Specifying Mining Algorithms with Iterative User-Defined Aggregates: A Case Study

Fosca Giannotti[1], Giuseppe Manco[1], and Franco Turini[2]

[1] CNUCE-CNR - Via Alfieri 1 - 56010 Pisa Italy
{F.Giannotti,G.Manco}@cnuce.cnr.it
[2] Department of Computer Science - Corso Italia 40 - 56125 Pisa Italy
turini@DI.Unipi.IT

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 deductive rules, and data mining algorithms are constructed by means of iterative user-defined aggregates. Iterative user-defined aggregates have a fixed scheme that allows the modularization of data mining algorithms, thus providing a way to exploit domain knowledge in the right point. As a case study, the paper presents user-defined aggregates for specifying a version of the apriori algorithm. Some performance analyses and comparisons are discussed in order to show the effectiveness of the approach.

1 Introduction and Motivations

The problem of incorporating data mining technology into query systems has been widely studied in the current literature [12,9,14,10]. In such a context, the idea of integrating data mining algorithms in a deductive environment [7,5] is very powerful, since it allows the direct exploitation of domain knowledge within the specification of the queries, the specification of ad-hoc interest measures that can help in evaluating the extracted knowledge, and the modelization of the interactive and iterative features of knowledge discovery in a uniform way. However, the main drawback of a deductive approach to data mining query languages concerns efficiency: a data mining algorithm can be worth substantial optimizations that come both from a smart constraining of the search space, and from the exploitation of efficient data structures. The case of association rules is a typical example of this. Association rules are computed from frequent itemsets, that actually can be efficiently computed by exploiting the *apriori property* [15], and by speeding-up comparisons and counting operations with the adoption of special data structures (e.g., lookup tables, hash trees, etc.). Detailed studies [1] have shown that a direct specification of the algorithms within a query language lacks of performance effectiveness.

A partial solution to this problem has been proposed in [5,14,2]. In these approaches, data mining algorithms are modeled as "black boxes" integrated within the system. 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 the tuning of the search on the basis of specific properties of the problem at hand. As an example, using black boxes we cannot directly exploit domain knowledge within the algorithm, nor we can "on-the-fly" evaluate interest measures of the discovered patterns.

The above considerations yield an apparent mismatch: it is unfeasible to specify directly and implement data mining algorithms using the query language itself, and by the converse it is inconvenient to integrate data mining algorithms within query languages as predefined modules. In this paper we propose to combine the advantages of the two approaches in a uniform way. Following the approach of [7,5], we adopt aggregates as an interface to mining tasks in a deductive database. Moreover, data mining algorithms are specified by means of *iterative* user-defined aggregates, i.e., aggregates that are computed using a fixed scheme. Such a feature allows to modularize data mining algorithm and integrate domain knowledge in the right points, thus allowing crucial domain-oriented optimizations.

On the other side, user-defined predicates can be implemented by means of *hot-spot refinements* [3,4]. That is, we can extend the deductive databases with new data types, (like in the case of object-relational data systems), that can be efficiently accessed and managed using ad-hoc methods. Such data types and methods can be implemented by the user-defined predicates, possibly in other programming languages, with a reasonable trade-off between specification and efficient implementation.

The main advantages of such an approach are twofold:

- on the one side, we maintain an adequate declarative approach to the specification of data mining algorithms.
- on the other side, we can exploit specific (physical) optimizations improving the performance of the algorithms exactly where they are needed.

As a case study, the paper presents how such a technique can be used to specify a version of the apriori algorithm, capable of taking into account domain knowledge in the pruning phase. We recall the `patterns` aggregate defined in [5], and provide a specification of such an aggregate as an iterative user-defined aggregate. Hence, we provide an effective implementation of the aggregate by exploiting user-defined predicates.

The paper is organized as follows. Section 2 introduces the notion of iterative user-defined aggregates, and justifies their use in the specification of data mining algorithms. In Sect. 3 we introduce the `patterns` iterative aggregate for mining frequent itemsets. In particular, we show how user-defined predicates can be exploited to efficiently implement the aggregate. Finally, in Sect. 4 some performance analyses and comparisons are discussed in order to show the effectiveness of the approach.

2 Iterative User-Defined Aggregates

In [11] we formalize the notion of logic-based knowledge discovery support environment, as a deductive database programming language that models inductive rules as well as deductive rules. Here, an inductive rule provides a smooth integration of data mining and querying. In [5,7] we propose the modeling of an inductive rule by means of *aggregate functions.* The capability of specifying (and efficiently computing) aggregates is very important in order to provide a basis of a logic-based knowledge discovery support environment. To this purpose, the Datalog++ logic-based database language [16,17,8] provides a general framework for dealing with user-defined aggregates. We use such aggregates as the means to introduce mining primitives into the query language.

In general, a *user-defined aggregate* [17] is defined as a *distributive* aggregate, i.e., a function f inductively defined over a (nondeterministically sorted) set S:

$$f(\{x\}) = g(x) \tag{1}$$

$$f(S \cup \{x\}) = h(f(S), x) \tag{2}$$

We can directly specify the base and inductive cases, by means of ad-hoc user-defined predicates `single`, `multi` and `return`, used implicitly in the evaluation of the aggregate rule

$$\mathtt{p}(\mathtt{K}_1, \ldots, \mathtt{K}_\mathtt{m}, \mathtt{aggr}\langle\mathtt{X}\rangle) \leftarrow \mathtt{Rule\ body}.$$

In particular, $\mathtt{single}(\mathtt{aggr}, \mathtt{X}, \mathtt{C})$ associates to the first tuple `X` in the nondeterministic ordering a value, according to (1), and $\mathtt{multi}(\mathtt{aggr}, \mathtt{Old}, \mathtt{X}, \mathtt{New})$ computes the value of the aggregate `aggr` associated to the current value `X` in the current ordering, by incrementally computing it from the previous value, according to (2).

However, in order to define complex aggregation functions, (such as mining functions), the main problem with the traditional user-defined aggregate model is the impossibility of defining more complex forms of aggregates than distributive ones. In many cases, even simple aggregates may require multiple steps over data in order to be computed. A simple way [6] of coping with the problem of multiple scans over data can be done by extending the specification of the aggregation rule, in order to impose some user-defined conditions for iterating the scan over data. The main scheme shown in [17] requires that the evaluation of the query $\mathtt{p}(\mathtt{v}_1, \ldots, \mathtt{v}_\mathtt{m}, \mathtt{v})$ is done by first compiling the above program, and then evaluating the query on the compiled program. In the compiling phase, the program is rewritten into an equivalent, fixed rule scheme, that depends upon the user-defined predicates `single`, `multi` and `return`.

In [6,11] we slightly modify such rewriting, by making the scheme dependent upon the new `iterate` user-defined predicate. Such predicate specifies the condition for iterating the aggregate computation: the activation (and evaluation) of such a rule 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.

Example 1. The computation of the absolute deviation $S_n = \sum_x |\overline{x} - x|$ of a set of n elements needs at least two scans over the data. Exploiting `iterate` predicate, we can define S_n as a user-defined predicate:

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

The combined use of `multi` and `iterate` allows 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. ◁

Although the notion of iterative aggregate is in some sense orthogonal to the envisaged notion of inductive rule [11], the main motivation for introducing iterative aggregates is that the iterative schema shown above is common in many data mining algorithms. Usually, a typical data mining algorithm is an instance of an iterative schema where, at each iteration, some statistics are gathered from the data. The termination condition can be used to determine whether the extracted statistics are sufficient to the purpose of the task (i.e., they determine all the patterns), or whether no further statistics can be extracted.

Hence, the iterative schema discussed so far is a good candidate for specifying steps of data mining algorithms at low granularity levels. Relating aggregate specification with inductive rules makes it easy to provide an interface capable of specifying source data, knowledge extraction, background knowledge and interestingness measures. Moreover, we can specify the data mining task under consideration in detail, by exploiting ad-hoc definitions of `single`, `multi`, `iterate` and `return` iterative user-defined predicates.

3 The `patterns` Iterative Aggregate

In the following, we concentrate on the problem of mining frequent patterns from a dataset of transactions. We can integrate such mining task within the datalog++ database language, by means of a suitable inductive rule.

Definition 1 ([5]). *Given a relation* `r`*, the* `patterns` *aggregate is defined by the rule*

$$\texttt{p}(\texttt{X}_1,\ldots,\texttt{X}_\texttt{n},\texttt{patterns}\langle(\texttt{min\_supp},\texttt{Y})\rangle) \leftarrow \texttt{r}(\texttt{Z}_1,\ldots,\texttt{Z}_\texttt{m}) \qquad (3)$$

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

1. $\mathtt{t_1,\ldots,t_n}$ are distinct instances of the variables $\mathtt{X_1,\ldots,X_n}$, as resulting from the evaluation of `r`;
2. $\mathtt{s} = \{\mathtt{l_1,\ldots,l_k}\}$ 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 $f \geq \mathit{min\_supp}$.

$\square$

We can provide an explicit specification of the `patterns` aggregate in the above definition as an iterative aggregate. That is, we can directly implement an algorithm for computing frequent patterns, by defining the predicates `single`, `multi`, `return` and `iterate`.

The simplest specification adopts the purely declarative approach of *generating* all the possible itemsets, and then *testing* the frequency of the itemsets. It is easy to provide such a naive definition by means of the iterative schema proposed in Sect. 2:

$$
\begin{array}{ll}
\mathtt{single(patterns,(Sp,S),((Sp,1),IS))} & \leftarrow \mathtt{subset(IS,S)}.\\[6pt]
\mathtt{multi(patterns,((Sp,N),\_),(Sp,S),((Sp,N+1),IS))} & \leftarrow \mathtt{subset(IS,S)}.\\
\mathtt{multi(patterns,((Sp,N),IS),\_,(Sp,IS))}. & \\[6pt]
\mathtt{multi(patterns,(Sp,IS,N),(\_,S),(Sp,IS,N+1))} & \leftarrow \mathtt{subset(IS,S)}.\\
\mathtt{multi(patterns,(Sp,IS,N),(\_,S),(Sp,IS,N))} & \leftarrow \neg\mathtt{subset(IS,S)}.\\[6pt]
\mathtt{iterate(patterns,((Sp,N),IS),(Sp\times N,IS,0))}. & \\[6pt]
\mathtt{freturn(patterns,(Sp,IS,N),(IS,N))} & \leftarrow \mathtt{N} \geq \mathtt{Sp}.
\end{array}
$$

Such a specification works with two main iterations. In the first iteration (first three rules), the set of possible subsets are generated for each tuple in the dataset. The `iterate` predicate initializes the counter of each candidate itemset, and activates the computation of its frequency (performed by the remaining `multi` rules). The computation terminates when all itemsets frequencies have been computed, and frequent itemsets are returned as answers (by mean of the `freturn` rule). Notice that the `freturn` predicate defines the output format for the aggregation predicate: a suitable answer is a pair $(\mathtt{Itemset},\mathtt{N})$ such that `Itemset` is an itemset of frequency $\mathtt{N} > \mathtt{Sp}$, where `Sp` is the minimal support required.

Clearly, the above implementation is extremely inefficient, since it checks the support of all the possible itemsets. More precisely, 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. As a consequence, no pruning strategy is exploited; namely, unfrequent subsets are discarded at the end of the computation of the frequencies of all the subsets. Moreover, no optimized data structure, capable of speeding-up the computation of some costly operations, is used.

A detailed analysis of the *Apriori* algorithm [15] shown in Fig. 1, however, suggests a smarter specification. Initially, the algorithm computes the candidate itemsets of size 1 (*init phase*: step 1). The core of the algorithm is then a loop,

```
Algorithm Apriori($\mathcal{B}$, $\sigma$);

Input: a set of transactions $\mathcal{B}$, a support threshold $\sigma$;
Output: a set Result of frequent itemsets
Method: let initially $Result = \emptyset$, $k = 1$.
 1. $C_1 = \{a|a \in \mathcal{I}\}$;
 2. while $C_k \neq \emptyset$ do
 3.     foreach itemset $c \in C_k$ do
 4.         $supp(c) = 0$;
 5.     foreach $b \in \mathcal{B}$ do
 6.         foreach $c \in C_k$ such that $c \subseteq b$ do $supp(c)++$;
 7.     $L_k := \{c \in C_k |\ supp(c) > \sigma\}$;
 8.     $Result := Result \cup L_k$;
 9.     $C_{k+1} := \{c_i \cup c_j | c_i, c_j \in L_k \wedge |c_i \cup c_j| = k+1 \wedge \forall c \subset c_i \cup c_j$ such that $|c| = k : c \in L_k\}$;
10.     $k := k + 1$;
11. end while
```

Fig. 1. Apriori Algorithm for computing frequent itemsets

where the k-th iteration examines the set C_k of candidate itemsets of size k. During such an iteration the occurrences of each candidate itemset are computed scanning the data (*count phase*: steps 5-6). Unfrequent itemsets are then dropped (*prune phase*: step 7), and frequent ones are maintained in L_k. By exploiting the subset-frequency dependance, candidate itemsets of size $k+1$ can be built from pairs of frequent itemsets of size k differing only in one position (*enhance phase*: step 9). Finally, *Result* shall contain $\bigcup_k L_k$ (*itemsets phase*: step 8).

By exploiting iterative aggregates, we can directly specity all the phases of the algorithm. Initially, we specify the init phase,

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

The subsequent iterations resemble the steps of the apriori algorithm, that is counting the candidate itemsets, pruning unfrequent candidates and generating new candidates:

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

Such an approach exploits a substantial optimization, by avoiding to check a large portion of unfrequent itemsets. However, the implementation of the main operations of the algorithm is demanded to the predicates `singe_isets`, `prune`, `generate_candidates` and `count_isets`. As a consequence, the efficiency of the approach is parametric to the efficient implementation and evaluation of such predicates.

3.1 Exploiting User-Defined Predicates

In order to support complex database applications, most relational database systems support user-defined functions. Such functions can be invoked in 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* [4]. The user writes a large datalog++ 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 datalog++.

The $\mathcal{LDL}$++ [17,16] implementation of datalog++ 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 reveals very useful to provide new data types inside the $\mathcal{LDL}$++ model, in the style of Object-relational databases. For example, a reference to a C++ object can be returned as an answer, or passed as input, and the management of such a user-defined object is demanded to a set of external predicates.

We adopt such a model to implement hot-spot refinements of frequent itemsets mining. In the following we describe the implementation of an enhanced version of the Apriori algorithm, described in Fig. 1, by means user-defined predicates. In practice, we extend the allowed types of the $\mathcal{LDL}$++ system to include more complex structures, and provide some built-in predicates that efficiently manipulate such structures:

$$
\begin{array}{ll}
\mathtt{single(patterns,(Sp,S),((Sp,1),T))} & \leftarrow \mathtt{init(S,T)}.\\
\mathtt{multi(patterns,((Sp,N),T),(Sp,S),((Sp,N+1),T))} & \leftarrow \mathtt{init(S,T)}.\\
\mathtt{iterate(patterns,((Sp,N),T),(Sp \times N,T))} & \leftarrow \mathtt{prune(Sp,T),enhance(T)}.\\
\mathtt{multi(patterns,(Sp,T),(\_,S),(Sp,T))} & \leftarrow \mathtt{count(S,T)}.\\
\mathtt{iterate(patterns,(Sp,T),(Sp,T))} & \leftarrow \mathtt{prune(Sp,T),enhance(T)}.\\
\mathtt{freturn(patterns,(Sp,T),(I,S))} & \leftarrow \mathtt{itemset(T,(I,S))}.
\end{array}
$$

In such a schema, the variable `T` represents the reference to a structure of type *Hash-Tree* [15], which is essentially a prefix-tree with a hash table associated to each node. An edge is labelled with an item, so that paths from the root to an internal node represent itemsets. Figure 2 shows some example trees. Each node is labelled with a tag denoting the support of the itemset represented by the path from the root to the node. An additional tag denotes whether the node can generate new candidates. The predicates `init`, `count`, `enhance`, `prune` and `itemset` are *user-defined predicates* that implement, in C++, complex operators, exemplified in Fig. 2, over the given hash-tree. More specifically:

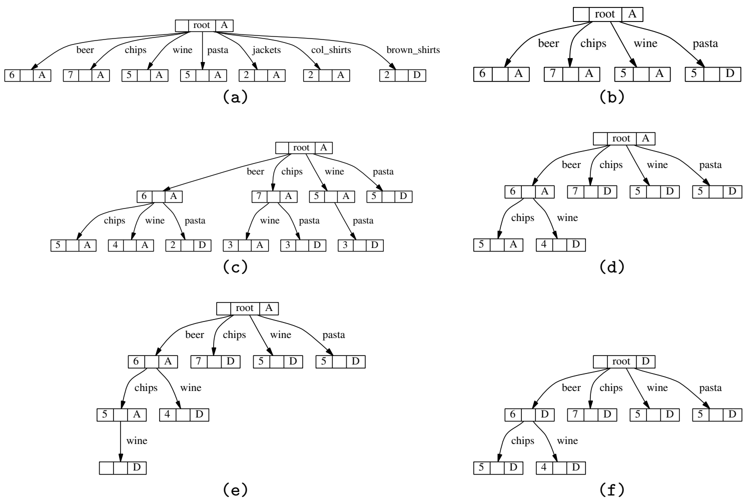

Fig. 2. a) Tree initialization. b) pruning. c) tree enhancement and counting. d) pruning. e) tree enhancement. f) cutting

- The `init(I, T)` predicate initializes and updates the frequencies of the 1-itemsets available from `I` in `T` (Fig. 2a). For each item found in the transaction `I`, either the item is already into 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. 2c). 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 level is found, the counter is incremented.
- The `prune(M, T)` predicate removes from `T` all the itemsets whose frequencies are less than `M` (Figs. 2b and d). Leaf 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)` predicates combines the frequent k-itemsets in `T` and generates the candidate $k+1$-itemsets. New candidates are generated in two step. In the first step, a leaf node is merged with each of its siblings, and new sons are generated. For example in Fig. 2e, the node labelled with `beer` – `chips` is merged with its sibling `beer` – `wine`, generating the new node labelled with `beer` – `chips` – `wine`. In order to ensure that every new node represents an actual candidate of size $n+1$, we need to check whether all the subsets of the itemset of size n are actually in the hash tree. Such an operation consists in a traversal of the tree from the enhanced node to the root node; for each

analyzed node, we simply check whether its subtree is also a subtree of its ancestor. Subtrees that do not satisfy such a requirement are cut (Fig. 2f).

– Finally, the `itemset(T,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 one answer for each itemset found.

The above schema provides a declarative specification that is parametric to the intended meaning of the user-defined predicates adopted. The schema minimalizes the "black-box" structure of the algorithm, needed to obtain fast counting of candidate itemsets and efficient pruning, and provides many opportunities of optimizing the execution of the algorithm both from a database optimization perspective and from a "constraints" embedding perspective [13].

4 Performance Analysis

In this section we analyze the impact of the architecture we described in the previous sections to the process of extracting association rules from data. The performance analysis that we undertook compared the effect of mining association rules according to four different architectural choices:

1. *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 Cache-Mine approach. The main motivation is to compare the effects of such an implementation with a similar one in the $\mathcal{LDL}$++ deductive database. 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 [14].
2. *DB2 interactive*, an Apriori implementation in which data is read tuple by tuple from the DBMS. This approach is very 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 such a context switching, it is interesting to see how the approach behaves compared to the $\mathcal{LDL}$++ approach.
3. *$\mathcal{LDL}$++*, the implementation of the rules mining aggregate `patterns`, by means of the iterative aggregate specified in the previous section.
4. a plain Apriori implementation (*Apriori* in the following), that reads data from a 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 present.

We tested the effect of a very simple form of mining query -one that is expressible also in SQL- that retrieves data from a single table and applies

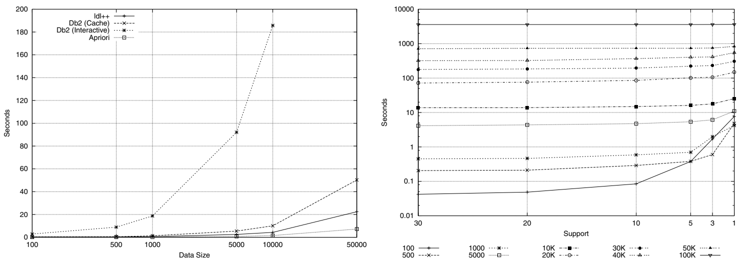

Fig. 3. Performance comparison and summary

the mining algorithm. In $\mathcal{LDL}$++ terms, the experiments were performed by querying $\texttt{ans(min\_supp, R)}$, where $\texttt{ans}$ was defined as

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

and the $\texttt{transaction(ID, ItemSet)}$ relation is a materialized table.

In order to populate the `transaction` predicate (and its relational counterpart), we used the synthetic data generation utility described in [15, Sect. 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.

The following figures show how the performances of the various solutions change 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 1 and 2 were implemented using the IBM DB2 universal database v6.1.

In Fig. 3 it can be seen that, as expected, the *DB2 (interactive)* solution gives the worst results: since a cursor is maintained against 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 [1]. 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.

Figure 3 shows that the *$\mathcal{LDL}$++* approach outperforms the *DB2 (Batch)* approach. However, as soon as the size of the dataset is increased, the difference between the two approaches tends to decrease: the graphs show that the *$\mathcal{LDL}$++* performance gradually worsens, and we can expect that, for larger datasets, *DB2 (Batch)* can outperform *$\mathcal{LDL}$++*. Such a behavior finds its explanation in the processing overhead of the deductive system with respect to the relational system, which can be quantified, as expected, by a constant factor.

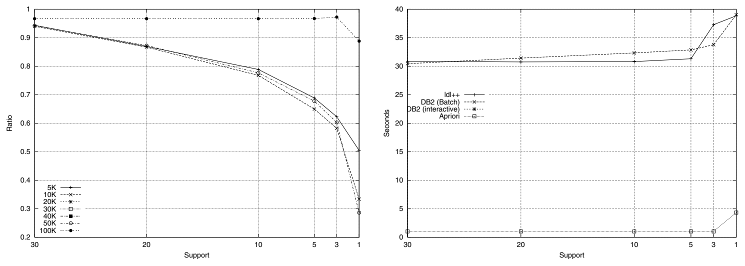

Fig. 4. Context switching overhead

The seconds graph in Fig. 3 summarizes the performance of the $\mathcal{LDL}$++ system for different values of the data size. The performance graph has a smooth (almost linear) curve. 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 Fig. 4. The ratio is 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 dataset, that does not contain frequent itemsets (except for very low support values), and hence the predominant computational cost is due to context switching.

5 Conclusions and Future Work

Iterative aggregates have the advantage of allowing the specification of data mining tasks at the desired abstraction level: from a conceptual point of view, they allow a direct use of background knowledge in the algorithm specification; from a physical point of view, they give the opportunity of directly integrating proper knowledge extraction optimizations. In this paper we have shown how the basic framework allows physical optimizations: an in-depth study of how to provide high-level optimization by means of direct exploitation of background knowledge has to be performed.

The problem of tailoring optimization techniques to mining queries is a major research topic, in a database-oriented approach to data mining. It is not surprising that such a topic is even more substantial in deductive-based approaches, like the one presented in this paper. In [11] we have shown some examples of how a logic based language can benefit of a thorough modification of the underlying abstract machine, and how other interesting ways of coping with efficiency can be investigated (for example, by extracting expressive subsets of teh language viable for efficient implementation).

We are currently interested in formalizing such modifications in order to provide a mapping of deductive mining query specifications to query plan generations and optimizations.

References

1. R. Agrawal, S. Sarawagi, and S. Thomas. Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications. *Data Mining and Knowledge Discovery*, 4(3):89–125, 2000.
2. P. Alcamo, F. Domenichini, and F. Turini. An XML Based Environment in Support of the Overall KDD Process. In *Procs. of the 4th International Conference on Flexible Query Answering Systems (FQAS2000)*, Advances in Soft Computing, pages 413–424, 2000.
3. S. Chaudhuri and K. Shim. Optimization of Queries with User-Defined Predicates. *ACM Transactions on Database Systems*, 24(2):177–228, 1999.
4. D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an Open Architecture for $\mathcal{LDL}$. In *Proc. 15th Int. Conf. on Very Large Data Bases (VLDB89)*, pages 195–204, 1989.
5. F. Giannotti and G. Manco. Querying Inductive Databases via Logic-Based User-Defined Aggregates. In *Proc. 3rd European Conference on Principles and Practices of Knowledge Discovery in Databases*, number 1704 in Lecture Notes on Artificial Intelligence, pages 125–135, September 1999.
6. F. Giannotti and G. Manco. Declarative Knowledge Extraction with Iterative User-Defined Aggregates. In *Procs. 4th International Conference on Flexible Query Answering Systems (FQAS2000)*, Advances in Soft Computing, pages 445–454, 2000.
7. F. Giannotti and G. Manco. Making Knowledge Extraction and Reasoning Closer. In *Proc. 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, number 1805 in Lecture Notes in Computer Science, April 2000.
8. F. Giannotti, G. Manco, M. Nanni, and D .Pedreschi. Nondeterministic, Nonmonotonic Logic Databases. *IEEE Trans. on Knowledge and Data Engineering.* To appear.
9. J. Han, Y. Fu, K. Koperski, W. Wang, and O. Zaiane. DMQL: A Data Mining Query Language for Relational Databases. In *SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'96)*, 1996.
10. T. Imielinski and A. Virmani. MSQL: A Query Language for Database Mining. *Data Mining and Knowledge Discovery*, 3(4):373–408, 1999.
11. G. Manco. *Foundations of a Logic-Based Framework for Intelligent Data Analysis.* PhD thesis, Department of Computer Science, University of Pisa, April 2001.
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 *Proc. 22th Int. Conf. on Very Large Data Bases (VLDB96)*, pages 122–133, 1996.
15. R. Srikant. *Fast Algorithms for Mining Association Rules and Sequential Patterns.* PhD thesis, University of Wisconsin-Madison, 1996.
16. 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.
17. 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, Berlin, 1998.