overall process.

The role of domain, or background, knowledge is relevant at each step of the KDD process: *which attributes discriminate best, how can we characterize a correct/useful profile, what are the interesting exception conditions, etc.*, are all examples of domain dependent notions. Notably, in the evaluation phase we need to associate with each inferred knowledge structure some quality function [HS94] that measures its information content. However, while it is possible to define quantitative measures for certainty (e.g., estimated prediction accuracy on new data) or utility (e.g., gain, speed-up, etc.), notions such as novelty and understandability are much more subjective to the task, and hence difficult to define. Here, in fact, the specific measurements needed depend on a number of factors: the business opportunity, the sophistication of the organization, past history of measurements, and the availability of data.

The position that we maintain in this paper is that a coherent formalism, capable of dealing uniformly with induced knowledge and background, or domain, knowledge, would represent a breakthrough in the design and development of decision support systems, in several challenging application domains.

Other proposal in the current literature have given experimental evidence that the knowledge discovery process can take great advantage of a powerful knowledge-representation and reasoning formalism [14, 11, 15, 5]. In this context, the notion of *inductive database*, proposed in [4, 12], is a first attempt to formalize the notion of interactive mining process. An inductive database provides a unified and transparent view of both inferred (deductive) knowledge, and all the derived patterns, (the induced knowledge) over the data.

The objective of this paper is to demonstrate how a logic-based database language, such as $\mathcal{LDL}$++ [17], can support the various steps of the KDD process by providing: a high degree of expressiveness, the ability to formalize the overall KDD process and the capability of separating the concerns between the specification level and the mapping to the underlying databases and data mining tools. We generalize the notion of Inductive Databases proposed in [4, 12] to the case of Deductive Databases. In our proposal, deductive databases resemble relational databases while user defined aggregates provided by $\mathcal{LDL}$++ resemble the mining function and results. Such mechanism provides a flexible way to customize, tune and reason on both the evaluation function and the extracted knowledge. In the paper we show how such a mechanism can be exploited in the task of association rules mining. The interested reader is referred to an extended version [7] of this paper, which covers the bayesian classification data mining task.

2 Logic Database Languages

Deductive databases are database management systems whose query languages and storage structures are designed around a logical model of data. The underlying technology is an extension to relational databases that increases the power of the query language. Among the other features, the rule-based extensions support the specification of queries using recursion and negation.

We adopt the $\mathcal{LDL}$++ deductive database system, which provides, in addition to the typical deductive features, a highly expressive query language with advanced mechanisms for non-deterministic, non-monotonic and temporal reasoning [9, 18].

In deductive databases, the extension of a relation is viewed as a set of facts, where each fact corresponds to a tuple. For example, let us consider the predicate `assembly(Part, Subpart)` containing parts and their immediate subparts. The predicate `partCost(BasicPart, Supplier, Cost)` describes the basic parts, i.e., parts bought from external suppliers rather than assembled internally. Moreover, for each part the predicate describes the supplier, and for each supplier the price charged for it. Examples of facts are:

```
assembly(bike, frame).      partCost(top_tube, reed, 20).
assembly(bike, wheel).      partCost(fork, smith, 10).
assembly(wheel, nipple).
```

Rules constitute the main construct of LDL++ programs. For instance, the rule

$$\mathtt{multipleSupp(S)} \leftarrow \mathtt{partCost(P1,S,\_)}, \mathtt{partCost(P2,S,\_)}, \mathtt{P1} \neq \mathtt{P2}.$$

describes suppliers that sell more than one part. The rule corresponds to the SQL join query

```
SELECT P1.Supplier
FROM partCost P1, partCost P2
WHERE P1.Supplier = P2.Supplier
         AND P1.BasicPart <> P2.BasicPart
```

In addition to the standard relational features, LDL++ provides recursion and negation. For example, the rule

$$\mathtt{allSubparts(P,S)} \leftarrow \mathtt{assembly(P,S)}.$$
$$\mathtt{allSubparts(P,S)} \leftarrow \mathtt{allSubparts(P,S1)}, \mathtt{assembly(S1,S)}.$$

computes the transitive closure of the relation assembly. The following rule computes the least cost for each basic part by exploiting negation:

$$\mathtt{cheapest(P,C)} \leftarrow \mathtt{partCost(P,\_,C)}, \neg\mathtt{cheaper(P,C)}.$$
$$\mathtt{cheaper(P,C)} \leftarrow \mathtt{partCost(P,\_,C1)}, \mathtt{C1} < \mathtt{C}.$$

2.1 Aggregates

A remarkable capability is that of expressing distributive aggregates (i.e., aggregates computable by means of a distributive and associative operator), which are definable by the user [18]. For example, the following rule illustrates the use of a sum aggregate, which aggregates the values of the relation `sales` along the dimension `Dealer`:

$$\mathtt{supplierTot(Date,Place,sum\langle Sales\rangle)} \leftarrow \mathtt{sales(Date,Place,Dealer,Sales)}.$$

Such rule corresponds to the SQL statement

```
SELECT Date, Place, SUM(Sales)
FROM sales
GROUP BY Date, Place
```

From a semantic viewpoint, the above rule is a syntactic sugar for a program that exploits the notions of nondeterministic choice and XY-stratification [6, 17, 9]. In order to compute the following aggregation predicate

$$\mathtt{q(Y,aggr\langle X\rangle)} \leftarrow \mathtt{p(X,Y)}.$$

we exploit the capability of imposing a nondeterministic order among the tuples of the relation `p`,

$$\mathtt{ordP(Y,nil,nil)} \leftarrow \mathtt{p(X,Y)}.$$
$$\mathtt{ordP(Z,X,Y)} \leftarrow \mathtt{ordP(Z,\_,X)}, \mathtt{p(Y,Z)}, \mathtt{choice(X,Y)}, \mathtt{choice(Y,X)}.$$

Here nil is a fresh constant, conveniently used to simplify the program. If the base relation p is formed by k tuples for a given value s of Y, then there are $k!$ possible outcomes for the query ordP(X, Y), namely a set:

$$\{\texttt{ordP}(\texttt{s},\texttt{nil},\texttt{nil}),\texttt{ordP}(\texttt{s},\texttt{nil},\texttt{t}_1),\texttt{ordP}(\texttt{s},\texttt{t}_1,\texttt{t}_2),\ldots,\texttt{ordP}(\texttt{s},\texttt{t}_{\texttt{k}-1},\texttt{t}_\texttt{k})\}$$

for each permutation $\{(\texttt{t}_1, \texttt{s}), \ldots, (\texttt{t}_\texttt{k}, \texttt{s})\}$ of the tuples of P. Therefore, in each possible outcome of the mentioned query, the relation ordP is a total (intransitive) ordering of the tuples of p. The double choice constraint in the recursive rule specifies that the successor and predecessor of each tuple of p is unique.

As shown in [17], we can then exploit such an ordering to define distributive aggregates, inductively defined as $f(\{x\}) = g(x)$ and $f(S \cup \{x\}) = h(f(S), x)$. By defining the base and inductive cases by means of ad-hoc user-defined predicates single and multi, we can then obtain an incremental computation of the aggregation function:

$$\texttt{aggrP}(\texttt{Aggr},\texttt{Z},\texttt{nil},\texttt{C}) \leftarrow \texttt{ordP}(\texttt{Z},\texttt{nil},\texttt{X}),\texttt{X} \neq \texttt{nil},\texttt{single}(\texttt{Aggr},\texttt{X},\texttt{C}).$$

$$\texttt{aggrP}(\texttt{Aggr},\texttt{Z},\texttt{Y},\texttt{C}) \leftarrow \texttt{ordP}(\texttt{Z},\texttt{X},\texttt{Y}),\texttt{aggrP}(\texttt{Aggr},\texttt{X},\texttt{C}_1),\texttt{multi}(\texttt{Aggr},\texttt{Y},\texttt{C}_1,\texttt{C}).$$

Finally, the originary rule can be translated into

$$\texttt{q}(\texttt{Y},\texttt{C}) \leftarrow \texttt{ordP}(\texttt{Y},\_,\texttt{X}),\neg\texttt{ordP}(\texttt{Y},\texttt{X},\_),\texttt{aggrP}(\texttt{aggr},\texttt{Y},\texttt{X},\texttt{C}).$$

Example 1 ([18]). The aggregate sum can be easily defined by means of the following rules:

$$\texttt{single}(\texttt{sum},\texttt{X},\texttt{X}).$$

$$\texttt{multi}(\texttt{sum},\texttt{X},\texttt{SO},\texttt{SN}) \leftarrow \texttt{SN} = \texttt{SO} + \texttt{X}.$$

□

In [18], a further extension to the approach is proposed, in order to deal with more complex aggregation functions. Practically, we can manipulate the results of the aggregation function by means of two predicates freturn and ereturn. The rule definining the aggregation predicate is translated into the following:

$$\texttt{q}(\texttt{Z},\texttt{R}) \leftarrow \texttt{ordP}(\texttt{Z},\texttt{X},\texttt{Y}),\texttt{aggrP}(\texttt{aggr},\texttt{Z},\texttt{X},\texttt{C}),\texttt{ereturn}(\texttt{aggr},\texttt{Y},\texttt{C},\texttt{R}).$$
$$\texttt{q}(\texttt{Z},\texttt{R}) \leftarrow \texttt{ordP}(\texttt{Z},\texttt{X},\texttt{Y}),\neg\texttt{ordP}(\texttt{Z},\texttt{Y},\_),\texttt{aggrP}(\texttt{aggr},\texttt{Z},\texttt{Y},\texttt{C}),\texttt{freturn}(\texttt{aggr},\texttt{C},\texttt{R}).$$

where the first rule defines *early returns* (i.e., results of intermediate computations), and the second rule defines *final returns*, i.e., final results.

Example 2 ([18]). The aggregate maxpair considers tuples (c_i, n_i), where n_i is a real number, and returns the value c_i with the greater value of n_i. The aggregate can be defined by means of single, multi and freturn:

$$\texttt{single}(\texttt{maxpair},(\texttt{C},\texttt{P}),(\texttt{C},\texttt{P})).$$

$$\texttt{multi}(\texttt{maxpair},(\texttt{C},\texttt{P}),(\texttt{CO},\texttt{PO}),(\texttt{C},\texttt{P})) \leftarrow \texttt{P} \geq \texttt{PO}.$$
$$\texttt{multi}(\texttt{maxpair},(\texttt{C},\texttt{P}),(\texttt{CO},\texttt{PO}),(\texttt{CO},\texttt{PO})) \leftarrow \texttt{P} < \texttt{PO}.$$

$$\texttt{freturn}(\texttt{maxpair},(\texttt{CO},\texttt{PO}),\texttt{CO}).$$

□

3 Logic-Based Inductive Databases

In [4], an inductive database schema is defined as a pair $\mathcal{R} = (\mathbf{R}, (\mathcal{Q}_{\mathbf{R}}, e, \mathcal{V}))$, where $\mathbf{R}$ is a database schema, $\mathcal{Q}_{\mathbf{R}}$ is a collection of patterns, $\mathcal{V}$ is a set of result values and e is an evaluation function mapping each instance $\mathbf{r}$ of $\mathbf{R}$ and each pattern $\theta \in \mathcal{Q}_{\mathbf{R}}$ in $\mathcal{V}$. An inductive database instance is then defined as a pair $(\mathbf{r}, s)$, where $\mathbf{r}$ is an instance of $\mathbf{R}$ and $s \subseteq \mathcal{Q}_{\mathbf{R}}$.

A typical KDD process operates on both the components of an inductive database, by querying both components of the pair (assuming that s is materialized as a table, and that the value $e(\mathbf{r},\theta)$ is available for each value θ of s).

A simple yet powerful way of formalizing such ideas in a query language is that of exploiting user-defined aggregates. Practically, we can formalize the inductive part of an inductive database (i.e., the triple $(\mathcal{Q}_{\mathbf{R}}, e, \mathcal{V})$) by means of rules that instantiate the following general schema:

$$s(u\_d\_aggr\langle e, X_1, \ldots, X_n\rangle) \leftarrow \mathbf{r}(Y_1, \ldots, Y_m). \tag{1}$$

Intuitively, this rule defines the format of any subset s of $\mathcal{Q}_{\mathbf{R}}$. The patterns in s are obtained from a rearranged subset $\mathtt{X}_1, \ldots, \mathtt{X}_n$ of the tuples $\mathtt{Y}_1, \ldots, \mathtt{Y}_m$ in $\mathbf{r}$. The structure of s is defined by the formal specification of the aggregate u_d_aggr, in particular by the `freturn` rule.

The tuples resulting from the evaluation of such rule, represent patterns in $\mathcal{Q}_{\mathbf{R}}$ and depend by the evaluation function e. The computation of the evaluation function must be specified by u_d_aggr as well.

Example 3. Consider the patterns *"the items in the corresponding column of the relation* `transaction(Tid, Item, Price, Qty)` *with the average value more than a given threshold"*. The inductive database has $\mathbf{R} \equiv \mathtt{transaction}$, $\mathcal{Q}_{\mathbf{R}} = \{i | i \in dom(\mathbf{R}[Item])\}$, $\mathcal{V} = I\!\!R$ and $e(\mathbf{r}, i) = avg(\{p \times q | (t, i, p, q) \in \mathbf{r}\}$. The above inductive schema is formalized, according to (1) with the following rule:

$$\mathtt{s(avgTh\langle(\sigma, Itm, Val)\rangle) \leftarrow transaction(\_, Itm, Prc, Qty), Val = Prc \times Qty.}$$

Where the aggregate `avgThres` is defined, as usual, by means of the predicates

```
single(avgThres, (T, I, V), (T, I, V, 1)).

multi(avgThres, (T, I, VN), (T, I, VO, NO), (T, I, V, N)) ← V = VN + VO, N = NO + 1.
multi(avgThres, (T, I, VN), (T, I, VO, NO), (T, I, VO, NO)).
multi(avgThres, (T, I, VN), (T, IO, VO, NO), (T, I, VN, 1)) ← I ≠ IO.

freturn(avgThres, (T, I, V, N), (I, A)) ← A = V/N, A ≥ T.
```

For each item, both the sum and the count of the occurrences is computed. When all the tuples have been considered, the average value of each item is computed, and returned as answer if and only if it is greater than the given threshold. □

The advantage of such an approach is twofold. First, we can directly exploit the schema (1) to define the evaluation function e. Second, the "inductive" predicate s itself can be used in the definition of more complex queries. This defines a uniform way of providing support for both the deductive and the inductive components.

4 Association Rules

As shown in [2], the problem of finding association rules consist of two problems: the problem of finding frequent itemsets and consequently the problem to find rules from frequent itemsets. Frequent itemsets are itemsets that appear in the database with a given frequency. So, from a conceptual point of view, they can be seen as the results of an aggregation function over the set possible values of an attribute. Hence, we can refine the idea explained in the previous section, by defining a predicate p by means of the rule

$$\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_\mathtt{n},\mathtt{patterns}\langle(\mathtt{min\_supp},[\mathtt{Y}_1,\ldots,\mathtt{Y}_\mathtt{m}])\rangle) \leftarrow \mathtt{q}(\mathtt{Z}_1,\ldots,\mathtt{Z}_\mathtt{m}).$$

In this rule, the variables $\mathtt{X}_1,\ldots,\mathtt{X}_\mathtt{n},\mathtt{Y}_1,\ldots,\mathtt{Y}_\mathtt{m}$ are a rearranged subset of the variables $\mathtt{Z}_1,\ldots,\mathtt{Z}_\mathtt{k}$ of q. The aggregate `patterns` computes the set of predicates $p(s,f)$ where:

1. $s = \{l_1,\ldots,l_l\}$ is a rearranged subset of the values of $\mathtt{Y}_1,\ldots,\mathtt{Y}_\mathtt{m}$ in a tuple resulting from the evaluation of q.
2. f is the support of the set s, such that $f \geq \mathit{min\_supp}$.

It is easy to provide a (naive) definition of the `patterns` aggregate:

```
single(patterns, (Sp, Set), (SSet, Sp, 1)) ← subset(SSet, Set).

multi(patterns, (Sp, SetN), (SSetO, Sp, N), (SSetO, Sp, N)) ←
                                              ¬subset(SSetO, SetN).
multi(patterns, (Sp, SetN), (SSetO, Sp, N), (SSetO, Sp, N + 1)) ←
                                              subset(SSetO, SetN).
multi(patterns, (Sp, SetN), (SSetO, Sp, N), (SSet, Sp, 1)) ←
                              ¬subset(SSetO, SetN), subset(SSet, SetN),
                                       ¬subset(SSet, SSetO).

freturn(patterns, (SSet, Sp, N), (SSet, N)) ← N ≥ Sp.
```

For each tuple, the set of possible subsets are generated. The `single` predicate initializes the first subset that can be computed from the first tuple, by setting their frequency to 1. As soon as following tuples are examined (with the `multi` predicate), the frequency of the subsets computed before the tuple under consideration is incremented (provided that it is a subset of the current tuple), and the frequency of new subsets obtained from the current tuple are preset to 1.

The freturn predicate defines the output format and conditions for the aggregation predicate: a suitable answer is a pair (SubSet, N) such that SubSet is an itemset of frequency $N > Sp$, where Sp is the minimal support required.

A typical example application consists in the computation of the frequent itemsets of a basket relation:

$$\texttt{frequentPatterns}(\texttt{patterns}\langle(\texttt{m},\texttt{S})\rangle) \leftarrow \texttt{basketSet}(\texttt{S}).$$
$$\texttt{basketSet}(\langle\texttt{E}\rangle) \leftarrow \texttt{basket}(\texttt{T},\texttt{E}).$$

where the predicate basketSet collects the baskets in a set structure[1]. Rules can be easily generated from frequent patterns by means of rules like

$$\begin{array}{r}\texttt{rules}(\texttt{L},\texttt{R},\texttt{S},\texttt{C}) \leftarrow \texttt{frequentPatterns}(\texttt{A},\texttt{S}), \texttt{frequentPatterns}(\texttt{R},\texttt{S}_1),\\ \texttt{subset}(\texttt{R},\texttt{A}), \texttt{difference}(\texttt{A},\texttt{R},\texttt{L}), \texttt{C} = \texttt{S}/\texttt{S1}.\end{array} \quad (r_1)$$

Notice, however, that such an approach, though semantically clean, is very inefficient, because of the large amount of computations needed at each step[2]. In [10] we propose a technique which allows a compromise between loose and tight coupling, by adopting external specialized algorithms (and hence specialized data structures), but preserving the integration with the features of the language. In such proposal, inductive computations may be considered as aggregates, so that the proposed representation formalism is unaffected. However, the inductive task is performed by an external ad-hoc computational engine. Such an approach has the main advantage of ensuring ad-hoc optimizations concerning the mining task transparently and independently from the deductive engine. In our case the patterns aggregate is implemented with some typical algorithm for the computation of the association rules. (e.g., Apriori algorithm [2]). The aggregation specification can hence be seen as a middleware between the core algorithm and the data set (defined by the body of the rule) against which the algorithm is applied.

The rest of the section shows some examples of complex queries whithin the resulting logic language. In the following we shall refer to the table with schema and contents exemplified in 1.

Example 4. "Find patterns with at least 3 occurrences from the daily transactions of each customer":

$$\texttt{frequentPatterns}(\texttt{patterns}\langle(3,\texttt{S})\rangle) \leftarrow \texttt{transSet}(\texttt{D},\texttt{C},\texttt{S}).$$

$$\texttt{transSet}(\texttt{D},\texttt{C},\langle\texttt{I}\rangle) \leftarrow \texttt{transaction}(\texttt{D},\texttt{C},\texttt{I},\texttt{P},\texttt{Q}).$$

By querying frequentPatterns(F, S) we obtain, among the answers, the tuples $(\{pasta\}, 3)$ and $(\{pasta, wine\}, 3)$. □

[1] Again, in $\mathcal{LDL}$++ the capability of defining set-structures (and related operations) is guaranteed by the choice construct and by XY-stratification.

[2] Practically, the aggregate computation generates $2^{|I|}$ sets of items, where I is the set of different items appearing in the tuples considered during the computation. Pruning of unfrequent subsets is made at the end of the computation of all subsets. Notice, however, that clever strategies can be defined (e.g., computation of frequent maximal patterns [3]).

```
transaction(12-2-97, cust1, beer, 10, 10).
transaction(12-2-97, cust1, chips, 3, 20).
transaction(12-2-97, cust1, wine, 20, 2).
transaction(12-2-97, cust2, wine, 20, 2).
transaction(12-2-97, cust2, beer, 10, 10).
transaction(12-2-97, cust2, pasta, 2, 10).
transaction(12-2-97, cust2, chips, 3, 20).
transaction(13-2-97, cust2, jackets, 100, 1).
transaction(13-2-97, cust2, col_shirts, 30, 3).
transaction(13-2-97, cust3, wine, 20, 1).
transaction(13-2-97, cust3, beer, 10, 5).
transaction(13-2-97, cust1, chips, 3, 20).
transaction(13-2-97, cust1, beer,10,2).
transaction(15-2-97, cust1,pasta,2,10).
transaction(15-2-97, cust1,chips,3,10).
transaction(16-2-97, cust1,jackets,120,1).
transaction(16-2-97, cust2,wine,20,1).
transaction(16-2-97, cust2,pasta,4,8).
transaction(16-2-97, cust3, chips, 3, 20).
transaction(16-2-97, cust3,col_shirts,25,3).
transaction(16-2-97, cust3,brown_shirts,40,2).
transaction(18-2-97, cust2,beer,8,12).
transaction(18-2-97, cust2,beer,10,10).
transaction(18-2-97, cust2,chips,3,20).
transaction(18-2-97, cust2,chips,3,20).
transaction(18-2-97, cust3,pasta,2,10).
transaction(18-2-97, cust1,pasta,3,5).
transaction(18-2-97, cust1,wine,25,1).
transaction(18-2-97, cust1, chips, 3, 20).
transaction(18-2-97, cust1, beer, 10, 10).
```

Table 1. A sample `transaction` table.

Example 5. "Find patterns with at least 3 occurrences from the transactions of each customers":

$$\mathtt{frequentPatterns(patterns\langle(3,S)\rangle)} \leftarrow \mathtt{transSet(C,S)}.$$

$$\mathtt{transSet(C,\langle I\rangle)} \leftarrow \mathtt{transaction(D,C,I,P,Q)}.$$

Differently from the previous example, where transactions were grouped by customer and by date, the previous rules group transactions by customer. We then compute the frequent patterns on the restructured transactions

$\mathtt{transSet(cust1, \{beer, chips, jackets, pasta, wine\})}$
$\mathtt{transSet(cust2, \{beer, chips, col\_shirts, jackets, pasta, wine\})}$
$\mathtt{transSet(cust3, \{beer, brown\_shirts, chips, col\_shirts, pasta, wine\})}$

obtaining, e.g., the pattern $(\{\mathtt{beer, chips, pasta, wine}\}, 3)$. □

Example 6. "Find association rules with a minimum support 3 from daily transactions of each customer". This can be formalized by rule (r_1). Hence, by querying $\mathtt{rules(L,R,S,C)}$, we obtain the association rule $(\{\mathit{pasta}\}, \{\mathit{wine}\}, 3, 0.75)$.

We can further postprocess the results of the aggregation query. For example, the query $\mathtt{rules(\{A,B\},\{beer\},S,C)}$ computes "two-to-one" those rules where the consequent is the `beer` item. An answer is $(\{\mathtt{chips, wine}\}, \{\mathtt{beer}\}, 3, 1)$. □

Example 7. The query *"find patterns from daily transactions of high-spending customers (i.e., customers with at least 70 of total expense ad at most 3 items brought), such that each pattern has at least 3 occurrences"* can be formalized as follows:

$$\mathtt{frequentPatterns(patterns\langle(3,S)\rangle)} \leftarrow \mathtt{transSet(D,C,S,I,V)}, \mathtt{V} > 70, \mathtt{I} \leq 3.$$
$$\mathtt{transSet(D,C,\langle I\rangle,count\langle I\rangle,sum\langle V\rangle)} \leftarrow \mathtt{transaction(D,C,I,P,Q)}, \mathtt{V} = \mathtt{P} * \mathtt{Q}.$$

The query `frequentPatterns(F,S)` returns the patterns $(\mathtt{beer}, 3)$, $(\mathtt{chips}, 4)$ and $(\mathtt{beer}, \mathtt{chips}, 3)$ that characterize the class of high-spending customers. □

Example 8 ([10]). The query *"find patterns from daily transactions of each customer, at each generalization level, such that each pattern has a given occurrency depending from the generalization level"* is formalized as follows:

$$\begin{aligned}&\mathtt{itemsGeneralization(0,D,C,I,P,Q)} \leftarrow \mathtt{transaction(D,C,I,P,Q)}.\\&\mathtt{itemsGeneralization(I+1,D,C,AI,P,Q)} \leftarrow\\&\qquad\qquad \mathtt{itemsGeneralization(I,D,C,S,P,Q),category(S,AI)}.\end{aligned}$$

$$\mathtt{itemsGeneralization(I,D,C,\langle S\rangle)} \leftarrow \mathtt{itemsGeneralization(I,D,C,S,P,Q)}.$$

$$\begin{aligned}&\mathtt{freqAtLevel(I,patterns\langle(Supp,S)\rangle)} \leftarrow\\&\qquad\qquad \mathtt{itemsGeneralization(I,D,C,S),suppAtLevel(I,S)}.\end{aligned}$$

where the `suppAtLevel` predicate tunes the support threshold at a given item hierarchy. The query is the result of a tighter coupling of data preprocessing and result interpretation and postprocessing: we investigate the behaviour of rules over an item hierarchy. Suppose that the following tuples define a part-of hierarchy:

```
category(beer, drinks)          category(wine, drinks)
category(pasta, food)           category(chips, food)
category(jackets, wear)         category(col_shirts, wear)
category(brown_shirts, wear)
```

Then, by querying `freqAtLevel(I, F, S)` we obtain, e.g., (0, `beer`, `chips`, `wine`, 3), (1, `food`, 9), (1, `drinks`, 7) and (1, `drinks`, `food`, 6). □

Example 9. The query *"find rules that are interestingly preserved by drilling-down an item hierarchy"* is formalized as follows:

$$\begin{aligned}&\mathtt{rulesAtLevel(I,L,R,S,C)} \leftarrow \mathtt{freqAtLevel(I,A,S),freqAtLevel(I,R,S_1)},\\&\qquad\qquad \mathtt{subset(R,A),difference(A,R,L),C = S/S_1}.\end{aligned}$$

$$\begin{aligned}&\mathtt{preservedRules(L,R,S,C)} \leftarrow \mathtt{rulesAtLevel(I+1,L_1,R_1,S_1,C_1)},\\&\qquad\qquad \mathtt{rulesAtLevel(I,L,R,S,C),setPartOf(L,L_1)},\\&\qquad\qquad\quad \mathtt{setPartOf(R,R_1),C > C1}.\end{aligned}$$

Preserved rules are defined as those rules valid at any generalization level, such that their confidence is greater than their generalization[3]. □

5 Final Remark

We have shown that the mechanism of user-defined aggregates is powerful enough to model the notion of inductive database, and to specify flexible query answering capabilities.

[3] The choice for such an interest measure is clearly arbitrary and subjective. Other significant interest measures can be specified (e.g., the interest measure defined in [16]).

A major limitation in the proposal is efficiency: it has been experimentally shown that specialized algorithms (on specialized data structures) have a better performance than database-oriented approaches (see, e.g., [1]). Hence, in order to improve performance considerably, a thorough modification of the underlying database abstract machine should be investigated. Notice in fact that, with respect to ad hoc algorithms, when the programs specified in the previous sections are executed on a Datalog++ abstract machine, the only available optimizations for such programs are the traditional deductive databases optimizations [8]. Such optimizations techniques, however, need to be further improved by adding ad-hoc optimizations. For the purpose of this paper, we have been assuming to accept a reasonable worsening in performance, by describing the aggregation formalism as a semantically clean representation formalism, and demanding the computational effort to external ad-hoc engines [10]. This, however, is only a partial solution to the problem, in that more refined optimization techniques can be adopted. For example, in example 6, we can optimize the query by observing that directly computing rules with three items (even by counting the transactions with at least three items) is less expensive than computing the whole set of association rules, and then selecting those with three items. Some interesting steps in this direction have been made: e.g., [13] proposes an approach to the optimization of datalog aggregation-based queries, and in [13] a detailed discussion of the problem of the optimized computation of optimized computation of constrained association rules is made. However, the computational feasibility of the proposed approach to more general cases is an open problem.

References

1. R. Agrawal, S. Sarawagi, and S. Thomas. Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications. In *Procs. of ACM-SIGMOD'98*, 1998.
2. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, 1994.
3. R. Bayardo. Efficiently Mining Long Patterns from Databases. In *Proc. ACM Conf. on Management of Data (Sigmod98)*, pages 85–93, 1998.
4. J-F. Boulicaut, M. Klemettinen, and H. Mannila. Querying Inductive Databases: A Case Study on the MINE RULE Operator. In *Proc. 2nd European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD98)*, volume 1510 of *Lecture Notes in Computer Science*, pages 194–202, 1998.
5. U.M. Fayyad, G. Piatesky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining.* AAAI Press/the MIT Press, 1996.
6. F. Giannotti, D. Pedreschi, and C. Zaniolo. Semantics and Expressive Power of Non Deterministic Constructs for Deductive Databases. To appear in *Journal of Logic Programming.*
7. F. Giannotti and G. Manco. Querying inductive databases via logic-based user-defined aggregates. Technical report, CNUCE-CNR, June 1999. Available at `http://www-kdd.di.unipi.it`.
8. F. Giannotti, G[iuseppe Manco, M. Nanni, and D. Pedreschi. Nondeterministic, Nonmonotonic Logic Databases. Technical report, Department of Computer Science Univ. Pisa, September 1998. Submitted for publication.

9. F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi. Query Answering in Nondeterministic, Nonmonotonic, Logic Databases. In *Procs. of the Workshop on Flexible Query Answering*, number 1395 in Lecture Notes in Artificial Intelligence, march 1998.
10. F. Giannotti, G. Manco, M. Nanni, D. Pedreschi, and F. Turini. Integration of deduction and induction for mining supermarket sales data. In *Proceedings of the International Conference on Practical Applications of Knowledge Discovery (PADD99)*, April 1999.
11. J. Han. Towards On-Line Analytical Mining in Large Databases. *Sigmod Records*, 27(1):97–107, 1998.
12. H. Mannila. Inductive databases and condensed representations for data mining. In *International Logic Programming Symposium*, pages 21–30, 1997.
13. R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory Mining and Pruning Optimizations of Constrained Associations Rules. In *Proc. ACM Conf. on Management of Data (Sigmod98)*, June 1998.
14. S. Ceri R. Meo, G. Psaila. A New SQL-Like Operator for Mining Association Rules. In *Proceedings of The Conference on Very Large Databases*, pages 122–133, 1996.
15. W. Shen, K. Ong, B. Mitbander, and C. Zaniolo. Metaqueries for Data Mining. In *Advances in Knowledge Discovery and Data Mining*, pages 375–398. AAAI Press/The MIT Press, 1996.
16. R. Srikant and R. Agrawal. Mining Generalized Association Rules. In *Proc. of the 21th Int'l Conference on Very Large Databases*, 1995.
17. C. Zaniolo, N. Arni, and K. Ong. Negation and Aggregates in Recursive Rules: The $\mathcal{LDL}$++ Approach. In *Proc. 3rd Int. Conf. on Deductive and Object-Oriented Databases (DOOD93)*, volume 760 of *Lecture Notes in Computer Science*, 1993.
18. C. Zaniolo and H. Wang. Logic-Based User-Defined Aggregates for the Next Generation of Database Systems. In *The Logic Programming Paradigm: Current Trends and Future Directions.* Springer Verlag, 1998.