Specifying Mining Algorithms with Iterative User-Defined Aggregates

Fosca Giannotti, Giuseppe Manco, and Franco Turini, *Member*, *IEEE*

Abstract—We present a way of exploiting domain knowledge in the design and implementation of data mining algorithms, with special attention to frequent patterns discovery, within a deductive framework. In our framework, domain knowledge is represented by way of deductive rules, and data mining algorithms are specified by means of iterative user-defined aggregates and implemented by means of user-defined predicates. This choice allows us to exploit the full expressive power of deductive rules without loosing in performance. Iterative user-defined aggregates have a fixed scheme, in which user-defined predicates are to be added. This feature allows the modularization of data mining algorithms, thus providing a way to integrate the proper domain knowledge exploitation in the right point. As a case study, the paper presents how user-defined aggregates can be exploited to specify and implement a version of the a priori algorithm. Some performance analyzes and comparisons are discussed in order to show the effectiveness of the approach.

Index Terms—Data mining, query languages, constraint and logic languages, rule-based databases, user-defined aggregates, association rules.

1 Introduction and Motivations

The problem of incorporating data mining technology into query systems has been widely studied in the recent literature [7], [31], [10]. Here, a vision has been consolidated, in which the knolwedge discovery process is realized essentially as a query process. A crucial point, however, is to find the best trade off between expressiveness and performance. On the side of expressiveness, approaches based on deductive databases [14], [16], [18] offer very flexible query languages, which allow:

- direct exploitation of domain knowledge within the specification of the queries,
- specification of ad hoc interest measures that can help in evaluating the extracted knowledge, and
- modelization of the interactive and iterative features of knowledge discovery in a uniform way.

On the side of efficiency, it is unfeasible to directly specify and implement data mining algorithms within a query language [3], [4]. Typically, the performance of a data mining algorithm can be improved both by a smart constraining of the search space and by the exploitation of efficient data structures. The case of association rules is a typical example. Association rules are computed from frequent itemsets, which can be efficiently extracted by exploiting the *apriori property* [5] and by speeding-up comparisons and counting operations with the adoption of special data structures (e.g., lookup tables, hash trees, etc.).

Ways to integrate optimized data mining algorithms in query languages are proposed in [18], [32], [2]. In these approaches, data mining algorithms are modeled as separate modules. The interaction between the data mining algorithm and the query system is provided by defining a representation formalism of discovered patterns within the language and by collecting the data to be mined in an ad hoc format (a cache), directly accessed by the algorithm. However, such a decoupled approach has the main drawback of not allowing a systematic interaction within the search process. As an example, when using separate modules we cannot directly exploit domain knowledge within the algorithm, nor can we evaluate interest measures of the discovered patterns "on-the-fly."

In this paper, we propose an approach capable of providing both a direct specification of the algorithms in the query language and the use of separate modules for the efficient implementation of critical aspects of the algorithm. Our approach can be summarized as follows:

- Data mining algorithms are specified by *iterative user-defined aggregates* that provide the structure for the parameters and the results of the data mining procedures and the structure in which user-defined predicates have to be inserted.
- User-defined predicates allow the extension of a deductive database with new data types (like in the case of object-relational systems) that can be efficiently accessed and managed using ad hoc methods. This provides an open architecture upon which is based the *efficient implementation* of data mining algorithms, in the sense that the execution of expensive components may be demanded to external solvers.

- *F. Giannotti is with the Institute of Information Science and Technologies (ISTI-CNR), Italian National Research Council, Via Moruzzi 1, I56124 Pisa, Italy. E-mail: Fosca.Giannotti@isti.cnr.it.*
- *G. Manco is with the Institute of High Performance Computing and Networks (ICAR-CNR), Italian National Research Council, Via Bucci 41c, I87036 Rende (CS), Italy. E-mail: manco@icar.cnr.it.*
- *F. Turini is with the Department of Computer Science, University of Pisa, Via Buonarroti 2, I56127 Pisa, Italy. E-mail: turini@di.unipi.it.*

Manuscript received 2 Oct. 2002; revised 27 June 2003; accepted 28 Aug. 2003.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 117498.

- Finally, *knowledge discovery tasks* are defined by sets of clauses which use the appropriate user-defined aggregates and are combined with other predicates, defining the necessary rearrangements and reasoning on the source data at hand and the extracted knowledge.

Notice that, even though the approach of detailing the implementation of a data mining algorithm using a deductive language could appear cumbersome, nevertheless, the adoption of iterative aggregates can allow the compact specification of the general schema of the algorithm and the deferment of the implementation of critical points to user-defined predicates. More interestingly, the above framework allows the exploitation of domain knowledge both within the algorithm specification, to pursue domain-dependent optimizations, and within the knowledge discovery task definition, where the analysis is performed as a combination of deduction and induction.

In short, the contributions of the paper can be summarized as follows:

- We introduce the notion of iterative user-defined aggregate, as an extension of the notion of user-defined aggregate originally proposed in [46]. Iterative aggregates are a powerful tool for adding complex functional fragments within deductive databases and, as a consequence, they provide a basic step for modeling complex mining tasks within database languages.
- By exploiting aggregates, we formalize the notion of data mining query language as a deductive database programming language capable of representing and integrating both inferred and extracted knowledge. The expressiveness of a rule-based language combined with the capability of extracting induced knowledge makes such a deductive database language a powerful tool for formalizing the knowledge discovery process as a query process.
- As a case study, the paper presents how such a technique can be used to specify the frequent pattern mining task. We recall the `patterns` aggregate defined in [16] and provide a specification of such an aggregate as an iterative user-defined aggregate. The specification is essentially based on a version of the Apriori algorithm: Despite its datedness, the algorithm is still a significant example of how domain-knowledge specific hot-spot refinements can help in improving the performance of the mining. Hence, we provide an effective implementation of the algorithm by exploiting user-defined predicates.

The paper is organized as follows: In Section 2, we review the main features of a logic query language for data mining (originally proposed in [30]). Section 3 introduces the notion of iterative user-defined aggregates and justifies their use in the specification of data mining algorithms. In Section 4, we introduce the `patterns` iterative aggregate for mining frequent itemsets. Section 5 shows how user-defined predicates can be exploited to efficiently implement the aggregate. Finally, in Section 6, some performance analyses and comparisons are discussed in order to show the effectiveness of the approach and, in Section 7, a discussion on optimization opportunities is given.

2 Aggregates and Data Mining

In the recent literature, the need for combining knowledge discovery (induction) with background reasoning and querying (deduction) was formalized by the notion of *inductive database* [26]. Intuitively, an inductive database can be viewed as a relational database extended with the set of all patterns of a specified class that hold in the database. The induced patterns are available to the users as views, which can be either materialized or virtual. User queries can then combine base tables and induced patterns, thereby crossing the boundaries between deduction and induction. The user does not care whether he is dealing with deduced or induced knowledge, or whether the requested knowledge is materialized or virtual.

Some efforts were made to equip SQL with such inductive capabilities [23], [32], [27]. Such efforts, however, in general fail in providing a flexible and expressive query paradigm. Indeed, two important limitations of such linguistic extensions to SQL are, first, that they restrict to a predefined family of data mining tasks, and, second, the difficulty to introduce and take advantage of background knowledge. In this paper, we focus on $\mathrm{datalog}{+}{+}$, which we believe is more promising in this respect. As opposed to SQL-based approaches, deductive databases permit the encoding of ad hoc data mining tasks as well as the specification of background knowledge and, even more important, they allow to use one and the same language for both tasks.

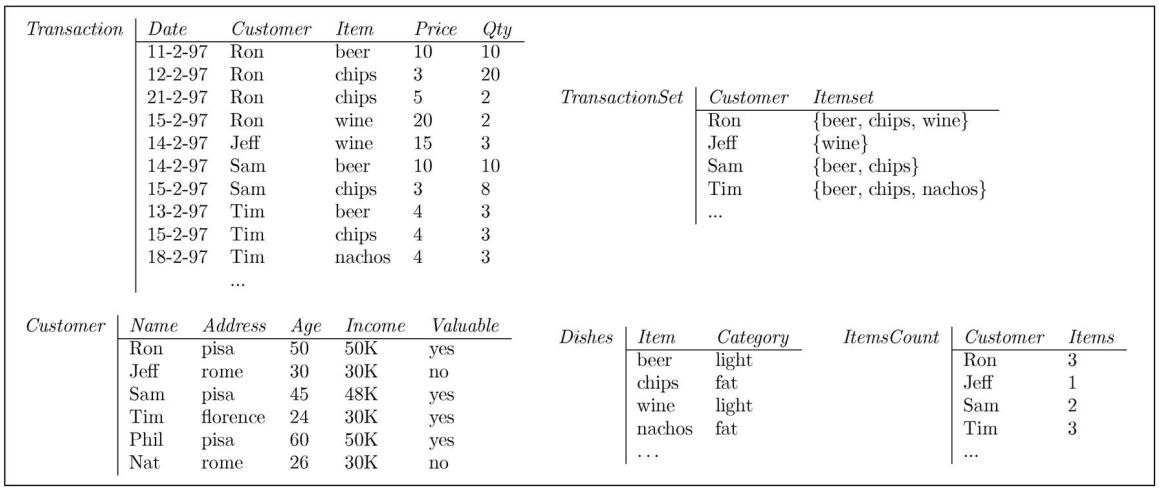
As a first step, we formalize the notion of logic-based knowledge discovery support environment as a deductive database programming language that models Horn clauses as well as inductive rules, where:

- Horn clauses specify both extensional relations and derived relations.
- Inductive rules contain a functional fragment, whose evaluation returns a relational table representing the patterns that are valid according to specific search criteria.

We choose to accomodate such functional fragments with the concept of aggregates, as we strongly believe that aggregates are a pervasive concept underlying many mining tasks.

Example 1. A frequent itemset $\{I_1, I_2\}$ is a database pattern with a validity specified by the estimation of the a posteriori probability $\Pr(\{I_1, I_2\}|\mathtt{r})$ (with respect to a given relational instance `r`). Such a probability can be estimated by means of *iceberg queries*. As an example, consider the following query referred to the *Transaction* relation of Fig. 1:

```
SELECT R1.Item, R2.Item, COUNT(Customer)
FROM Transaction R1, Transaction R2
WHERE R1. Customer = R2.Customer
  AND R1.Item <> R2.Item
GROUP BY R1.Item, R2.Item
HAVING COUNT(Customer) > σ
```



Transaction

Date	*Customer*	*Item*	*Price*	*Qty*
11-2-97	Ron	beer	10	10
12-2-97	Ron	chips	3	20
21-2-97	Ron	chips	5	2
15-2-97	Ron	wine	20	2
14-2-97	Jeff	wine	15	3
14-2-97	Sam	beer	10	10
15-2-97	Sam	chips	3	8
13-2-97	Tim	beer	4	3
15-2-97	Tim	chips	4	3
18-2-97	Tim	nachos	4	3
	...			

TransactionSet

Customer	*Itemset*
Ron	{beer, chips, wine}
Jeff	{wine}
Sam	{beer, chips}
Tim	{beer, chips, nachos}
...	

Customer

Name	*Address*	*Age*	*Income*	*Valuable*
Ron	pisa	50	50K	yes
Jeff	rome	30	30K	no
Sam	pisa	45	48K	yes
Tim	florence	24	30K	yes
Phil	pisa	60	50K	yes
Nat	rome	26	30K	no

Dishes

Item	*Category*
beer	light
chips	fat
wine	light
nachos	fat
...	

ItemsCount

Customer	*Items*
Ron	3
Jeff	1
Sam	2
Tim	3
...	

Fig. 1. Sample customers database.

It computes a functional aggregate that must satisfy a user-specified threshold constraint σ.

Example 2. A classifier is a model that describes a discrete attribute, called the *class*, in terms of other attributes. A classifier of the target attribute C of a relation $R(A_1, \ldots, A_n, C)$ is built from an instance r by estimating, for each possible value a of each attribute A_i, the probability $\Pr(C = c | A_i = a, \mathtt{r})$. Such a probability can be easily estimated by means of queries involving aggregates: For example, in order to build a classifier for the $Valuable$ attribute of the $Customer$ relation of Fig. 1, we need to evaluate queries of the form

```
SELECT Address as Feature, Valuable as Target,
       COUNT(*)
FROM Customer
GROUP BY Address, Valuable
  UNION
SELECT Age as Feature, Valuable as Target, COUNT(*)
FROM Customer
GROUP BY Age, Valuable
  UNION...
```

which populates a virtual count table [22] containing the needed statistics.

The datalog++ logic-based database language (and its implementation $\mathcal{LDL}++$) [44], [46], [17] provides a general framework for dealing with aggregates. For example, the following clause

$$\mathtt{items\_count(C, count\langle I\rangle)} \leftarrow \mathtt{transaction(D, C, I, P, Q)}$$

intensionally specifies the computation of the count aggregate with respect to the $Item$ attribute of the $Transaction$ table shown in Fig. 1. When evaluated, it computes the extension of the relation $ItemsCount$.

We use aggregates as the means for introducing mining primitives into the query language. More specifically, in our framework, we model an inductive rule defining a predicate P as a rule of the form

$$P(\alpha, Aggr\langle\beta\rangle) \leftarrow Q_1, \ldots, Q_n,$$

where

- $Aggr$ is the name of an aggregate, and
- the metavariables α and β represent terms obtained from the tuples resulting from the evaluation of the conjunctive query $Q_1, \ldots, Q_n$.

Such a construct provides a flexible and expressive model of interaction between induction and deduction. Indeed, the evaluation of $Q_1, \ldots, Q_n$ represents mainly the source data to mine, which in turn can be expressed by complex clauses. On the other side, $Aggr$ represents an interface to an inductive task, and provides its amalgamation within a deductive language. This permits encoding ad hoc data mining tasks as well as specifying background knowledge and, importantly, allows us to use one and the same language for both. Here is an example.

Example 3. With reference to the $Transaction$ table of Fig. 1, suppose we are interested in finding pairs of items purchased together by at least three customers. The following program performs this task:

$$\mathtt{pair(I_1, I_2, count\langle C\rangle)} \leftarrow \mathtt{transaction(D, C, I_1, \_, \_), transaction(D, C, I_2, \_, \_), I_1 \neq I_2.}$$
$$\mathtt{ans(I_1, I_2)} \leftarrow \mathtt{pair(I_1, I_2, C), C \geq 3.}$$

The first rule generates and counts all the possible pairs, and the second one selects the pairs purchased together by at least three customers. By querying ans(X, Y), we can obtain, for example, the answer ans(beer, chips) that models the corresponding inductive instance. Now, it is easy to see that, if an item has been purchased by less that three customers, then this item can never appear in the answer set. This principle, known as the *apriori* principle, can be considered as background knowledge, and it can be exploited to optimize the above program: Before looking at pairs of items, we filter items purchased by at least three customers:

$$\mathtt{singleton(I, count\langle C\rangle)} \leftarrow \mathtt{transaction(D, C, I, \_, \_).}$$
$$\mathtt{filter(I)} \leftarrow \mathtt{singleton(I, C), C \geq 3.}$$
$$\mathtt{pair(I_1, I_2, count\langle C\rangle)} \leftarrow \mathtt{transaction(D, C, I_1, \_, \_), transaction(D, C, I_2, \_, \_),}$$

$\texttt{filter}(\texttt{I}_1)$,$\texttt{filter}(\texttt{I}_2)$,$\texttt{I}_1 \neq \texttt{I}_2$.

$\texttt{ans}(\texttt{I}_1, \texttt{I}_2) \leftarrow \texttt{pair}(\texttt{I}_1, \texttt{I}_2, \texttt{C}), \texttt{C} \geq 3.$

Queries with filter conditions were investigated in [39]. Notice also that the clever exploitation of filter conditions is currently an active field of investigation [33].

3 Iterative User-Defined Aggregates

The datalog++ language allows the user to define explicitly distributive aggregate functions, inductively defined over a set S:

$$\begin{array}{lrcl} \textit{Base}: & f(\{x\}) & := & g(x) \\ \textit{Induction}: & f(S \cup \{x\}) & := & h(f(S), x). \end{array}$$

Users define aggregates in datalog++ by means of the `single` and `multi` predicates, where $\texttt{single}(\texttt{aggr}, \texttt{X}, \texttt{C})$ associates (in a nondeterministic order) a value `C` to the first tuple `X`, (this corresponds to the *Base* case), and $\texttt{multi}(\texttt{aggr}, \texttt{Old}, \texttt{X}, \texttt{New})$ computes the value `New` of the aggregate `aggr` associated to the current value `X` in the ordering, by exploiting the previous value `Old` (this corresponds to the *Inductive* case).

Example 4: ([46]). The aggregate `count` can be defined as follows:

$$\texttt{single}(\texttt{count}, \texttt{X}, 1).$$
$$\texttt{multi}(\texttt{count}, \texttt{SO}, \texttt{X}, \texttt{SN}) \leftarrow \texttt{SN} = \texttt{SO} + 1.$$

Such rules are implicitly used, for example, in the evaluation of the query $\texttt{items\_count}(\texttt{A}, \texttt{B})$ previously defined.

It turns out that an aggregate rule

$$\texttt{p}(\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{aggr}\langle\texttt{A}\rangle) \leftarrow Q_1, \ldots, Q_n$$

has a well-defined semantics, in terms of nondeterminism and XY-stratification supported by datalog++ [21], [44], [17], and it is amenable to an efficient implementation [46]. Moreover, in [46], an extension is proposed, which deals with more complex aggregation functions. There, the aggregate scheme is extended to include a *return* clause aimed at further restructuring the result of the aggregate.

Example 5. The sample variance of a set of values is defined as $s = \frac{1}{n-1}(\sum_i x_i^2 - \frac{1}{n}(\sum_j x_j)^2)$. The aggregate that computes such a measure over an attribute of a relation is defined by the following predicates:

$$\texttt{single}(\texttt{variance}, \texttt{X}, (\texttt{X}, \texttt{X} \times \texttt{X}, 1)).$$
$$\texttt{multi}(\texttt{variance}, (\texttt{S}, \texttt{SS}, \texttt{N}), \texttt{X}, (\texttt{S} + \texttt{X}, \texttt{SS} + \texttt{X} \times \texttt{X}, \texttt{N} + 1)).$$
$$\texttt{freturn}(\texttt{variance}, (\texttt{C}, \texttt{S}, \texttt{N}), 1/(\texttt{N} - 1) \times (\texttt{S} - 1/\texttt{N} \times \texttt{C} \times \texttt{C})).$$

The `single` and `multi` predicates simultaneously compute the sum, the size, and sum of the squares of a set of elements. The `freturn` predicate combines such values in order to compute the sample variance.

In this paper, we propose a further extension, capable of tackling the problem of defining aggregates requiring multiple scans over the data at hand. An example of such a class of aggregates is the absolute deviation $S_n = \sum_x |\overline{x} - x|$ of a set of n elements. In order to compute such an aggregate, we need to scan the data twice: First, to compute the mean value and second to compute the sum of the absolute differences.

A way of coping with the problem of multiple scans over data is to extend the semantics of a generic clause containing aggregates in order to guarantee a controlled form of iteration. More precisely, the specification of the aggregate can take into account some iterations over the `single` and `multi` predicates. We allow the explicit control of the iterations by means of a new predicate named `iterate`. Aggregates which can be specified by means of `single`/`multi`/`return` and `iterate` predicates are referred to in our framework as *iterative user-defined aggregates*, and their semantics is formalized as follows:[1]

Formally, an inductive clause containing an aggregate `aggr`,

$$\texttt{p}(\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{aggr}\langle\texttt{A}\rangle) \leftarrow B_1, \ldots B_n$$

(where $B_1, \ldots B_n$ are predicates, and $K_1, \ldots, K_m, A$ are terms obtained by an arrangement of the variables in $B_1, \ldots, B_n$) is rewritten into an equivalent program. The proposed rewriting extends the model of [46] by using XY-stratification [17] to formalize the iteration steps over the data:

$$\begin{array}{l} \texttt{cagr\_p}(\texttt{nil}, \texttt{Aggr}, \texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{X}, \texttt{New}) \leftarrow \\ \quad \texttt{next\_p}(\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{nil}, \texttt{X}), \texttt{X} \neq \texttt{nil}, \texttt{single}(\texttt{Aggr}, \texttt{X}, \texttt{New}). \\ \texttt{cagr\_p}(\texttt{I}, \texttt{Aggr}, \texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{Y2}, \texttt{New}) \leftarrow \\ \quad \texttt{next\_p}(\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{Y1}, \texttt{Y2}), \texttt{Y2} \neq \texttt{nil}, \\ \quad \texttt{cagr\_p}(\texttt{I}, \texttt{Aggr}, \texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{Y1}, \texttt{Old}), \\ \quad \texttt{multi}(\texttt{Aggr}, \texttt{Old}, \texttt{Y2}, \texttt{New}). \\ \texttt{cagr\_p}(\texttt{s}(\texttt{I}), \texttt{Aggr}, \texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{nil}, \texttt{New}) \leftarrow \\ \quad \texttt{next\_p}(\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{X}, \texttt{Y}), \neg\texttt{next\_p}(\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{Y}, \_), \\ \quad \texttt{cagr\_p}(\texttt{I}, \texttt{Aggr}, \texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{Y}, \texttt{Old}), \texttt{iterate}(\texttt{Aggr}, \texttt{Old}, \texttt{New}). \end{array}$$

Indeed, the main difference with regards to the schema shown in [46] is in the last clause that specifies the condition for iterating the aggregate computation. The activation (and evaluation) of such a clause is subject to the successful evaluation of the user-defined predicate `iterate`, so that any failure in evaluating it results in the termination of the computation. This guarantees that the proposed schema is conservative: Any aggregate specified solely by `single`, `multi`, and `freturn` is computed as usual.

Finally, the inductive rule is replaced by the following rule

$$\begin{array}{l} \texttt{p}(\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{S}) \leftarrow \\ \qquad \texttt{next\_p}(\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{X}, \texttt{Y}), \neg\texttt{next\_p}(\texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{Y}, \_), \\ \qquad \texttt{cagr\_p}(\texttt{I}, \texttt{Aggr}, \texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{Y}, \texttt{C}), \\ \qquad\quad \neg\texttt{cagr\_p}(\texttt{s}(\texttt{I}), \texttt{Aggr}, \texttt{K}_1, \ldots, \texttt{K}_\texttt{m}, \texttt{nil}, \texttt{C}) \\ \qquad \texttt{freturn}(\texttt{Aggr}, \texttt{C}, \texttt{S}). \end{array}$$

so that the aggregate produces a final result only when iteration stops. The following example shows how iterative aggregates can be specified according to the proposed scheme.

Example 6. By exploiting the `iterate` predicate, the absolute deviation S_n can be defined in the following way:

1. Further details can be found in [15].

```
p(K1, ..., Km, V)  ←  readTable(K1, ..., Km, A, Ao, Flag),
                      if (Flag = 1 then
                          if (iterate(aggr, Ao, Am) then
                              multi(aggr, A, Am, An),
                              writeTable(K1, ..., Km, An),
                              ereturn(aggr, A, Ao, V))
                          else
                              freturn(aggr, Ao, V))
                      else
                          (if (Flag = 2 then
                              single(aggr, A, An),
                              writeTable(K1, ..., Km, An),
                              ereturn(aggr, A, Ao, V))
                           else
                              (multi(aggr, A, Ao, An),
                               writeTable(K1, ..., Km, An),
                               ereturn(aggr, A, Ao, V))).
```

Fig. 2. Extended rule rewriting for iterative aggregates.

$\texttt{single}(\texttt{abserr}, \texttt{X}, (\texttt{nil}, \texttt{X}, 1))$.
$\texttt{multi}(\texttt{abserr}, (\texttt{nil}, \texttt{S}, \texttt{C}), \texttt{X}, (\texttt{nil}, \texttt{S} + \texttt{X}, \texttt{C} + 1))$.
$\texttt{multi}(\texttt{abserr}, (\texttt{M}, \texttt{D}), \texttt{X}, (\texttt{M}, \texttt{D} + (\texttt{M} - \texttt{X}))) \leftarrow \texttt{M} > \texttt{X}$.
$\texttt{multi}(\texttt{abserr}, (\texttt{M}, \texttt{D}), \texttt{X}, (\texttt{M}, \texttt{D} + (\texttt{X} - \texttt{M}))) \leftarrow \texttt{M} \leq \texttt{X}$.
$\texttt{iterate}(\texttt{abserr}, (\texttt{nil}, \texttt{S}, \texttt{C}), (\texttt{S}/\texttt{C}, 0))$.
$\texttt{freturn}(\texttt{abserr}, (\texttt{M}, \texttt{D}), \texttt{D})$.

The combined use of `multi` and `iterate` allows us to define two scans over the data: The first scan is defined to compute the mean value and the second one computes the sum of the absolute difference with the mean value.

Notice that the above rewriting is only a "semantic" rewriting. The actual rewriting exploits less declarative and more efficient features. Fig. 2 shows how, again by a simple extension of the schema proposed in [46], the inductive rule is rewritten. Here, we exploit two built-in predicates, namely, `readTable` and `writeTable`, that retrieve from and insert rows into an internal table. In the evaluation phase, the body of the inductive rule is evaluated, and the resulting tuples are stored into the internal table. When `readTable` is called, it incrementally gets values from the internal table. The `Flag` variable is instantiated with the status of the read operation, thus allowing a control in the computation of the aggregate.

3.1 Iterative Aggregates for Data Mining

Although the notion of iterative aggregate is in some sense orthogonal to the notion of inductive rule defined in Section 2, the main motivation for introducing iterative aggregates is that the iterative schema shown above is common to many data mining algorithms. Indeed, many traditional mining algorithms are defined as instances of an iterative schema where, at each iteration, some statistics are gathered from the data. Iterations stop when the extracted statistics are sufficient to the purpose of the task (i.e., they determine all the patterns), or alternatively when no further statistics can be extracted.

Example 7. The Apriori algorithm for the computation of frequent itemsets is based on the following schema:

1. At each iteration, frequencies of candidate itemsets are computed, and unfrequent candidate sets are removed. New candidate itemsets are computed by combining the frequent itemsets.
2. The cycle terminates when no further candidate itemsets are generated.

The two items identify the necessary conditions for specifying the computation of frequent itemsets by means of the iterative schema shown before.

Example 8. The K-means algorithm for the computation of clusters of tuples is defined by the following steps:

1. Initially, K random cluster centers are chosen.
2. At each iteration, each tuple is assigned to its closest cluster center, according to some similarity measure. Hence, cluster centers are recomputed averaging the values of each cluster component.
3. No more iteration is performed if the cluster centers are stable, i.e., if they do not change.

The specification of such operations within the iterative schema is immediate.

Example 9. A general schema for classification algorithms is the following [22]:

1. At each iteration, compute statistics over the attribute-set pairs.
2. Iterate until no further useful statistics are needed.

Such a recursive schema is common to many classification algorithms, such as decision-trees and Naive Bayes.

In this respect, the iterative schema discussed so far can be exploited for specifying steps of data mining algorithms at low granularity levels. We point out that a direct specification of the mining algorithm by means of iterative user-defined aggregates offers two main opportunities:

- It allows the use of background knowledge directly during the exploration of the search space, thus allowing the specification of more significant interest measures, both from a quantitative and a qualitative point of view.
- It allows a more integrated design of the various phases of the computational process, by pushing specific optimizations directly into the specification of data mining algorithms.

4 The `patterns` Iterative Aggregate

In the following, we concentrate on the problem of mining frequent patterns from a data set of transactions. The problem of frequent patterns discovery, which is the basis of many algorithms for mining association rules, has been extensively studied in the literature, both from the point of view of its specification in a data mining query language, and in the definition of efficient algorithms. According to Section 2, we can integrate such a mining task within the $\mathcal{LDL}++$ database language by means of a suitable inductive rule.

Definition 1. *Given a relation* `r`*, the* `patterns` *aggregate is specified by the rule*

$$\texttt{p}(\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}, \texttt{patterns}\langle(min\_supp, \texttt{Y})\rangle) \leftarrow \texttt{r}(\texttt{Z}_1, \ldots, \texttt{Z}_\texttt{m}),$$

where the variables $\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}, \texttt{Y}$ *are a rearranged subset of the variables* $\texttt{Z}_1, \ldots, \texttt{Z}_\texttt{m}$ *of* `r`*,* min_supp *is a value representing the*

support threshold, and the Y *variable denotes a set of elements. The aggregate* patterns *computes the set of predicates* $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n, \mathtt{s}, \mathtt{f})$*, where:*

1. $\mathtt{t}_1, \ldots, \mathtt{t}_n$ *are distinct instances of the variables* $\mathtt{X}_1, \ldots, \mathtt{X}_n$*, as they result from the evaluation of* r*;*
2. $\mathtt{s} = \{\mathtt{l}_1, \ldots, \mathtt{l}_k\}$ *is a subset of the value of* Y *in a tuple resulting from the evaluation of* r*;*
3. f *is the support of the set* s*, such that* $\mathtt{f} \geq min\_supp$.

Example 10. Let us consider the $Transaction$ relation in Fig. 1. In order to compute frequent patterns with at least three occurrences from transactions grouped by customer, we need to 1) organize the transactions, and 2) specify the mining aggregate. This is accomplished by the following clauses:

$$\mathtt{transactionSet}(\mathtt{C}, \langle \mathtt{I} \rangle) \leftarrow \mathtt{transaction}(\mathtt{D}, \mathtt{C}, \mathtt{I}, \mathtt{P}, \mathtt{Q}).$$
$$\mathtt{ans}(\mathtt{patterns}\langle(3, \mathtt{IS})\rangle) \leftarrow \mathtt{transactionSet}(\mathtt{C}, \mathtt{IS}).$$

The first clause groups tuples in the $Transaction$ relation on $Customer$. As a result, we obtain the $TransactionSet$ relation shown in Fig. 1. The second clause is an inductive rule: the $\mathtt{patterns}\langle(3, \mathtt{IS})\rangle$ aggregate extracts frequent itemsets from the grouped items. After the evaluation of the second clause, $ans(s, n)$ holds if s is a set of items purchased by $n \geq 3$ customers.

We are now in the position of exemplifying the usefulness of a query language where deduction and induction are integrated. While induction is used to learn patterns from data, deduction can be used both for preprocessing the data and for postprocessing the results of the mining step.

Example 11. Let us consider again the relations of Fig. 1. Suppose we are interested in the following task: Find sets s of "light" items (i.e., items involving low-calories food, as described by the background knowledge in the $Dishes$ relation), purchased by at least three customers living in Pisa. The following datalog++ program performs this task:

$$\begin{aligned}
\mathtt{itemsets}(\mathtt{patterns}\langle(3, \mathtt{IS})\rangle) \leftarrow\ & \mathtt{transactionSet}(\mathtt{C}, \mathtt{IS}),\\
& \mathtt{customer}(\mathtt{C}, \mathtt{pisa}, \mathtt{A}, \mathtt{I}, \mathtt{V}).\\
\mathtt{ans}(\mathtt{S}, \mathtt{N}) \leftarrow\ & \mathtt{itemsets}(\mathtt{S}, \mathtt{N}), \neg\mathtt{notLight}(\mathtt{S}).\\
\mathtt{notLight}(\mathtt{S}) \leftarrow\ & \mathtt{I} \in \mathtt{S}, \mathtt{dishes}(\mathtt{I}, \mathtt{X}), \mathtt{X} \neq \mathtt{light}.
\end{aligned}$$

In the first, inductive rule, frequent patterns are computed from purchases made by customers living in pisa. After the evaluation of this clause, $itemsets(s, n)$ holds if s is a set of items purchased by $n \geq 3$ customers living in pisa. The second and third clause restrict the result computed by the second clause to the sets of light items.

We can provide an explicit specification of the patterns aggregate as an iterative aggregate. That is, we can directly specify an algorithm for computing frequent patterns, by defining the predicates single, multi, freturn, and iterate.

The simplest specification adopts the purely declarative approach of *generating* all the possible itemsets, and then *testing* their frequency. The proposed algorithm is shown in Fig. 3. It is easy to specify such a naive algorithm by means of the iterative scheme proposed in Section 3:

Algorithm Naive(TB, σ);
Input: a transaction base **TB**, a support threshold σ;
Output: a set *Result* of frequent itemsets
Method: let initially $Result = \emptyset$, $C = \emptyset$, $n = 0$.

1. **foreach** transaction $s \in$ **TB do**
2. n++;
3. **foreach** itemset $x \subseteq s$
4. add x to C;
5. let $v = n \times \sigma$;
6. **foreach** itemset $x \in C$
7. $supp(x) = 0$;
8. **foreach** transaction $s \in$ **TB do**
9. **foreach** $x \in C$ **such that** $x \subseteq s$ **do** $supp(x)++$;
10. $Result := \{x \in C | supp(x) \geq v\}$;

Fig. 3. Naive Algorithm for computing frequent itemsets.

$$\begin{aligned}
\mathtt{single}(\mathtt{patterns}, (\mathtt{Sp}, \mathtt{S}), ((\mathtt{Sp}, 1), \mathtt{IS})) \ &\leftarrow \mathtt{subset}(\mathtt{IS}, \mathtt{S}).\\
\mathtt{multi}(\mathtt{patterns}, ((\mathtt{Sp}, \mathtt{N}), _), (\mathtt{Sp}, \mathtt{S}),\\
((\mathtt{Sp}, \mathtt{N}+1), \mathtt{IS})) &\leftarrow \mathtt{subset}(\mathtt{IS}, \mathtt{S}).\\
\mathtt{multi}(\mathtt{patterns}, ((\mathtt{Sp}, \mathtt{N}), \mathtt{IS}), _, ((\mathtt{Sp}, \mathtt{N}+1), \mathtt{IS})).&\\
\mathtt{multi}(\mathtt{patterns}, (\mathtt{Sp}, \mathtt{IS}, \mathtt{N}), (\mathtt{Sp}, \mathtt{S}),\\
(\mathtt{Sp}, \mathtt{IS}, \mathtt{N}+1)) \ &\leftarrow \mathtt{subset}(\mathtt{IS}, \mathtt{S}).\\
\mathtt{multi}(\mathtt{patterns}, (\mathtt{Sp}, \mathtt{IS}, \mathtt{N}), (\mathtt{Sp}, \mathtt{S}),\\
(\mathtt{Sp}, \mathtt{IS}, \mathtt{N})) \ &\leftarrow \neg\mathtt{subset}(\mathtt{IS}, \mathtt{S}).\\
\mathtt{iterate}(\mathtt{patterns}, ((\mathtt{Sp}, \mathtt{N}), \mathtt{IS}), (\mathtt{Sp} \times \mathtt{N}, \mathtt{IS}, 0)).&\\
\mathtt{freturn}(\mathtt{patterns}, (\mathtt{Sp}, \mathtt{IS}, \mathtt{N}), (\mathtt{IS}, \mathtt{N})) \ &\leftarrow \mathtt{N} \geq \mathtt{Sp}.
\end{aligned}$$

Such a specification requires two main iterations. In the first iteration (specified by the first three rules, corresponding to Steps 1-4 of Fig. 3), the set of possible subsets are generated for each tuple in the data set. The iterate predicate initializes the counter of each candidate itemset, and activates the computation of its frequency (performed by the remaining multi rules, corresponding to Steps 6-9 of Fig. 3). The computation terminates when the frequency of all itemsets has been computed, and frequent itemsets are returned as answers (by means of the freturn rule). Notice that the freturn predicate defines the output format for the aggregation predicate: a suitable answer is a pair (IS, N) such that IS is an itemset of frequency $\mathtt{N} > \mathtt{Sp}$, where Sp is the minimum support required.[2]

Clearly, the above implementation is extremely inefficient: The computation of the aggregate involves an exponential number of candidate itemsets to be considered and, for each itemset under consideration, the frequency has to be evaluated. As a consequence, the approach exhibits the following drawbacks:

- No pruning strategy is exploited: Unfrequent subsets are discarded at the end of the computation of the frequencies of all the subsets.
- No optimized data structure is used to speed up the computation.

Neverthless, the direct specification offers many opportunities for pushing domain-dependent constraints inside the algorithm itself. For example, the first two clauses can

2. Hereinafter, we suppose that the minimum support is expressed in percentage.

Algorithm Apriori($\mathcal{TB}$, σ);
Input: a transaction base $\mathcal{TB}$, a support threshold σ;
Output: a set *Result* of frequent itemsets
Method: let initially $Result = \emptyset$, $k = 1$.

1. $C_1 = \{a | a \in \mathcal{I}\}$; /* *init phase* */
2. **while** $C_k \neq \emptyset$ **do**
3. **foreach** itemset $c \in C_k$ **do**
4. $supp(c) = 0$;
5. **foreach** $b \in \mathcal{TB}$ **do** /* *count phase* */
6. **foreach** $c \in C_k$ **such that** $c \subseteq b$ **do** $supp(c) + +$;
7. $L_k := \{c \in C_k | \ supp(c) > \sigma\}$; /* *prune phase* */
8. $Result := Result \cup L_k$; /* *itemsets phase* */
9. $C_{k+1} := \{c_i \cup c_j | c_i, c_j \in L_k \wedge |c_i \cup c_j| = k+1, \ \forall c \subset c_i \cup c_j$ such that $|c| = k : c \in L_k\}$; /* *enhance phase* */
10. $k := k + 1$;
11. **end while**

Fig. 4. The Apriori Algorithm for computing frequent itemsets.

be replaced with the following clauses, which more properly handle the generation of patterns satisfying the query formulated in Example 11.

$$\texttt{single}(\texttt{patterns}, (\texttt{Sp}, \texttt{S}), ((\texttt{Sp}, 1), \texttt{IS})) \leftarrow \texttt{subset}(\texttt{IS}, \texttt{S}), \neg\texttt{notLight}(\texttt{S}).$$
$$\texttt{multi}(\texttt{patterns}, ((\texttt{Sp}, \texttt{N}), \_), (\texttt{Sp}, \texttt{S}), ((\texttt{Sp}, \texttt{N}+1), \texttt{IS})) \leftarrow \texttt{subset}(\texttt{IS}, \texttt{S}), \neg\texttt{notLight}(\texttt{S}).$$

More practically, we can consider a smarter exploration of the search space, by analyzing finer properties which can be modeled as background knowledge within the language itself. A paradigmatic example is the well-known Apriori Algorithm shown in Fig. 4, that implements a pruning strategy based on the observation that each frequent itemset contains only frequent subsets. Again, the algorithm can be specified using the scheme of iterative aggregates. Initially, we specify the init phase,

$$\texttt{single}(\texttt{patterns}, (\texttt{Sp}, \texttt{S}), ((\texttt{Sp}, 1), \texttt{IS})) \leftarrow \texttt{single\_isets}(\texttt{S}, \texttt{IS}).$$
$$\texttt{multi}(\texttt{patterns}, ((\texttt{Sp}, \texttt{N}), \texttt{IS}), (\texttt{Sp}, \texttt{S}), ((\texttt{Sp}, \texttt{N}+1), \texttt{ISS})) \leftarrow \texttt{single\_isets}(\texttt{S}, \texttt{SS}), \texttt{union}(\texttt{SS}, \texttt{IS}, \texttt{ISS}).$$

where the predicate single_isets specified, e.g., by the clause

$$\texttt{single\_isets}(\texttt{S}, \langle\{(\texttt{M}, 0)\}\rangle) \leftarrow \texttt{member}(\texttt{M}, \texttt{S}).$$

collects all the single items occurring in any transaction. The subsequent iterations resemble the steps of the Apriori Algorithm, namely, counting the candidate itemsets, pruning unfrequent candidates and generating new candidates:

$$\texttt{iterate}(\texttt{patterns}, ((\texttt{Sp}, \texttt{N}), \texttt{S}), (\texttt{Sp} \times \texttt{N}, \texttt{S})).$$
$$\texttt{iterate}(\texttt{patterns}, (\texttt{Sp}, \texttt{S}), (\texttt{Sp}, \texttt{SS})) \leftarrow \texttt{prune}(\texttt{Sp}, \texttt{S}, \texttt{IS}), \texttt{generate\_candidates}(\texttt{IS}, \texttt{SS}).$$
$$\texttt{multi}(\texttt{patterns}, (\texttt{Sp}, \texttt{IS}), (\texttt{Sp}, \texttt{S}), (\texttt{Sp}, \texttt{ISS})) \leftarrow \texttt{count\_isets}(\texttt{IS}, \texttt{S}, \texttt{ISS}).$$
$$\texttt{freturn}(\texttt{patterns}, (\texttt{Sp}, \texttt{ISS}), (\texttt{IS}, \texttt{N})) \leftarrow \texttt{member}((\texttt{IS}, \texttt{N}), \texttt{ISS}), \texttt{N} \geq \texttt{Sp}.$$

Such an approach exploits a substantial optimization by avoiding the check of a large portion of unfrequent itemsets. However, the implementation of the main operations of the algorithm concerns the predicates prune, generate_candidates, and count_isets. As a consequence, the efficiency of the approach depends on the efficient implementation and evaluation of such predicates.

In order to address the efficiency issues related to an explicit specification of such predicates in $\mathcal{LDL}++$, we can adopt a more effective approach, based on a compromise between loose and tight coupling, and formerly studied in [18]. In practice, the allowed types of the $\mathcal{LDL}++$ system are extended to include more complex structures, like in the case of Object-Relational systems, and provide some built-in predicates that efficiently manipulate such structures:

$$\texttt{single}(\texttt{patterns}, (\texttt{Sp}, \texttt{S}), ((\texttt{Sp}, 1), \texttt{T})) \leftarrow \texttt{init}(\texttt{S}, \texttt{T}).$$
$$\texttt{multi}(\texttt{patterns}, (\texttt{Sp}, \texttt{S}), ((\texttt{Sp}, \texttt{N}), \texttt{T}), ((\texttt{Sp}, \texttt{N}+1), \texttt{T})) \leftarrow \texttt{init}(\texttt{S}, \texttt{T}).$$
$$\texttt{iterate}(\texttt{patterns}, ((\texttt{Sp}, \texttt{N}), \texttt{T}), (\texttt{Sp} \times \texttt{N}, \texttt{T})) \leftarrow \texttt{prune}(\texttt{Sp}, \texttt{T}), \texttt{enahance}(\texttt{T}).$$

$$\texttt{multi}(\texttt{patterns}, (\texttt{Sp}, \texttt{S}), (\texttt{Sp}, \texttt{T}), (\texttt{Sp}, \texttt{T})) \leftarrow \texttt{count}(\texttt{S}, \texttt{T}).$$
$$\texttt{iterate}(\texttt{patterns}, (\texttt{Sp}, \texttt{T}), (\texttt{Sp}, \texttt{T})) \leftarrow \texttt{prune}(\texttt{Sp}, \texttt{T}), \texttt{enhance}(\texttt{T}).$$

$$\texttt{freturn}(\texttt{patterns}, (\texttt{Sp}, \texttt{T}), (\texttt{I}, \texttt{S})) \leftarrow \texttt{itemset}(\texttt{T}, (\texttt{I}, \texttt{S})).$$

In such a schema, the variable T represents the reference (e.g., the object identifier) to a *Hash-Tree* structure (detailed in Section 5). The predicates init, count, enhance, prune, and itemset are *user-defined predicates* that implement, in a procedural language such as C++, complex operations over the given Hash-tree structure. More specifically:

- The $\texttt{init}(\texttt{I}, \texttt{T})$ predicate initializes and updates the frequencies of the 1-itemsets available from I in T.
- The $\texttt{count}(\texttt{I}, \texttt{T})$ predicate updates the frequencies of each itemset in T according to the transaction I.
- The $\texttt{prune}(\texttt{M}, \texttt{T})$ predicate removes all the itemsets in T whose frequencies are less than M.

- The `enhance(T)` predicate combines the frequent k-itemsets in `T` and generates the candidate $k+1$-itemsets.
- Finally, the `itemset(T,S)` predicate extracts the frequent itemset `I` (with frequency `S`) from `T`.

The above schema maintains a declarative specification parametric to the intended meaning of the user-defined predicates adopted. Moreover, the schema minimizes the "black-box" structure of the algorithm needed to obtain fast counting of candidate itemsets and efficient pruning. Thus, it offers many opportunities for optimizing the execution of the algorithm both from a database optimization perspective and from a "constraints embedding" perspective [33].

The key issue is, of course, the capability of providing a suitable semantics and an efficient implementation of user-defined predicates. The following section explicitly deals with such issues. We have chosen the Apriori Algorithm to provide a proof of the concept of improving the performance of inductive procedures integrated into datalog++ because of its conceptual neatness and its historical importance. Many other, more effective algorithms for frequent pattern discovery have been proposed focused both on improving the efficiency of the classical Apriori Algorithm [36], [38], [35] and on mining more significant sets of frequent itemsets [43], [24], [6], [8]. Neverthless, we claim that these approaches might be integrated in our system with a similar strategy.

5 User-Defined Predicates for Data Mining

In order to support complex database applications, many relational database systems allow user-defined functions. Such functions can be invoked within queries making it easier for developers to implement their applications with significantly greater efficiency. The adoption of such features in a logic-based system provides even greater impact since they allow a user to develop large programs by *hot-spot refinement* [11]. The user writes a large $\mathcal{LDL}++$ program, validates its correctness and identifies the hot-spots, i.e., predicates in the program that are highly time consuming. Then, he can rewrite those hot-spots more efficiently in a procedural language, such as C++, maintaining the rest of the program in $\mathcal{LDL}++$.

The formal model for Datalog programs with external predicates is developed in [11], where the notion of *computed database* is introduced. The computed database is a finite set of infinite relations. For example, $<$, $>$, and other arithmetic predicates fall into this category. Conceptually, any external procedure, e.g., a program written in C++, can be viewed as a computed predicate, i.e., an infinite relation with some *finiteness constraints*. Such constraints specify the input/output requirements. A computed predicate `p` is said to satisfy a finiteness constraint of the form $\overline{X} \rightarrow \overline{Y}$ if and only if for each tuple μ in `p`, the set of tuples $\{\nu[\overline{Y}] | \nu \in \mathtt{p} \text{ and } \nu[\overline{X}] = \mu[\overline{X}]\}$ is finite. The finiteness constraint must be explicitly stated in the case of a computed predicate. This is done by specifying the input arguments $\overline{X}$ of the procedure corresponding to the computed predicate, and guaranteing that the procedure computes the whole set of values for $\overline{Y}$ in finite time, given any value for $\overline{X}$. It is responsibility of the query evaluator to invoke the procedure only after a value for $\overline{X}$ is known. Notice also that this imposes an ordering on the structure of a program.

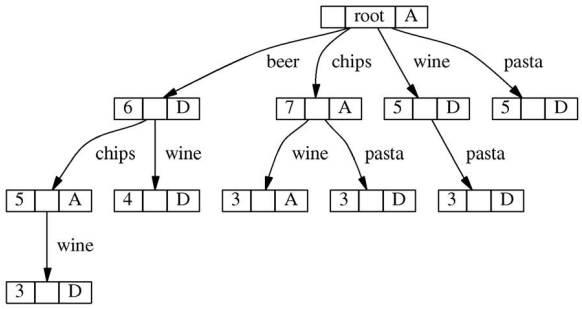


Fig. 5. Hash tree with items from the $Transaction$ relation.

$\mathcal{LDL}++$ allows the definition of external predicates written in C++, by providing mechanisms to convert objects between the $\mathcal{LDL}++$ representation and the external representations. The ad hoc use of such mechanisms provides new data types inside the $\mathcal{LDL}++$ model, in the style of Object-relational databases [1]. For example, a reference to a C++ object can be returned as an answer, or given as an input, and the management of such a user-defined object can be demanded to a set of external predicates.

We adopt such a model to implement hot-spot refinements to mining algorithm. In the following, we describe the implementation of an enhanced version of the Apriori Algorithm described in Fig. 4 by means of user-defined predicates. As described in Section 4, we use a variant of the hash-tree structure introduced in [5], which is essentially a prefix-tree with an hash table associated to each node. Fig. 5 shows an example istance of such a tree. An edge is labeled with an item, so that paths from the root to any other node represent itemsets. Each node is labeled with a tag denoting the support of the corresponding itemset. An additional tag denotes whether the node can generate new candidates, i.e., if the corresponding itemset can be further extended. Differently from the hash tree structure introduced in [5], the explicit representation of each itemset as a path in the tree guarantees a higher efficiency in several operations over the available itemsets (such as comparisons, merging operations, and counting), at the cost of higher memory requirements.

The predicates `init`, `count`, `enhance`, `prune`, and `itemet` introduced in Section 4 are *user-defined predicates* implementing complex operations, exemplified in Fig. 6, over such a tree structure:

- The `init(I,T)` predicate initializes and updates the frequencies of the 1-itemsets available from `I` in `T` (Fig. 6a). For each item found in the transaction `I`, either the item is already in the tree (in which case its counter is updated), or it is inserted and its counter set to 1.
- The `count(I,T)` predicate updates the frequencies of each itemset in `T` according to the transaction `I` (Fig. 6c). We define a simple recursive procedure that, starting from the first element of the current transaction, traverses the tree from the root to the leaves. When a leaf at a given depth is found, the corresponding counter is incremented.
- The `prune(M,T)` predicate removes from `T` all the itemsets whose frequencies are less than the support threshold represented by `M` (Figs. 6b and 6d). Leaf

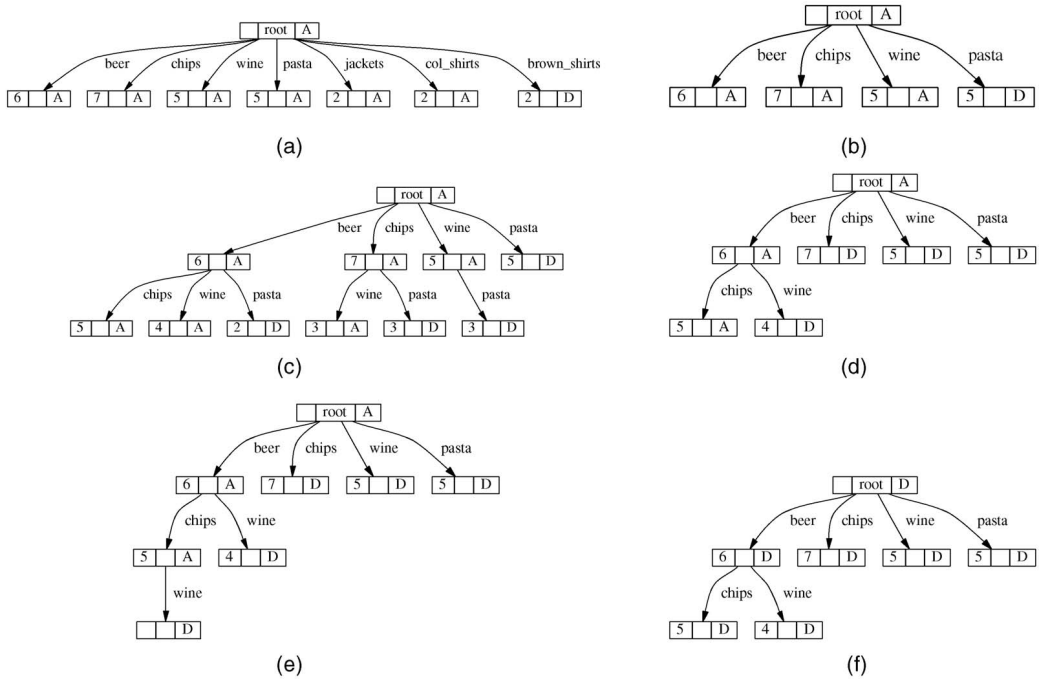


Fig. 6. (a) Tree initialization. (b) Pruning. (c) Tree enhancement and counting. (d) Pruning. (e) Tree enhancement. (f) Cutting.

nodes at a given depth (representing the size of the candidates) are removed if their support is lower than the given threshold.

- The `enhance(T)` predicate combines the frequent k-itemsets in `T` and generates the candidate itemsets of size $k+1$. New candidates are generated in two steps. In the first step, a leaf node at depth k is merged with each of its siblings, and new leaf nodes are generated. For example, in Fig. 6e, the node representing the itemset $\{\texttt{beer}, \texttt{chips}\}$ is merged with its sibling representing $\{\texttt{beer}, \texttt{wine}\}$, thus generating the new node representing $\{\texttt{beer}, \texttt{chips}, \texttt{wine}\}$. In order to ensure that every new node represents an actual candidate of size $k+1$, we need to check whether all the subsets of the itemset of size k are actually in the hash tree. The check is performed by a traversal of the path from the new node to the root node; for each node considered in the traversal, we simply check whether its subtree is also a subtree of its parent. Subtrees that do not satisfy such a requirement are cut (Fig. 6f).
- Finally, the `itemset(T,(I,S))` predicate extracts the frequent itemset `I` (whose frequency is `S`) from `T`. Since each leaf node represents an itemset, generation of itemsets is quite simple. The tree is traversed and itemsets are built accordingly. Notice that, differently from the previous predicates, where only one invocation was allowed, the `itemset` predicate allows multiple calls, providing an answer for each itemset found.

6 Performance Evaluation

In this section, we analyze the impact of the architecture we described in the previous sections to the process of extracting frequent itemsets from data. The analysis provides both insights on further possible improvements and shows that in this present form the loss of performance with respect to ad hoc implementations is not such as to induce one to give up the flexibility offered by the proposed environment.

The performance analysis that we undertook compared the effect of mining frequent itemsets according to three different architectural choices:

1. $\mathcal{LDL}++$, the implementation of the mining aggregate `patterns`, by means of the iterative aggregate specified in the previous section.
2. $DB2\ Batch$, an Apriori implementation that retrieves data from a SQL DBMS, stores such data in an intermediate structure and then performs the basic steps of the algorithm using such structures. Such an implementation conforms to the CacheMine approach, and it is common in many data mining software tools which provide access to data resident on DBMS. Conceptually, such an implementation can be thought of as the architectural support for an SQL extension, like, e.g., the `MINE RULE` construct shown in [32]. The main motivation here is to compare the effects of an implementation of a mining construct supported by a robust DBMS (IBM DB2 in our case) with a similar one in the $\mathcal{LDL}++$ deductive database.
3. $DB2\ interactive$, an Apriori implementation in which data are read tuple by tuple from the DBMS. This approach is quite easy to implement and manage, but has the main disadvantage of the large cost of context switching between the DBMS and the mining process. Since user-defined predicates need also context switching, it is interesting to see how the approach behaves compared to the $\mathcal{LDL}++$ approach.

data set	30		20		10		5		3		1	
T10.I2.D100	26	3	54	3	362	87	2000	2181	7742	16437	27720	236046
T10.I2.D500	6	0	28	6	132	120	492	489	1248	1362	13172	30291
T10.I2.D1K	6	0	28	6	110	135	386	453	1022	1473	8540	18534
T10.I2.D5K	4	0	22	9	98	84	380	543	890	1539	6726	15108
T10.I2.D10K	4	0	24	9	100	87	382	549	896	1569	6832	15085
T10.I2.D50K	4	0	22	9	94	63	368	516	864	1599	6180	14073
T5.I2.D100K	2	0	2	0	2	0	2	0	2	0	228	207

Fig. 7. Frequent itemsets and rules over synthesized data.

In addition, we compared the above solutions with a plain Apriori implementation that reads data from a sequential binary file. We used such an implementation to keep track of the actual computational effort of the algorithm on the given data size when no data retrieval and context switching overhead is required. Essentially, comparing the performance of a database-oriented solution with regard to Apriori allows to evaluate the loss in performance due to the overhead of the database engine. We expect the plain Apriori implementation to perform substantially better than the other approaches. Neverthless, the loss in performance can be reputed acceptable if the approach under consideration scales with an overhead bound by a constant factor with regard to Apriori.

We tested the effect of a very simple form of mining query—one that is expressible also in the SQL extensions—that retrieves data from a single table and applies the mining algorithm. In $\mathcal{LDL}++$ terms, the experiments were performed by querying $\texttt{ans}(min\_supp, \texttt{R})$, where ans was defined as

$$\texttt{ans}(\texttt{S}, \texttt{patterns}\langle(\texttt{S}, \texttt{ItemSet})\rangle) \leftarrow \texttt{transactionSet}(\texttt{C}, \texttt{ItemSet}).$$

The $\texttt{transactionSet}(\texttt{C}, \texttt{ItemSet})$ relation is a materialized table, containing tuples as shown in Fig. 1. It is worth recalling that the flexibility of the $\mathcal{LDL}++$ system allows one to define more complex mining queries, such as, e.g.,

$$\texttt{q}(\texttt{S}) \leftarrow \texttt{ans}(min\_supp, \texttt{S}), constr(\texttt{S}),$$

where $constr(\texttt{S})$ specifies a complex constraint over the itemset S. Now, the datalog++ language allows the direct specification of $constr(\texttt{S})$ within the language itself. Furthermore, the approach based on iterative aggregates allows us to tune the specification of the patterns aggregate by pushing such constraints within the algorithm itself. Such queries are hardly definable in simple terms in relational systems based on SQL with the consequence that processing requires an unacceptable programming overhead. Hence, a comparison of SQL-based systems and Datalog-based systems is significant only on equivalent fragments of the two languages.

Since the main objective of the experiments is to compare the performance of the architectural choices, we analyzed the performance on the given data sets for varying values of the support min_supp. In order to populate the transactionSet predicate (and its relational counterpart), we used the synthetic data generation utility described in [37, Section 2.4.3]. Data generation can be tuned according to the usual parameters: the number of transactions $|\mathcal{D}|$, the average size of the transactions $|T|$, the average size of the maximal potentially frequent itemsets $|I|$, the number of maximal potentially frequent itemsets $|\mathcal{I}|$, and the number of items N. We fixed $|I|$ to 2, and $|T|$ to 10 since such parameters affect the size of the frequent itemsets. All the remaining parameters were adjusted according to increasing values of $\mathcal{D}$: As soon as $\mathcal{D}$ increases, $|\mathcal{I}|$ and N are increased as well.

Fig. 7 describes the data sets used as benchmarks. For each data set, the table describes the number of frequent itemsets (and relevant associations) at the given support values. The choice of the first data set is motivated by the need to estimate the contribution of the $\mathcal{LDL}++$ implementation to the generation of results, i.e., how performance is affected when the number of generated itemsets is very large. Data sets with size ranging from 500 to 100K are useful to estimate the scalability of the approach with respect to the size of the data set. In particular, the last data set allows to measure the contribution of the tuple retrieval phase from the DBMS (or the $\mathcal{LDL}++$ internal system) in the cache population phase.

Figs. 8 and 9 show how the performance of the various solutions changes according to increasing values of $\mathcal{D}$ and decreasing values of the support. Experiments were made on a Linux system with two 400Mhz Intel Pentium II processors, with 128Mb RAM. Alternatives 2 and 3 were implemented using the IBM DB2 universal database v6.1.

From the figures, it can be seen that, as expected, the *DB2 interactive* solution gives the worst results: Since a cursor is maintained within the internal buffer of the database server, the main contribution to the cost is given by the frequent context switching between the application and the database server [4]. Moreover, decreasing values of support strongly influence its performance: Lower support values influence the size of the frequent patterns and, hence, multiple scans over the data are required.

Concerning $\mathcal{LDL}++$, it is worth noting that the adoption of an internal table, as described in Section 3, makes $\mathcal{LDL}++$ essentially compliant to the *Cache-Mine* approach. Moreover, we tried two different versions of the $\mathcal{LDL}++$ system. The first one adopts nested loops to implement aggregates, and a "lazy" management of intermediate results (that is, all the intermediate results are maintained in the internal table). A more efficient alternative uses hashing and implements an "eager" management (intermediate results are deleted as soon as they are used). Such differences have a strong impact on the performance: Indeed, the eager version substantially

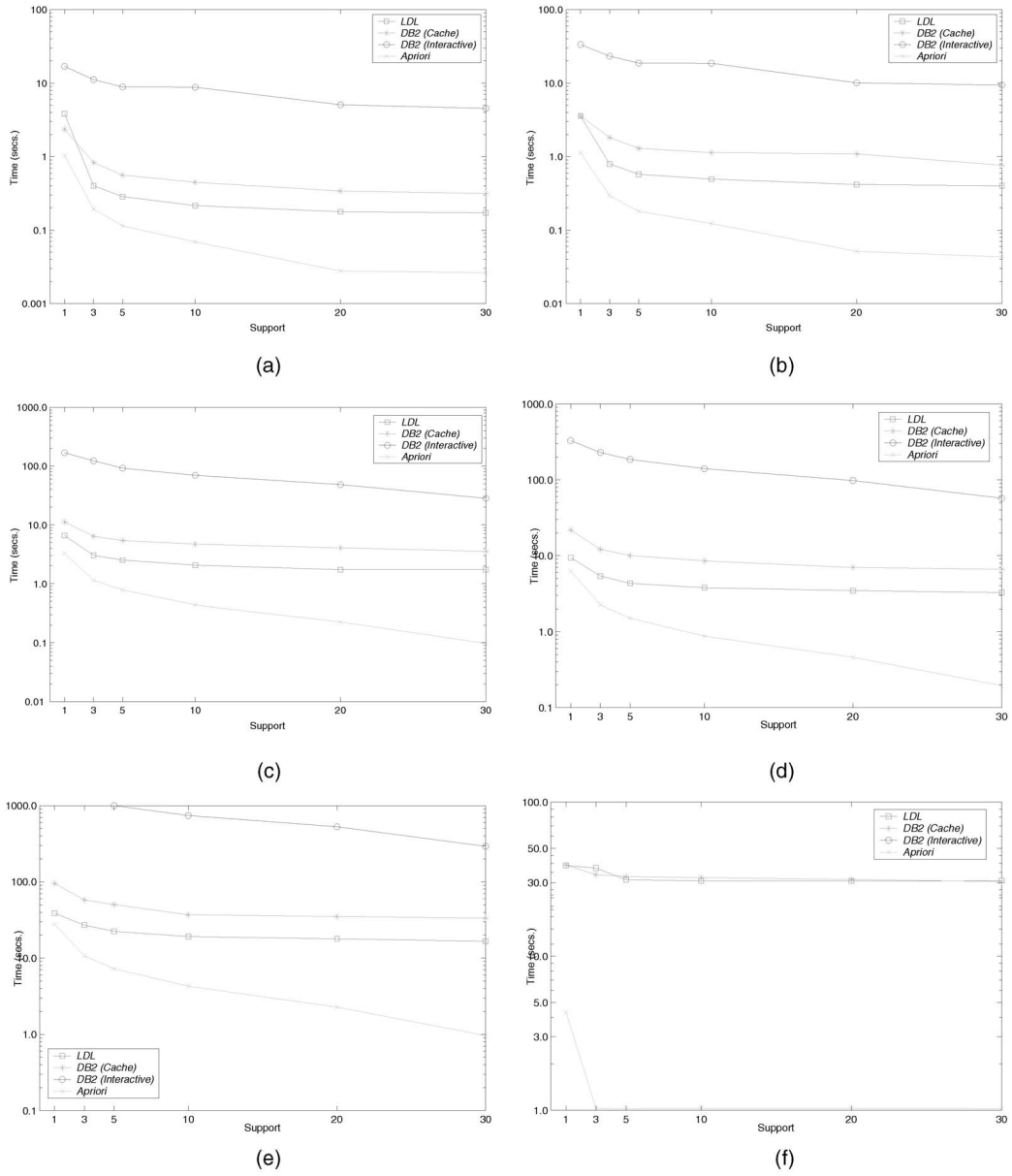


Fig. 8. Performance graphs. (a) $\mathcal{D} = 500,\ N = 500,\ |\mathcal{I}| = 100$. (b) $\mathcal{D} = 1,000,\ N = 1,000,\ |\mathcal{I}| = 100$. (c) $\mathcal{D} = 5,000,\ N = 3,000,\ |\mathcal{I}| = 100$. (d) $\mathcal{D} = 10,000,\ N = 3,000,\ |\mathcal{I}| = 100$. (e) $\mathcal{D} = 50,000,\ N = 5,000,\ |\mathcal{I}| = 100$. (f) $\mathcal{D} = 100,000,\ N = 30,000,\ |\mathcal{I}| = 1,000$.

outperforms the lazy version. Moreover, the five graphs of Figs. 8a, 8b, 8c, 8d, and 8e show that the $\mathcal{LDL}++$ approach (with eager implementation of aggregates) outperforms the *DB2 Batch* approach. This situation is even more evident in the three graphs of Figs. 9a, 9b, and 9c, where the overall situation is summarized and the performance of the approaches is compared for different values of the data size (in log scale), but with a fixed support. The performance graphs have a smooth (almost linear) curve.

The processing overhead of the deductive engine can be appreciated in the graph of Fig. 8f where, except for the lowest value of the support, the cost of the computation is mainly given by the data scan. Here, only low support values influence the result values (i.e., itemsets are generated only with support $\leq 1\%$) and, hence, the contribution of the mining algorithm to the overall cost is not relevant.

It is interesting to evaluate the contribution of the deductive component to the overall performance of the $\mathcal{LDL}++$ system. The graph of Fig. 9d summarizes the performance of the $\mathcal{LDL}++$ system for increasing values of the support and of the data size. The ratio between the data preprocessing of $\mathcal{LDL}++$ and the application of the Apriori algorithm (i.e., the context switching overhead) is shown in the graph of Fig. 9e. More precisely, the graph plots the value $(l-a)/l$, where l is the performance of the $\mathcal{LDL}++$ approach, and a is the performance of the Apriori Algorithm. The ratio is close to 1 when the internal management phase is predominant with respect to the application of the algorithm. As we can see, this ratio is particularly high with the last data set that does not contain frequent itemsets (except for very low support values) and, hence, the predominant computational cost is due to context switching. The other curves show that the performance worsens according to the algorithm.

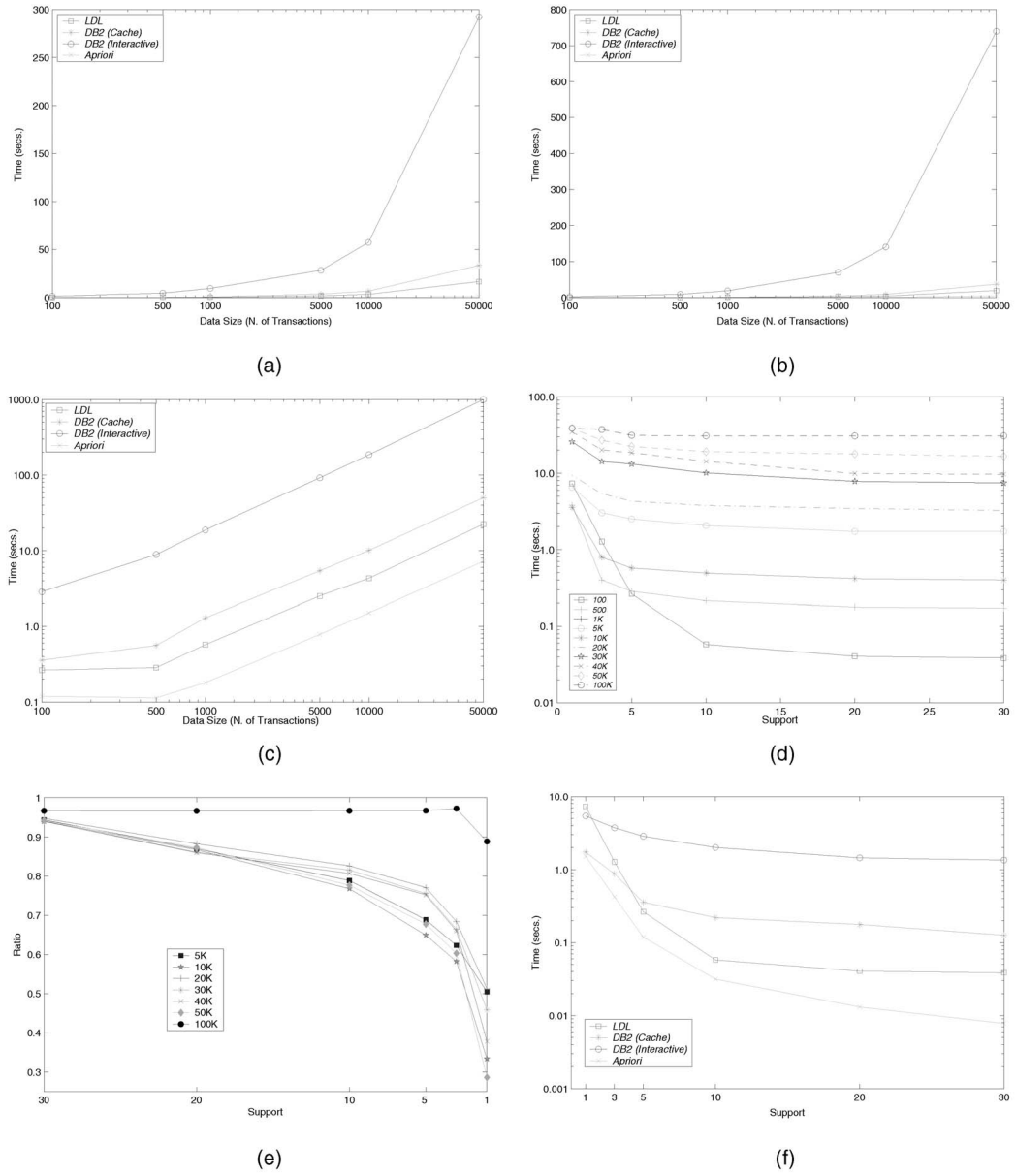


Fig. 9. Performance graphs (cont.). (a) Support = 30 percent. (b) Support = 1 percent. (c) Support = 5 percent. (d) $\mathcal{LDL}++$ scalability. (e) Context switching overhead. (f) $\mathcal{D} = 100$, $N = 100$, $|\mathcal{I}| = 100$, $|T| = 5$.

To conclude, it is interesting to observe the contribution of the result coding phase to the performance of the approach. In such an implementation, an itemset is represented as a complex term of the form $itemset(s, n)$. For example, an answer of the $\mathcal{LDL}++$ system to the query $\texttt{ans}(30, \texttt{R})$ is given by the predicate $\texttt{ans}(30, \texttt{itemset}(\{\texttt{beer}, \texttt{chips}\}, 5))$. The coding of such a result has influence on the overall cost, as shown in Fig. 9f, where the performance of the $\mathcal{LDL}++$ approach has a notable worsening, even over a very small data set. Such a behavior is mainly due to the generation of a very large number of itemsets associated with such a data set, as observed in Fig. 7.

To summarize, the following conclusions can be drawn from the experiments.

1. In general, the mechanism of user-defined predicates embedded into the system provides an acceptable performance, compared to different solutions based on DBMS. In particular, the context switching overhead makes the performance of the *DB2* approaches generally worst than the $\mathcal{LDL}++$ approach. This effect is partially mitigated by the *DB2 batch* approach, in which an internal cache is adopted, thus separating the querying phase from the mining phase. When the querying phase is predominant over the mining phase, we can expect that a *DB2 batch* approach can outperform the $\mathcal{LDL}++$ approach. Such a behavior finds its explanation in the processing overhead of the deductive system with respect to the relational system provided in DB2, which can be quantified by a constant factor. However, in situations where a stronger interaction between mining and querying is needed, an architectural solution similar to the one adopted by $\mathcal{LDL}++$ appears more suitable. This is also confirmed by similar approaches based on SQL, such as the ATLaS system [41].

2. The implementation in $\mathcal{LDL}++$ is less performant than the *Apriori* implementation in C. The slow-down can be explained by two main reasons: First, the overhead due to the coding of the algorithms directly as a user-defined aggregate, and second the context-switching overhead due to the encoding of the results. However, the loss in performance due to the coding is bound by a constant factor. This demonstrates that rather complex algorithms and data structures can be succintly expressed in $\mathcal{LDL}++$, thus granting the capability of specifying and executing complex mining queries, at the cost of a modest performance overhead.

7 A Discussion on Optimizations

The implementation shown in the previous sections is a first step toward a significant integration between inductive tasks and deductive tasks. As a prototype, however, it suffers several drawbacks:

- Scalability is the first issue. The $\mathcal{LDL}++$ system is mainly a main-memory system, so that whenever the blocks of memory that store relations are full, swapping strategies must be adopted in order to deal with larger tables. This obviously can influence the system performance considerably.
- A second issue is the optimization of the execution of user-defined predicates. As an example, it is well-known from the literature [5], [37] that the count phase is by far the most expensive in the Apriori Algorithm. Thus, the compiler has to provide an optimized execution plan, in order to ensure that the user-defined predicate implementing such a phase is called with the minimum impact over the overall performance [9].
- The most important problem is concerned with optimizing the query evaluation process. The application of a mining algorithm can be by itself a costly operation, and such a cost reflects on its implementation as an aggregate. The evaluation of more complex rules defined using aggregate predicates has to be studied in depth, since it is amenable to substantial optimizations [25], [34]. For example, if we consider the rule

$$\mathtt{p(S, patterns\langle(S, I)\rangle)} \leftarrow \mathtt{r(T, I)}.$$

where relation r has a very large extension, the evaluation of the query $\mathtt{p(S, R, C)}$ is expensive, no matter how efficiently we implement the aggregate. However, let us suppose that p is used to define the following predicate:

$$\mathtt{q(L, S_1)} \leftarrow \mathtt{p(0.001, \{L, R\}, S_1), S_1 > 0.5, s(R, \_)}.$$

There are many important considerations concerning the evaluation of q:

1. In a way similar to traditional *constant pushing* techniques, we can move the constraint $\mathtt{S_1} > 0.5$ in the evaluation of p since it is less expensive to evaluate $\mathtt{p(0.5, \{L, R\}, S_1)}$.
2. The predicate $\mathtt{p(0.5, \{L, R\}, S_1)}$ is amenable to further optimizations, since it requires only itemsets of size 2. This corresponds to a better tuning of the compiler, that should be able to recognize such situations and rewrite the rule defining p into simpler, more efficient rules:

$$\begin{aligned}
&\mathtt{tmp_p_1(V, count\langle T\rangle)} && \leftarrow \mathtt{r(T, I), member(V, I)}.\\
&\mathtt{tmp_p_2(S, L, R, count\langle T\rangle)} && \leftarrow \mathtt{tmp_p_1(L, C_1), tmp_p_1(R, C_2),}\\
&&& \quad \mathtt{C_1 > S, C_2 > S, r(T, I),}\\
&&& \quad \mathtt{member(R, I), member(L, I)}.\\
&\mathtt{p(S, \{L, R\}, S_1)} && \leftarrow \mathtt{tmp_p_2(S, L, R, S_1), S_1 > S}.
\end{aligned}$$

3. The join between p and s can be moved in the evaluation of p, allowing a further performance gain. For example, the second rule can be rewritten as

$$\begin{aligned}
\mathtt{tmp_p_2(S, L, R, count\langle T\rangle)} \leftarrow\ & \mathtt{tmp_p_1(L, C_1), tmp_p_1(R, C_2),}\\
& \mathtt{s(R, _), C_1 > S, C_2 > S, r(T, I),}\\
& \mathtt{member(R, I), member(L, I)}.
\end{aligned}$$

To summarize, the initial program

$$\begin{aligned}
&\mathtt{p(S, patterns\langle(S, I)\rangle)} \leftarrow \mathtt{r(T, I)}.\\
&\mathtt{q(L, S_1)} \leftarrow \mathtt{p(0.001, \{L, R\}, S_1), S_1 > 0.5, s(R, _)}.
\end{aligned}$$

can be rewritten, in the compilation phase, into the more efficient program

$$\begin{aligned}
&\mathtt{tmp_p_1(V, count\langle T\rangle)} && \leftarrow \mathtt{r(T, I), member(V, I)}.\\
&\mathtt{tmp_p_2(S, L, R, count\langle T\rangle)} && \leftarrow \mathtt{tmp_p_1(L, C_1), tmp_p_1(R, C_2),}\\
&&& \quad \mathtt{s(R, _), C_1 > S, C_2 > S, r(T, I),}\\
&&& \quad \mathtt{member(R, I), member(L, I)}.\\
&\mathtt{p(S, \{L, R\}, S_1)} && \leftarrow \mathtt{tmp_p_2(S, L, R, S_1), S_1 > S}.\\
&\mathtt{q(L, S_1)} && \leftarrow \mathtt{p(0.5, \{L, R\}, S_1)}.
\end{aligned}$$

that is amenable to further traditional deductive databases optimizations, such as the ones described in [40], [45], [1], [17].

The above example shows that a thorough modification of the underlying logic abstract machine needs to be investigated. Other interesting ways of coping with efficiency can be investigated as well. One way, for example, is to identify a subset of relevant features that can be transferred into more specialized languages, that allow more efficient implementations [28], [29].

In this respect, it is interesting to notice that similar approaches to the formalization of user-defined aggregates in relational query languages allow the direct specification of data mining tasks in SQL. For example, the AXL approach [42] and its extention ATLaS [41] has many similarities with the approach proposed in this paper (and indeed AXL is a successful attempt to introduce user-defined aggregates in SQL, by adopting both procedural and declarative data manipulation).

There are some significant differences between the approach of this paper and the ATLaS approach. For example, the possibility of expressing recursive aggregates is very powerful in ATLaS. Many algorithms are easy to specify without exploiting external user-defined predicates. From the opposite side, however, the use of procedural features, such as updating an internal table or checking for

the status of the interpreter (i.e., keeping track of the executions that are actually computed) gives raise to too complex formalizations, while instead the adoption of external predicates, with a clear meaning and a clear input/output interface, still maintains a simple semantics. Moreover, the capability to express complex queries by means of sets of clauses, as well as negation and recursion, make an appraoch based on datalog++ more suitable to model complex applications requiring interactions between induction and background knowledge. To this purpose, many other solutions aimed at specifying and evaluating mining constraints using a deductive database language are currently under investigation [12], [13], [20]. Notwithstanding, the ATLaS approach shows that specialized languages for mining/olap tasks could benefit of even a subset of the features of a logic database language, easy to implement in an efficient way.

8 Conclusions

In this paper, we described some aspects of the development of a logic database language with data mining mechanisms to model extraction, representation and utilization of both induced and deduced knowledge. The whole development consists in three main steps.

- The datalog++ [17] logic database language (and its implementation $\mathcal{LDL}++$) has a formal iterated fixpoint semantics that makes the language viable for efficient implementation. Its formal foundation accomodates temporal, nonmonotonic, and nondeterministic reasoning. Such distinguishing features are amenable to model complex applications.
- In particular, datalog++ allows the defining of iterative user-defined aggregates. Aggregates provide a standard interface for the specification of data mining tasks in the deductive environment, i.e., they allow the modeling of mining tasks as operations unveiling hidden knowledge. We used such features to model a set of data mining primitives. Frequent pattern discovery is an example such primitive.
- We studied an in-depth implementation of the above features, in order to provide an efficient runtime support for the resulting language. We provided an implementation of the above aggregates by modeling the most computationally intensive phases by means of hot-spot refinements, i.e., specialized data structures and user-defined predicates, which efficiently accomplish such phases. In this paper, we used the discovery of frequent patterns as an example of this process. As a result, we compared the effectiveness of the approach with respect to standard approaches in literature, demonstrating that the underlying system provides a viable trade off between expressiveness and efficiency.

In summary, we believe that the contributions of our work are the proof that deductive databases plus iterative user-defined aggregates realize the idea of implementing the knowledge discovery process as a query process. Furthermore, we found that the hot-spot refinement technique provides enough performance of the algorithms to make very convenient the balance between expressiveness of the language and loss of efficiency with respect to ad hoc and inflexible solutions.

There are some issues that are worth further investigations. In particular, an in-depth study of how to provide high-level optimization by means of direct exploitation of background knowledge has to be performed. Furthermore, tailoring of optimization techniques such as pushing constraints into mining queries (as shown in Section 7) is a major research topic in the context of data mining query languages. Clearly, the study of optimization techniques is even more substantial in logic-based languages.

Acknowledgments

The authors would like to thank Carlo Zaniolo for useful comments and discussions and Hiaxun Wang for his support with the $\mathcal{LDL}++$ system. This paper is a revised, extended version of the paper appeared as [19].

References

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases.* Addison-Wesley, 1995.
[2] P. Alcamo, F. Domenichini, and F. Turini, "An XML Based Environment in Support of the Overall KDD Process," *Proc. Fourth Int'l Conf. Flexible Query Answering Systems,* pp. 413-424, 2000.
[3] R. Agrawal and K. Shim, "Developing Tightly-Coupled Data Mining Applications on a Relational Database System," *Proc. Second Int'l Conf. Knowledge Discovery and Data Mining,* pp. 287-290, 1996.
[4] R. Agrawal, S. Sarawagi, and S. Thomas, "Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications," *Data Mining and Knowledge Discovery,* vol. 4, nos. 2/3, pp. 89-125, 2000.
[5] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 20th Int'l Conf. Very Large Databases,* pp. 487-499, 1994.
[6] R. Bayardo, "Efficiently Mining Long Patterns from Databases," *Proc. ACM Conf. Management of Data,* pp. 85-93, 1998.
[7] J.-F. Boulicaut, M. Klemettinen, and H. Mannila, "Querying Inductive Databases: A Case Study on the MINE RULE Operator," *Proc. Second European Conf. Principles and Practice Knowledge Discovery in Databases,* pp. 194-202, 1998.
[8] D. Bourdick, M. Calimlin, and J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," *Proc. 17th Int'l Conf. Data Eng.,* pp. 443-452, 2001.
[9] S. Chaudhuri and K. Shim, "Optmization of Queries with User-Defined Predicates," *ACM Trans. Database Systems,* vol. 24, no. 2, pp. 177-228, 1999.
[10] M.S. Chen, J. Han, and P.S. Yu, "Data Mining: An Overview from a Database Perspective," *IEEE Trans. Knowledge and Data Eng.,* vol. 8, no. 6, pp. 866-883, Dec. 1996.
[11] D. Chimenti, R. Gamboa, and R. Krishnamurthy, "Towards an Open Architecture for $\mathcal{LDL}$," *Proc. 15th Int'l Conf. Very Large Data Bases,* pp. 195-204, 1989.
[12] L. De Raedt, "Data Mining as Constraint Logic Programming," *Computational Logic: Logic Programming and Beyond,* pp. 526-547, 2002.
[13] L. De Raedt, "A Logical Database Mining Language," *Proc. 10th Int'l Conf. Inductive Logic Programming,* pp. 78-92, 2000.
[14] F. Giannotti and G. Manco, "Making Knowledge Extraction and Reasoning Closer," *Proc. Fourth Pacific-Asia Conf. Knowledge Discovery and Data Mining,* pp. 360-371, 2000.
[15] F. Giannotti and G. Manco, "Declarative Knowledge Extraction with Iterative User-Defined Aggregates," *Proc. Fourth Int'l Conf. Flexible Query Answering Systems,* pp. 445-454, 2000.
[16] F. Giannotti and G. Manco, "Querying Inductive Databases via Logic-Based User-Defined Aggregates," *Proc. Third European Conf. Principles and Practices of Knowledge Discovery in Databases,* pp. 125-135, 1999.

[17] F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi, "Nondeterministic, Nonmonotonic Logic Databases," *IEEE Trans. Knowledge and Data Eng.*, vol. 13, no. 5, pp. 813-823, Sept./Oct. 2001.
[18] F. Giannotti, G. Manco, D. Pedreschi, and F. Turini, "Experiences with a Logic-Based Knowledge Discovery Support Environment," *Selected Papers of the Sixth Italian Congress of Artificial Intellingence,* pp. 202-213, 2000.
[19] F. Giannotti, G. Manco, and F. Turini, "Specifying Mining Algorithms with Iterative User-Defined Aggregates: A Case Study," *Proc. Fifth European Conf. Principles and Practice of Knowledge Discovery in Databases,* pp. 128-139, 2001.
[20] F. Giannotti, G. Manco, and J. Wijsen, "Logical Languages for Data Mining," *Logics for Emerging Applications of Databases,* J. Chomicki et al., eds., pp. 325-361, 2003.
[21] F. Giannotti, D. Pedreschi, and C. Zaniolo, "Semantics and Expressive Power of Non Deterministic Constructs for Deductive Databases," *J. Computer and Systems Sciences,* vol. 62, no. 1, pp. 15-42, 2001.
[22] G. Graefe, U. Fayyad, and S. Chaudhuri, "On the Efficient Gathering of Sufficient Statistics for Classification from Large SQL Databases," *Proc. Fourth Int'l Conf. Knowledge Discovery and Data Mining,* pp. 204-208, 1998.
[23] J. Han, Y. Fu, K. Koperski, W. Wang, and O. Zaiane, "DMQL: A Data Mining Query Language for Relational Databases," *Proc. ACM SIGMOD Workshop Research Issues on Data Mining and Knowledge Discovery,* 1996.
[24] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *Proc. ACM Conf. Management of Data,* pp. 1-12, 2000.
[25] J. Hellerstein, "Optimization Techniques for Queries with Expensive Methods," *ACM Trans. Database Systems,* vol. 23, no. 2, pp. 113-157, 1998.
[26] T. Imielinski and H. Mannila, "A Database Perspective on Knowledge Discovery," *Comm. ACM,* vol. 39, no. 11, pp. 58-64, 1996.
[27] T. Imielinski and A. Virmani, "MSQL: A Query Language for Database Mining," *Data Mining and Knowledge Discovery,* vol. 3, no. 4, pp. 373-408, 1999.
[28] T. Johnson, L. Lakshmanan, and R.T. Ng, "The 3W Model and Algebra for Unified Data Mining," *Proc. 26th Int'l Conf. Very Large Data Bases,* pp. 21-32, 2000.
[29] L. Lakshmanan, F. Sadri, and S.N. Subramanian, "On Efficiently Implementing `schemasql` on a SQL Database," *Proc. 25th Int'l Conf. Very Large Data Bases,* pp. 471-482, 1999.
[30] G. Manco, "Foundations of a Logic-Based Framework for Intelligent Data Analysis," PhD thesis, Dept. of Computer Science, Univ. of Pisa, Apr. 2001.
[31] H. Mannila, "Inductive Databases and Condensed Representations for Data Mining," *Proc. Int'l Logic Programming Symp.,* pp. 21-30, 1997.
[32] R. Meo, G. Psaila, and S. Ceri, "A New SQL-Like Operator for Mining Association Rules," *Proc. 22th Int'l Conf. Very Large Databases,* pp. 122-133, 1996.
[33] R. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang, "Exploratory Mining and Pruning Optimizations of Constrained Associations Rules," *Proc. ACM Conf. Management of Data,* pp. 13-24, 1998.
[34] S. Nestorov and S. Tsur, "Integrating Data Mining with Relational DBMS: A Tightly-Coupled Approach," *Proc. Fourth Int'l Workshop Next Generation Information Technologies and Systems,* pp. 295-311, 1999.
[35] J.S. Park, M.-S. Chen, and P.S. Yu, "An Effective Hash-Based Algorithm for Mining Association Rules," *Proc. ACM Conf. Management of Data,* pp. 175-187, 1997.
[36] A. Savasere, E. Omiecinski, and S.B. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases," *Proc. 21th Int'l Conf. Very Large Databases,* pp. 432-444, 1995.
[37] R. Srikant, "Fast Algorithms for Mining Association Rules and Sequential Patterns," PhD thesis, Univ. of Wisconsin-Madison, 1996.
[38] H. Toivonen, "Sampling Large Databases for Association Rules," *Proc. 22th Int'l Conf. Very Large Databases,* pp. 134-145, 1996.
[39] D. Tsur et al., "Query Flocks: A Generalization of Association-Rule Mining," *Proc. ACM Conf. Management of Data,* pp. 1-12, 1998.
[40] J. Ullman, *Principles of Database and Knowledge-Base Systems,* vols. I-II, New York: Computer Science Press, 1989.
[41] H. Wang and C. Zaniolo, "ATLaS: A Native Extension of SQL for Data Mining," *Proc. Third SIAM Conf. Data Mining,* 2003.
[42] H. Wang and C. Zaniolo, "Using SQL to Build New Aggregates and Extenders for Object-Relational Systems," *Proc. 26th Int'l Conf. Very Large Data Bases,* pp. 166-175, 2000.
[43] M. Zaki and C. Hsiao, "CHARM: An Efficient Algorithm for Closed Itemset Mining," *Proc. Second SIAM Int'l Conf. Data Mining,* 2002.
[44] C. Zaniolo, N. Arni, and K. Ong, "Negation and Aggregates in Recursive Rules: The $\mathcal{LDL}++$ Approach," *Proc. Third Int'l Conf. Deductive and Object-Oriented Databases,* pp. 204-221, 1993.
[45] C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, and R. Zicari, *Advanced Database Systems.* Morgan Kaufman, 1997.
[46] C. Zaniolo and H. Wang, "Logic-Based User-Defined Aggregates for the Next Generation of Database Systems," *The Logic Programming Paradigm: Current Trends and Future Directions,* K.R. Apt et al., eds., Springer Verlag, 1998.

Fosca Giannotti received a degree in computer science (Laurea in Scienze dell'Informazione) summa cum laude, in 1982, from the University of Pisa. She is currently a senior researcher at Institute of Information Science and Technologies (ISTI) of the Italian National Research Council in Pisa where she leads the Knowledge Discovery and Delivery Laboratory. She has been a researcher at the R&D labs of several Italian software companies, and a visiting scientist at the Microelectronics and Computer Technology Corporation (MCC), Austin, Texas, involved in the LDL (Logic Database Language) project. Her current research interests include data mining query languages, data and Web mining, spatio-temporal reasoning, and mining. She has taught classes on databases and data mining at universities in Italy and abroad and has coordinated a master course on data mining in collaboration with the faculty of Economics at the University of Pisa. She served in the organization and in the scientific committee of various conferences in the area of logic programming, databases, and data mining.

Giuseppe Manco received the degree in computer science (Laurea in Scienze dell'Informazione) summa cum laude, in 1994, and the PhD degree in computer science from the University of Pisa. He is currently a senior researcher at the Institute of High Performance Computing and Networks (ICAR-CNR) of the National Research Council of Italy, and a contract professor at University of Calabria, Italy. He has been a contract researcher at the CNUCE Institute in Pisa, and a visiting fellow at the CWI Institute in Amsterdam, the Netherlands. His current research interests include deductive databases; knowledge discovery and data mining; Web databases and semistructured data management. He has taught classes on databases and data mining at universities in Italy and is currently involved in the scientific committee of various conferences and workshops on databases and data mining.

Franco Turini received the degree in computer science (Laurea in Scienze dell' Informazione) summa cum laude, in 1973, from the University of Pisa. He is a full professor in the Department of Computer Science at the University of Pisa. In 1978/1980, he was a visiting scientist at Carnegie-Mellon University (Pittsburgh) and of the IBM Research Center in San Jose, California, afterwards. In 1992/1993, he was visiting professor at the University of Utah. He is a member of the IEEE and ACM. His research interests include design, implementation, and formal semantics of programming languages, logic programming and deductive databases, and knowledge discovery. He has been coordinator of several national and international research projects. He served in the organization and in the scientific committee of various conferences in the area of logic programming, artificial intelligence, and data mining.