Noname manuscript No.
(will be inserted by the editor)

An Inductive Database System Based on Virtual Mining Views

Hendrik Blockeel · Toon Calders · Élisa Fromont · Bart Goethals · Adriana Prado · Céline Robardet

the date of receipt and acceptance should be inserted later

Abstract Inductive databases integrate database querying with database mining. In this article, we present an inductive database system in which the query language is SQL. In particular, we propose an intuitive and elegant framework based on virtual mining views, which are relational tables that virtually contain the complete output of data mining algorithms executed over a given dataset. We show that several types of patterns and models that are implicitly present in the data, such as itemsets, association rules, and decision trees, can be represented and queried with traditional SQL using a unifying framework. As a proof of concept, we illustrate a complete data mining scenario on real-word biological data with SQL queries over the mining views.

1 Introduction

Data mining is an interactive process in which different tasks may be performed sequentially. What is more, the output of those tasks may be combined

Hendrik Blockeel
Katholieke Universiteit Leuven, Belgium
Leiden Institute of Advanced Computer Science, Universiteit Leiden, The Netherlands
E-mail: hendrik.blockeel@cs.kuleuven.be

Toon Calders
Technische Universiteit Eindhoven, The Netherlands E-mail: t.calders@tue.nl

Élisa Fromont · Adriana Prado
Université de Lyon (Université Jean Monnet), CNRS, Laboratoire Hubert Curien, UMR5516, F-42023 Saint-Etienne, France E-mail: {elisa.fromont, adriana.bechara.prado}@univ-st-etienne.fr

Bart Goethals
Universiteit Antwerpen, Belgium E-mail: bart.goethals@ua.ac.be

Céline Robardet
Université de Lyon, INSA-Lyon, CNRS, LIRIS, UMR5205, F-69621, France E-mail: celine.robardet@insa-lyon.fr

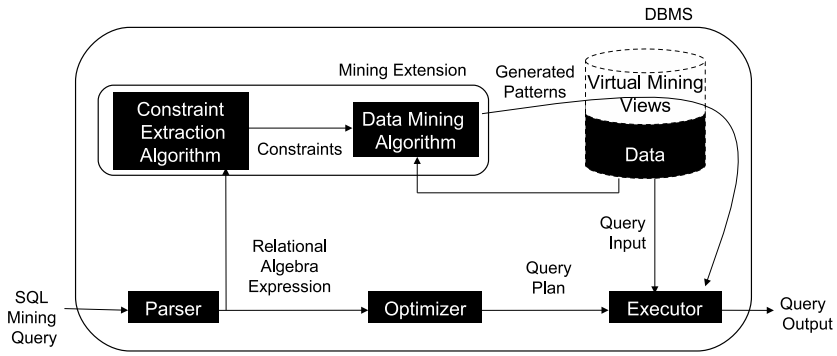


Fig. 1 An Inductive Database.

to be used as input for subsequent ones. As a result, in order to effectively support this knowledge discovery process, the integration of data mining into database management systems has become necessary. The concept of inductive databases has been proposed so as to achieve such integration [1].

In order to tackle the ambitious task of building an inductive database, one has to i) choose a query language that can be general enough to cover most of the data mining and machine learning toolkit while providing enough flexibility to the users in terms of constraints, ii) ensure a closure property to be able to reuse intermediate results, and iii) provide an intuitive way to interpret the results.

In this article, we describe how such an inductive database can be implemented in practice, as presented in [2–7]. Contrary to the numerous proposals of data mining query languages [8–16], we propose a relational database model based on the so-called *virtual mining views*. The mining views are relational tables that virtually contain the complete output of data mining tasks. E.g., for itemset mining, there is a table called *Sets* virtually storing all frequent patterns. In other words, as far as the user is concerned, all possible patterns are stored. On the physical layer, however, these tables are actually empty; whenever a query is formulated selecting, for instance, itemsets from these tables, the database system triggers a data mining algorithm (e.g., Apriori [17]), which computes the result of the query, in exactly the same way that normal views in databases are only computed at query time, and only to the extent necessary for answering the query.

The complete model is illustrated in Figure 1. A user can use the mining views in his or her query as if they were regular database tables. Given a query, the parser is then invoked by the database system, creating an equivalent relational algebra expression. At this point, the expression is processed by the *Mining Extension* which extracts from the query the constraints to be pushed into the data mining algorithms. The output of these algorithms is then materialized in the mining views. After the materialization, the work

flow of the system continues as usual and, as a result, the query is executed as if all patterns and models were always stored in the database.

Note that the system can potentially support as many virtual mining views as types of patterns of interest. To make the framework more general, however, such patterns should be represented by an intuitive common set of mining views. One possible instantiation of the proposed framework is presented in this article.

Besides the design of a general and intuitive framework, another problem that arises when creating such an inductive database concerns the types of constraints that can actually be extracted from the mining queries and how they can be efficiently exploited by the system. We also present in this article an algorithm to extract constraints from SQL queries over the mining views and discuss the mining views' materialization process itself.

All ideas presented in this article, from querying the mining views and extracting constraints from the queries to the actual execution of the data mining process itself and the materialization of the mining views, have been implemented into the well-known open source database system PostgreSQL[1]. The main steps of the implementation are also presented here.

The rest of this article is therefore organized as follows. The next section formally describes the mining views framework. In Section 3, we present the constraint extraction procedure and discuss the exploitation of such constraints in Section 4. The implementation of the system along with an extended illustrative data mining scenario is presented in Section 5. Section 6 discusses related work and we conclude in Section 7.

2 The Mining Views Framework

In this section, we present the mining views framework in detail. This framework consists of a set of relational tables, called mining views, which virtually represent the complete output of data mining tasks. In reality, the mining views are empty and the database system finds the required tuples only when they are queried by the user.

2.1 The Mining View Concepts

We assume to be working in a database that contains the table $T(A_1, \ldots, A_n)$, having only categorical attributes. The domain of A_i is denoted by $dom(A_i)$, for all $i = 1 \ldots n$. A tuple of T is therefore an element of $dom(A_i) \times \ldots \times dom(A_n)$. The active domain of A_i of T, denoted by $adom(A_i, T)$, is defined as the set of values that are currently assigned to A_i, that is, $adom(A_i, T) := \{t.A_i \mid t \in T\}$.

In the mining views framework, the patterns extracted from table T are generically represented by what we call *concepts*. We denote a concept as a

[1] `http://www.postgresl.org/`

conjunction of attribute-value pairs that is definable over table T. For example,

$$(Outlook = \text{'Sunny'} \wedge Humidity = \text{'High'} \wedge Play = \text{'No'})$$

is a concept defined over the classical relational data table Playtennis(Day, $Outlook$,$Temperature$, $Humidity$,$Wind$,$Play$) [18], which is illustrated in Figure 2.

Playtennis

Day	Outlook	Temperature	Humidity	Wind	Play
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Fig. 2 Data table Playtennis.

To represent each concept as a database tuple, we use the symbol '?' as the *wildcard value* and assume it does not exist in the active domain of any attribute of T.

Definition 1 A concept over table T is a tuple $(c_1, \ldots, c_n)$ with $c_i \in adom(A_i) \cup \{\text{'?'}\}$, for all i=1 ... n.

Following Definition 1, the concept above, which is defined over table Playtennis(Day,$Outlook$,$Temperature$,$Humidity$,$Wind$,$Play$), is represented by the tuple

$$(\text{'?'}, \text{'Sunny'}, \text{'?'}, \text{'?'}, \text{'High'}, \text{'?'}, \text{'No'}).$$

We are now ready to introduce the mining view $T\_Concepts$. In the proposed framework, the mining view $T\_Concepts(cid, A_1, \ldots, A_n)$ virtually contains all concepts that are definable over table T. We assume that these concepts can be sorted in lexicographic order and that an identifier can unambiguously be given to each concept.

Definition 2 The mining view $T\_Concepts(cid, A_1, \ldots, A_n)$ contains one tuple $(cid, c_1, \ldots, c_n)$ for every concept defined over table T. The attribute cid uniquely identifies the concepts.

Figure 3 shows a sample of the mining view *Playtennis_Concepts*, which virtually contains all concepts definable over table Playtennis. In fact, the mining view *T_Concepts* represents exactly a *data cube* [19] built from table T, with the difference that the wildcard value "ALL" introduced in [19] is replaced by the value '?'. By following the syntax introduced in [19], the mining view *T_Concepts* would be created with the SQL query shown in Figure 4 (consider adding the identifier *cid* after its creation).

Playtennis_Concepts

cid	Day	Outlook	Temperature	Humidity	Wind	Play
id1	?	?	?	?	?	?
id2	?	?	?	?	?	Yes
id3	?	?	?	?	Weak	Yes
id4	?	?	?	?	Strong	Yes
...	...	...	...	...	...	...
...	?	?	?	High	Weak	Yes
...	?	?	?	Normal	Weak	Yes
...	...	...	...	...	...	...
...	?	?	Cool	High	Weak	Yes
...	?	?	Mild	High	Weak	Yes
...	?	?	Hot	High	Weak	Yes
...	...	...	...	...	...	...
...	?	Sunny	Cool	High	Weak	Yes
...	?	Overcast	Cool	High	Weak	Yes
...	?	Rain	Cool	High	Weak	Yes
...	...	...	...	...	...	...
...	D1	Sunny	Cool	High	Weak	Yes
...	D2	Sunny	Cool	High	Weak	Yes
...	...	...	...	...	...	...

Fig. 3 The mining view *Playtennis_Concepts*.

```
1. create table T_Concepts
2. select A1, A2,..., An
3. from T
4. group by cube A1, A2,..., An
```

Fig. 4 The data cube that represents the contents of the mining view *T_Concepts*.

2.2 Representing Patterns and Models as Sets of Concepts

We now explain how patterns extracted from the table Playtennis can be represented by the concepts in the mining view *Playtennis_Concepts*. In the remainder of this section, we refer to table Playtennis as T and use the concepts in Figure 5 for the illustrative examples (an identifier has been given to each of the concepts in Figure 5).

Playtennis_Concepts

cid	Day	Outlook	Temperature	Humidity	Wind	Play
...	...	...	...	...	...	...
101	?	?	?	?	?	Yes
102	?	?	?	?	?	No
103	?	Sunny	?	High	?	?
104	?	Sunny	?	High	?	No
105	?	Sunny	?	Normal	?	Yes
106	?	Overcast	?	?	?	Yes
107	?	Rain	?	?	Strong	No
108	?	Rain	?	?	Weak	Yes
109	?	Rain	?	High	?	No
110	?	Rain	?	Normal	?	Yes
...	...	...	...	...	...	...

Fig. 5 A sample of the mining view *Playtennis_Concepts*, which is used for the illustrative examples in Section 2.2.

2.2.1 Itemsets and Association Rules

As itemsets in a relational database are conjunctions of attribute-value pairs, they are represented as concepts in the mining views framework. Itemsets are represented by the mining view *T_Sets*(*cid*, *supp*, *sz*), which is defined as follows:

Definition 3 The mining view *T_Sets*(*cid*, *supp*, *sz*) contains a tuple for each itemset, where *cid* is the identifier of the itemset (concept), *supp* is its support (the number of tuples satisfied by the concept), and *sz* is its size (the number of attribute-value pairs in which there are no wildcards).

Similarly, association rules are represented by the view *T_Rules*(*rid*, *cida*, *cidc*, *cid*, *conf*), described by the definition below:

Definition 4 The mining view *T_Rules*(*rid*,*cida*,*cidc*,*cid*,*conf*) contains a tuple for each association rule that can be extracted from table *T*. We assume that a unique identifier, *rid*, can be given to each rule. The attribute *rid* is the rule identifier, *cida* is the identifier of the concept representing its left hand side (referred to here as antecedent), *cidc* is the identifier of the concept representing its right hand side (referred to here as consequent), *cid* is the identifier of the union of those two concepts, and *conf* is the confidence of the rule.

Figure 6 shows the mining views *T_Sets* and *T_Rules*, and illustrates how the rule "if outlook is sunny and humidity is high, you should not play tennis" is represented in these views by using three of the concepts given in Figure 5. The rule has identification number 1.

Note that the choice of the schema for representing itemsets and association rules also implicitly determines the complexity of the queries a user needs to write. For example, one of the three concept identifiers for an association rule, *cid*, *cida*, or *cidc*, is redundant as it can be determined from the other two. Also, in the given representation one could even express the itemset mining

T_Sets

cid	supp	sz
102	5	1
103	3	2
104	3	3
...	...	...

T_Rules

rid	cida	cidc	cid	conf
1	103	102	104	100%
...	...	...	...	...

Fig. 6 Mining views for representing itemsets and association rules. The attributes *cida*, *cidc*, and *cid* refer to concepts given in Figure 5.

task without the view *T_Concepts*, as it can also be expressed in SQL. Nevertheless, it would imply that the user would have to write more complicated queries, as shown in Example 1.

Example 1 Consider the task of extracting from table T all itemsets (and their supports) with size equal to 5 and support of at least 3. Query (A) in Figure 7 shows how this task is performed in the proposed framework. Without the mining views *T_Concepts* nor *T_Sets*, this task would be executed with a much more complicated query, as given in query (B) in Figure 7.

Example 2 In Figure 8, query (C) is another example query over itemsets, while query (D) is an example query over association rules.

Query (C) asks for itemsets having support of at least 3, size of at most 5, and which contain the attribute-value pair"Outlook=Sunny". Query (D) asks for association rules having support of at least 3 and confidence of at least 80%.

Observe that common mining tasks and the constraints "minimum support" and "minimum confidence" can be expressed quite naturally with SQL queries over the mining views. Additionally, note that the mining views provide a very clear separation between the two mining operations, while at the same time allowing their composition, as association rules are built on top of frequent itemsets.

2.3 Decision Trees

A decision tree learner typically learns a single decision tree from a dataset. This setting strongly contrasts with discovery of itemsets and association rules, which is set-oriented: given certain constraints, the system finds all itemsets or association rules that fit the constraints. In decision tree learning, given a set of (sometimes implicit) constraints, one tries to find one tree that fulfills the constraints and, besides that, optimizes some other criteria, which are again not specified explicitly but are a consequence of the algorithm used.

In the inductive databases context, we treat decision tree learning in a somewhat different way, which is more in line with the set-oriented approach. Here, a user would typically write a query asking for all trees that fulfill a certain set of constraints, or optimizes a particular condition. For example,

(A)

```
select C.*, S.supp
from T_Sets S, T_Concepts C
where C.cid = S.cid
  and S.sz = 5
  and S.supp >= 3
```

(B)

```
select Day,Outlook,Temperature,Humidity,Wind,'?', count(*)
from T
group by Day,Outlook,Temperature,Humidity,Wind
having count(*) >= 3
union
select Day,Outlook,Temperature,Humidity,'?',Play, count(*)
from T
group by Day,Outlook,Temperature,Humidity, Play
having count(*) >= 3
union
select Day,Outlook,Temperature,'?',Wind,Play, count(*)
from T
group by Day,Outlook,Temperature,Wind,Play
having count(*) >= 3
union
select Day,Outlook,'?',Humidity,Wind,Play, count(*)
from T
group by Day,Outlook,Humidity,Wind,Play
having count(*) >= 3
union
select Day,'?',Temperature,Humidity,Wind,Play, count(*)
from T
group by Day,Temperature,Humidity,Wind,Play
having count(*) >= 3
union
select '?',Outlook,Temperature,Humidity,Wind,Play, count(*)
from T
group by Outlook,Temperature,Humidity,Wind,Play
having count(*) >= 3
```

Fig. 7 Example queries over itemsets with (query (A)) and without (query (B)) the mining views $T\_Concepts$ and T_Sets.

the user might ask for the tree with the highest training set accuracy among all trees of size of at most 5. This leads to a much more declarative way of mining for decision trees, which can easily be integrated into the mining views framework.

In a decision tree, each path from the root to a leaf node can be regarded as a conjunction of attribute-value pairs. Thus, a decision tree can be represented in our framework by a set of concepts, where each concept represents one path.

The collection of all decision trees predicting a particular target attribute A_i is therefore represented in our framework by the mining view $T\_Trees\_A_i$ ($treeid$,cid), which is defined as follows:

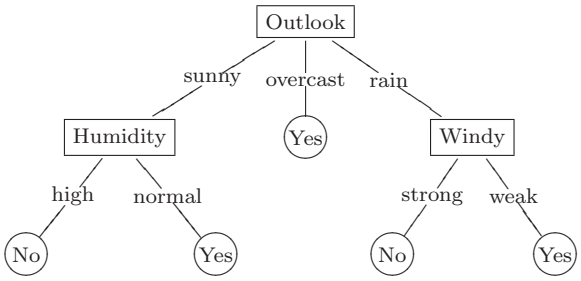
(C)

```
select C.*, S.supp, S.sz
from T_Concepts C, T_Sets S
where C.cid = S.cid
  and S.supp >= 3
  and S.sz <= 5
  and C.Outlook = 'Sunny'
```

(D)

```
select Ante.*, Cons.*,
       S.supp, R.conf
from T_Sets S, T_Rules R,
     T_Concepts Ante,
     T_Concepts Cons
where R.cid = S.cid
  and Ante.cid = R.cida
  and Cons.cid = R.cidc
  and S.supp >= 3
  and R.conf >= 80
```

Fig. 8 Example queries over itemsets and association rules.



T_Trees_Play

treeid	cid
1	104
1	105
1	106
1	107
1	108
...	...

T_Treescharac_Play

treeid	acc	sz
1	100%	8
...	...	...

Fig. 9 Mining views representing a decision tree which predicts the attribute *Play*. Each attribute *cid* of view *T_Trees_Play* refers to a concept given in Figure 5.

Definition 5 The mining view $T\_Trees\_A_i(treeid, cid)$ is such that, for every decision tree predicting a particular target attribute A_i, it contains as many tuples as the number of leaf nodes it has. We assume that a unique identifier, *treeid*, can be given to each decision tree. Each decision tree is represented by a set of concepts *cid*, where each concept represents one path from the root to a leaf node.

Additionally, a view representing several characteristics of a tree learned for one specific target attribute A_i is defined:

Definition 6 The mining view $T\_Treescharac\_A_i(treeid, acc, sz)$ contains a tuple for every decision tree in $T\_Trees\_A_i$, where *treeid* is the decision tree identifier, *acc* is its corresponding accuracy, and *sz* is its size in number of nodes.

Figure 9 shows how an example decision tree is represented in the mining views *T_Trees_Play* and *T_Treescharac_Play* by using the concepts in Figure 5. The example decision tree predicts the attribute *Play* of table *T* and has identification number 1.

Example 3 In Figure 10, we present two example queries over decision trees.

(E)

```
create table BestTrees as
select T.treeid, C.*, D.*
from T_Concepts C,
     T_Trees_Play T,
     T_Treescharac_Play D
where T.cid = C.cid
  and T.treeid = D.treeid
  and D.sz <= 5
  and D.acc =
     (select max(acc)
      from T_Treescharac_Play
      where sz <= 5)
```

(F)

```
select T1.treeid,
       C1.*, C2.*
from T_Trees_Play T1,
     T_Trees_Play T2,
     T_Concepts C1,
     T_Concepts C2,
     T_Treescharac_Play D
where T1.cid = C1.cid
  and T2.cid = C2.cid
  and T1.treeid = T2.treeid
  and T1.treeid = D.treeid
  and C1.Outlook = 'Sunny'
  and C2.Wind = 'Weak'
  and D.sz <= 5
  and D.acc >= 80
```

Fig. 10 Example queries over decision trees.

Query (E) creates a table called "BestTrees" with all decision trees that predict the attribute *Play* and have maximal accuracy among all possible decision trees of size of at most 5. Observe that in order to store the results back into the database, the user simply needs to use the statement "create table as", available in a variety of database systems that are based on SQL.

Query (F) asks for decision trees having an attribute test on "Outlook=Sunny" and on "Wind=Weak", with a size of at most 5 and an accuracy of at least 80%.

Prediction In order to classify a new tuple using a learned decision tree, one simply searches for the concept in this tree (path) that is satisfied by the new tuple. More generally, if we have a test set S, all predictions of the tuples in S are obtained by equi-joining S with the semantic representation of the decision tree given by its concepts. We join S to the concepts of the tree by using a variant of the equi-join that requires that either the values are equal, or there is a wildcard value.

Consider the table BestTrees created after the execution of query (E), in Figure 10. Figure 11 shows a query that predicts the attribute *Play* for all unclassified tuples in an example table Test_Set(Day,$Outlook$,$Temperature$, $Humidity$,$Wind$) by using the tree in table BestTrees that has identification number 1.

(G)

```
select S.*, T.Play
from Test_Set S,
     BestTrees T
where (S.Day = T.Day or T.Day = '?')
  and (S.Outlook = T.Outlook or T.Outlook  = '?')
  and (S.Temperature = T.Temperature or T.Temperature = '?')
  and (S.Humidity = T.Humidity or T.Humidity = '?')
  and (S.Wind = T.Wind or T.Wind = '?')
  and T.treeid = 1
```

Fig. 11 An example prediction query.

2.4 Combining Patterns and Models

In the mining views framework, it is also possible to perform composed data mining tasks. In other words, it is possible to formulate data mining tasks that consist of a combination of different types of patterns. For example, consider query (H) in Figure 12. The query asks for decision trees predicting the attribute *Play* with a size of at most 5, a path of which is an itemset that generates a rule with support of at least 3 and confidence of at least 80%. Notice that since in the proposed framework the query language is SQL, the user can create new combinations of patterns by simply involving mining views corresponding to different mining tasks in the same SQL query.

(H)

```
select T.*, C.*, S.supp
from T_Sets S, T_Rules R,
     T_Concepts C,
     T_Trees_Play T,
     T_Treescharac_Play D
where C.cid = S.cid
  and S.cid = R.cid
  and T.cid = R.cid
  and T.treeid = D.treeid
  and D.sz <= 5
  and S.supp >= 3
  and R.conf >= 80
```

Fig. 12 Example query combining patterns.

2.5 Putting It All Together

For every data table $T(A_1, \ldots, A_n)$ in the database, with T having only categorical attributes, the virtual mining views framework consists of a set of relational tables, called virtual mining views, which virtually contain the com-

plete output of data mining tasks executed over T. These mining views are the following:

- $T\_Concepts(cid, A_1, \ldots, A_n)$.
- $T\_Sets(cid, supp, sz)$.
- $T\_Rules(rid, cida, cidc, cid, conf)$.
- $T\_Trees\_A_i(treeid, cid)$, for all i=1 ... n.
- $T\_Treescharac\_A_i(treeid, acc, sz)$, for all i=1 ... n.

Although we consider here data tables having only categorical attributes, numerical attributes can easily be considered as long as they are first discretized. Also, observe that the mining views with the characteristics of the patterns, namely T_Sets, T_Rules, and $T\_Treescharac\_A$, can be extended with more attributes that represent other interestingness measures, such as the lift of an association rule.

As shown in the examples given in the previous subsections, in order to retrieve patterns over table T, the user simply needs to write SQL queries over the proposed mining views. The semantics of these queries is the same as that of queries over traditional relational tables.

Another important thing to note is that if the user wants to mine itemsets, association rules, or learn a decision tree from only a portion of table T, he or she should first create a new table T' from T, applying the appropriate selections and (or) projections. Then, the mining views associated with T', which are automatically created, will represent the patterns extracted from that corresponding portion of the data.

3 Constraint Extraction

In the previous section, we showed how a variety of data mining tasks and well-known constraints are expressed with SQL queries over the mining views. Nevertheless, recall that the mining views are virtual tables. Consequently, in order to answer a query involving one or more of these views, the system first needs to materialize them, that is, fill them with the corresponding mining objects (i.e., itemsets, association rules or decision trees).

Storing the whole collection of mining objects is not tractable. After all, the number of all possible objects can be extremely high and impractical to store. On the other hand, as shown in Example 4, the entire set of objects does not always need to be stored, but only those satisfying the constraints in the given SQL.

Example 4 Consider the SQL query in Figure 13. The query asks for decision trees targeting attribute *Play*, having a path containing an attribute test on "Outlook=Sunny", and also a path containing an attribute test on "Wind=Weak". Besides, the trees must have a size of at most 5 and an accuracy of at least 80%. Naturally, to answer this query, not all decision trees must be stored in view T_Trees_Play, $T\_Treescharac\_Play$, and $T\_Concepts$, but only those satisfying the aforementioned constraints.

```
select T1.*, C1.cid, C2.cid, D.acc
from T_Trees_Play T1,
     T_Trees_Play T2,
     T_Concepts C1,
     T_Concepts C2,
     T_Treescharac_Play D
where T1.cid = C1.cid
   and T2.cid = C2.cid
   and T1.treeid = T2.treeid
   and T1.treeid = D.treeid
   and C1.Outlook = 'Sunny'
   and C2.Wind = 'Weak'
   and D.sz <= 5
   and D.acc >= 80
```

Fig. 13 Example query over decision trees.

Observe that for the system to determine the set of objects to be stored in the mining views, it must be capable of detecting the constraints in the given SQL query. Towards this goal, we describe in this section an algorithm that extracts constraints from a mining query (i.e., a query that involves mining views), such as the one in Figure 13.

It is worth noticing that this extraction process is not simply a straightforward mapping from the conditions constructed from attribute names and constants in the where-clause of the input query (e.g., Outlook = 'Sunny', Wind = 'Weak', sz $\leq$ 5, and acc $\geq$ 80, in the query in Figure 13) to the constraints on the mining objects to be stored. To better clarify the goal of this process, consider, as another example, the query in Figure 14. Observe that it has the same constants in its where-clause as the query in Figure 13, but it asks for something different: here, the resultant decision trees must have attribute tests on "Outlook=Sunny" and on "Wind=Weak" on exactly the same concept (path), which is not the case of the decision trees asked by the query in Figure 13. The proposed algorithm is therefore meant to also detect such non-trivial constraints in the SQL queries.

```
select T1.*, C1.cid, C2.cid, D.acc
from T_Trees_Play T1,
     T_Concepts C1,
     T_Treescharac_Play D
where T1.cid = C1.cid
   and T1.treeid = D.treeid
   and C1.Outlook = 'Sunny'
   and C1.Wind = 'Weak'
   and D.sz <= 5
   and D.acc >= 80
```

Fig. 14 Another example query over decision trees.

We will consider in this section SQL queries that can be translated into relational algebra expressions [20]. The proposed algorithm, therefore, works on this type of expressions, rather than on the SQL query itself. Such a relational algebra expression has the advantage of being procedural, as opposed to SQL, which is declarative, making the constraint extraction easier.

3.1 Relational Algebra

Before proceeding to the details of the proposed algorithm, we briefly review the main concepts of relational algebra.

A relational algebra expression describes a sequence of operations on relations, which results in the answer of the query. Consider, for example, the SQL query shown in Figure 13. An equivalent relational algebra expression for this SQL query is as follows (for ease of presentation, the aliases T1, T2, C1, C2, and D, which were given to the mining views in the example SQL query, are also used here):

$$\begin{aligned}\pi_{T1.*,C1.cid,C2.cid,D.acc}(((&\sigma_{C1.Outlook=`Sunny'}\, Concepts\ C1)\\ &\bowtie_{C1.cid=T1.cid}\ (Trees\_Play\ T1))\\ \bowtie_{T1.treeid=T2.treeid}\ ((&\sigma_{C2.Wind=`Weak'}\, Concepts\ C2)\\ &\bowtie_{C2.cid=T2.cid}\ (Trees\_Play\ T2)))\\ \bowtie_{T1.treeid=D.treeid}\ (&\sigma_{D.sz\leq 5}(\sigma_{D.acc\geq 80\%}\, Treescharac\_Play\ D))\end{aligned}$$

This expression can also be represented by its syntax tree, as illustrated in Fig. 15. Although all mining views have prefix T, in the remainder of this section, we will omit it for ease of presentation.

In a relational algebra tree, the leaf nodes represent the base relations, the internal nodes represent the intermediate relations, which are constructed by a specific relational algebra operation, and the root node represents the relation which is the query result. For example, the *selection* operation

$$\sigma_{C1.Outlook=\text{`Sunny'}}\ Concepts\ C1,$$

represented by node (f), constructs a relation which is composed by those tuples from the relation *Concepts* that satisfy the predicate $C1.Outlook =$ 'Sunny'. The *join* operation

$$R_f \bowtie_{C1.cid=T1.cid} R_b$$

of node (i) (where R_f and R_b are the relations represented by nodes (f) and (b), respectively) combines tuples from R_f and R_b that have the same *cid* value into a single tuple. For every tuple of the first relation, R_f, and every tuple of the second relation, R_b, a new tuple, which is the concatenation of the two

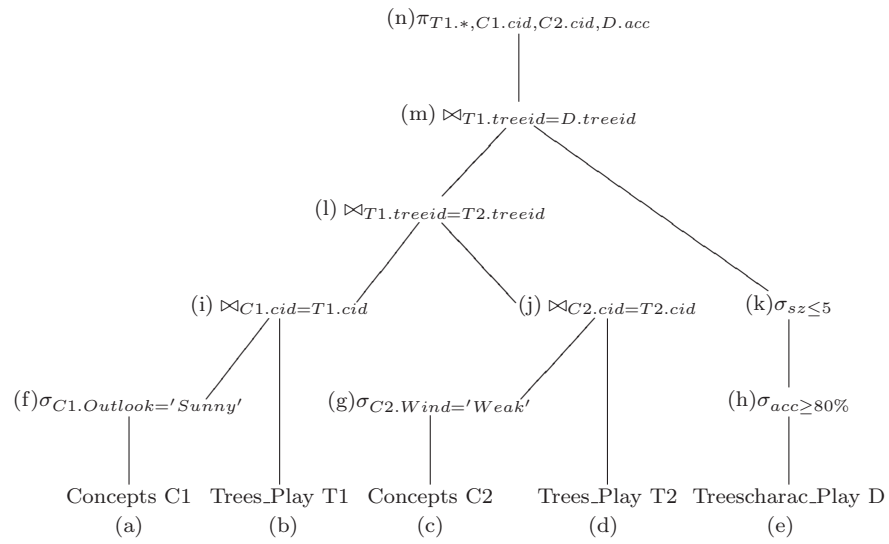


Fig. 15 An equivalent relational algebra tree for the query in Figure 13.

tuples, is in the resulting relation if it satisfies the predicate $C1.cid = T1.cid$. The *projection* operation

$$\pi_{T1.*,C1.cid,C2.cid,D.acc}$$

of node (n) produces a new relation that has only the attributes T1.*treeid*,C1.*, C2.* (note that, following standard SQL practice, the symbol '*' denotes the complete list of attributes).

Observe that the relational algebra operations can be unary or binary. This immediately translates into the degree of the nodes in the tree. For instance, the operation in node (f) (*selection*) is unary because it is applied to a single relation, while the operation in node (i) (*join*) is binary. For the sake of simplicity, we will continue to work with such expression trees instead of the relational algebra expressions themselves.

3.2 Algorithm

Consider a scenario of a database with a single table $T(A_1, \ldots, A_n)$ and its respective mining views. We describe the proposed constraint extraction algorithm by first introducing the notion of *annotation*.

3.2.1 Annotation

An annotation represents the necessary set of mining objects to be stored in the mining views in order to answer a given mining query. To build an annotation, two questions need to be addressed. The first one is: how to represent this

particular set of mining objects? In our approach, we opted to represent the whole collection of possible objects with the entity-relationship diagram (ER diagram) [21] given in Figure 16.

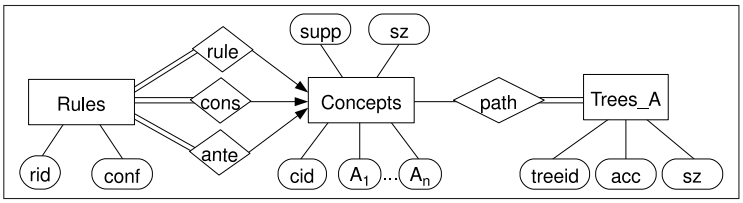


Fig. 16 The entity-relationship diagram of the whole collection of mining objects considered in this work.

In this diagram, the entities represent the different mining objects considered in this work, their corresponding attributes and relationships:

- The entity Rules represents the entire collection of association rules that can be extracted from T. It has 3 relationships with the entity Concepts, namely "ante" (from "antecedent"), "cons" (from "consequent") and the union of the two, referred to here as "rule". The relationship "rule" associates each rule with the concept from which the rule itself is generated, while "ante" and "cons" associate each rule with the concepts representing its antecedent and consequent, respectively. This entity also has the identifier (Rid) and the confidence (Conf) of the rules as attributes.
- The entity Trees_A represents the whole collection of decision trees that can be learned from table T, targeting an specific attribute A. This entity has attributes Treeid (identifier of the trees), Acc (accuracy of the trees), and Sz (size of the trees). In addition, it has a relationship, called "path" with the entity Concepts, which represents the fact that all of its instances has at least one concept as a path.
- Finally, the entity Concepts represents the entire collection of concepts that can be learned from table T. Note that this entity also has the support (Supp) and size (Sz) of each of its instances as attributes.

Note that since the user may impose some constraints in his or her mining query, the answer for such query can be represented as a particular instantiation of this ER diagram, the instances of which are those mining objects that satisfy the constraints in the query. As a result, the second question addressed is: how can we extract this particular instantiation from the input mining query, which actually refers to the virtual mining views, not to the mining objects themselves? Indeed, the mining views are a particular translation of the ER diagram to relational tables. For example, the mining view *Trees_A* is in fact the translation of the relationship "path" (i.e., one tuple in the mining view *Trees_A* contains a decision tree identifier as well as a concept identifier), while the mining view *Treescharac_A* is the translation of the entity

Trees_A. Given this, in our proposed model, a decision tree targeting a particular attribute A is spread over multiple tuples and mining views, as follows. One decision tree leads to: (a) one tuple in the mining view *Treescharac_A* (its characteristics), (b) multiple tuples (one per path) in the mining view *Trees_A*, and (c) multiple tuples in the mining view *Concepts* (its paths).

In the case of the entity Concepts, observe that its translation leads to the mining views *Concepts* and *Sets* (the latter contains the attributes *supp* and *sz*), while that of the entity Rules along with its relationships "ante", "cons", and "rule" lead only to the mining view *Rules*.

Such representational problem (i.e., the problem of mapping objects into relational tables), known as *impedance mismatch* [22], makes the extraction of the constraints on the mining objects more complicated. Therefore, we introduce here the notion of an annotation to facilitate this process: an annotation is a mapping from the constraints on the mining views in the mining query to the constraints on the mining objects in the ER diagram above. The annotation indicates a particular instantiation of that ER diagram whose instances are represented in terms of the entities they belong to and the constraints satisfied by them. Moreover, as the output attributes of the query (i.e., those that are being projected by the query) refer to attributes of the mining views, not to the attributes in the ER diagram, the annotation also includes a mapping from the output attributes of the corresponding query to the actual attributes in the ER diagram. The latter are called *originating attributes*. This mapping is also necessary due to the recursive nature of the algorithms that construct such annotations, which will be presented in more details later in this section.

Example 5 Consider the relational algebra query in Figure 15. The annotation associated to it is given in Figure 17. As described at the beginning of this section, this query asks for decision trees predicting attribute *Play*, with a size of at most 5 and an accuracy of at least 80%. In the annotation, this is therefore represented by the entity Trees_Play and the aforementioned constraints on its attributes *sz* and *acc*. Besides, these trees must have a path with an attribute test on "Outlook=Sunny", and also a path with an attribute test on "Wind=Weak". Since each path of a decision tree is an instance of the relationship "path" between the entities Trees_Play and Concepts, these constraints are represented in the annotation by two connections (labeled dotted lines) between the entity Trees_Play and the entity Concepts: one representing a path containing "Outlook=Sunny" and the other one containing "Wind=Weak".

The bottom line of the annotation represents the mapping from the output attributes of the query to their originating attributes in the ER diagram.

3.3 Algorithmic Details

Given a mining query, we now present the algorithmic details of how to construct the annotation for its equivalent relational algebra tree. Note that in this type of tree, each node n is associated with a sub-query, rooted at n.

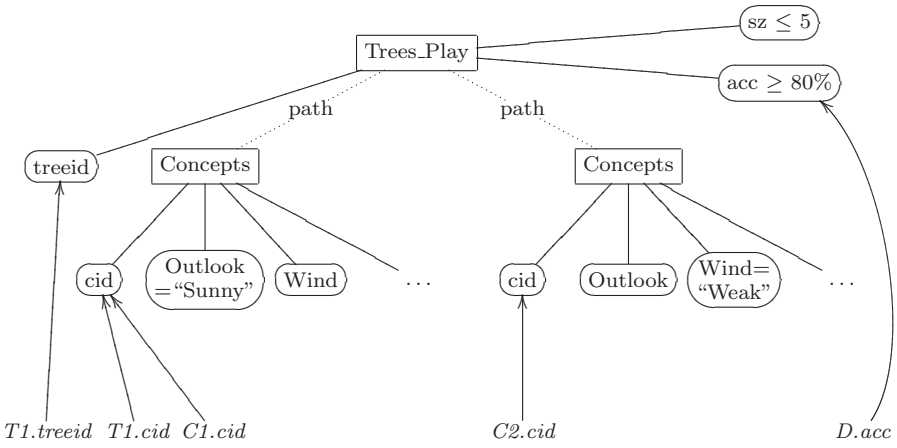


Fig. 17 Annotation for the query in Figure 15.

Proceeding in a bottom-up manner, the algorithm constructs for each node in the relational algebra tree an annotation that shows which mining objects (and under which constraints) should be stored in the mining views in order to answer the sub-query corresponding to that node. Except for the leaves, the annotation for a node is constructed using the annotations for its children as building blocks. The root's annotation is therefore the final result of the algorithm.

In the following, we describe how the annotation for leaf nodes and internal nodes are constructed. Concerning an internal node, the precise steps for this construction depends on its type, that is, the relational algebra operation it represents. A short overview of the following subsections is given below:

- **Leaf nodes**: The annotations for the leaf nodes are discussed next in Subsection 3.3.1.
- $\sigma_{Attr\theta a}$ (***Selection* with predicate** $Attr\theta a$): This is the type of, e.g., node (f) in Figure 15. The annotation for this type of node is built by Algorithm 1, which is described in Subsection 3.3.2.
- $\sigma_{Attr_1\theta Attr_2}$ (***Selection* with predicate** $Attr_1\theta Attr_2$): The Algorithm 2 constructs the annotation for this type of node, and it is discussed in Subsection 3.3.3.
- $\times$ (***Cartesian-Product***): The annotation for nodes of type $\times$ is discussed also in Subsection 3.3.3.
- $\pi_{Attr_1,...Attr_k}$ (***Projection***): This is the case of the root node (n) in Figure 15, the annotation for which is discussed in Subsection 3.3.3.
- Every operation $R_1 \bowtie_{Attr_1\theta Attr_2} R_2$ (e.g, node (i) in Figure 15) is replaced by the operation $\sigma_{Attr1\theta Attr2}(R_1 \times R_2)$. This operation is discussed in Subsection 3.3.3.
- $-$ (***Set-difference***) and $\cup$ (***Union***): These operations are discussed in Subection 3.3.4.

3.3.1 Leaf Nodes

The sub-query associated to a leaf node can be seen as a query of type "select * from X", where X is the mining view represented by the node. For example, the sub-query associated with node (a) in the relational algebra tree in Figure 15 asks for the collection of all possible concepts from the mining view *Concepts*. Therefore, the annotation for a leaf node n is simply the part of the ER diagram whose translation resulted in the mining view represented by n.

Figure 18 shows the annotations for all possible types of leaf nodes that represent the mining views considered in this work. Every annotation also includes a mapping from the output attributes of the sub-query associated with the node itself to the originating attributes in the ER diagram.

It is worth noticing that if a leaf node represents the data table T, no annotation is constructed for that node, as T does not need to be materialized.

3.3.2 Selection with Predicate Attrθa

We now describe how the annotation for nodes representing a selection operator with predicate *Attr*θa is constructed. Consider, e.g, node (f) in the example tree, which is reproduced on the top right of Figure 19. Its associated sub-query selects only those tuples coming from node (a) that satisfy the selection predicate "C1.Outlook=Sunny". This means that to answer this sub-query we only need those concepts that contain "Outlook=Sunny". The annotation for this node is therefore constructed by simply associating the constraint '="Sunny"' to the originating attribute of C1.Outlook in the annotation of node (a) (which is depicted at the bottom of Figure 19). The resultant annotation is shown at the top left of Figure 19, while the steps for constructing this type of annotation is presented in Algorithm 1.

Algorithm 1: Annotate_Selection1(annotation for child node,Attr,a,θ).

Input: annotation for child node, *Ann*,
Attr (an output attribute in *Ann*),
a (a constant),
$\theta \in \{=, \neq, \leq, \geq, <, >\}$.

Output: annotation for node of type *Attr*θa.

```
1 begin
2 |  A := originating attribute of Attr in Ann;
3 |  Let C be the conjunction of constraints already associated with A;
4 |  C := C ∧ θa;
5 |  return Ann with the new constraint C for A;
6 end
```

3.3.3 Join $R_1 \bowtie_{Attr_1 \theta Attr_2} R_2$

Here we discuss the construction of the annotation for nodes of type join $R_1 \bowtie_{Attr_1 \theta Attr_2} R_2$. As remarked before, this operation is first replaced by the

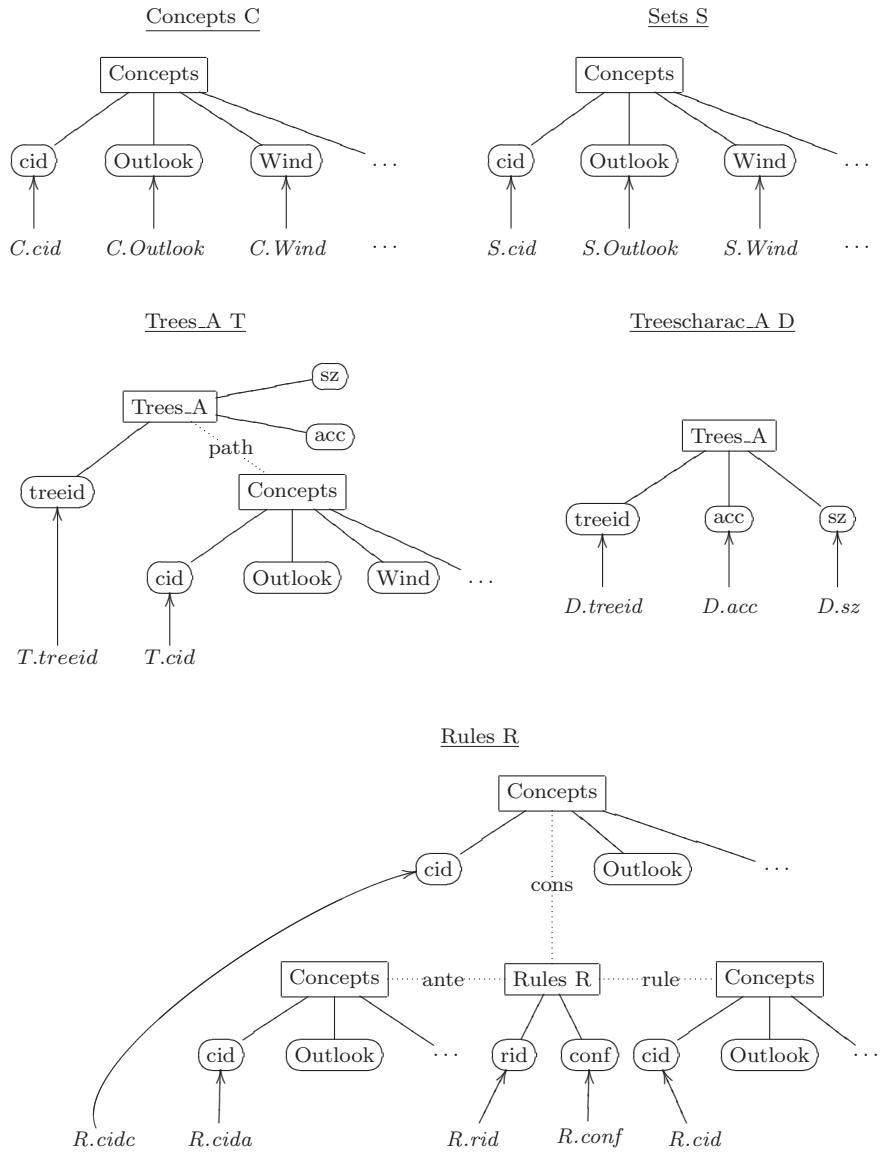


Fig. 18 Annotations for the leaf nodes representing the proposed mining views.

operation $\sigma_{Attr_1 \theta Attr_2}(R_1 \times R_2)$. For instance, consider node (i) in the example tree, which is reproduced at the top of Figure 20. Its annotation is constructed by first building the annotation for the Cartesian product $R_f \times R_b$ followed by the annotation for the selection operation "C1.cid=T1.cid". The annotation

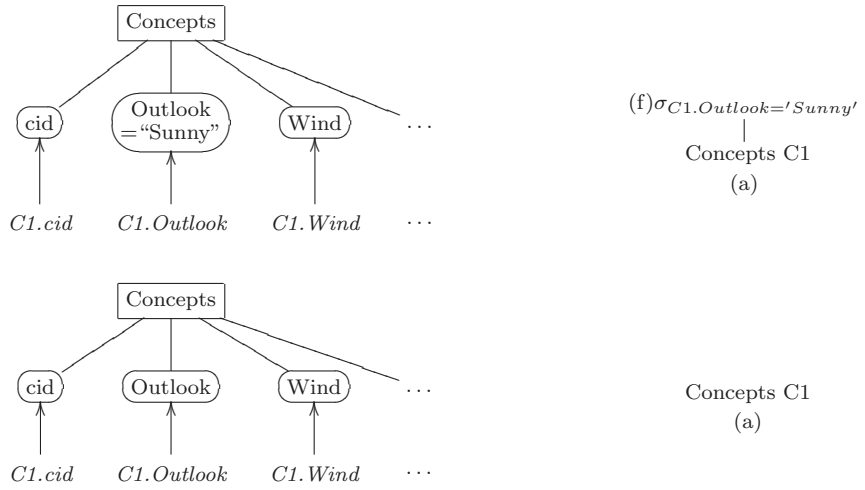


Fig. 19 Annotation for node (f) (top left), its corresponding sub-query (top right), and the annotation for its child node, (a) (bottom).

for the Cartesian product, illustrated at the bottom of Figure 20, is simply the union of the annotations for nodes (f) and (b), since they represent the mining objects that are necessary for the execution of such operation. The bottom part of Figure 20 is therefore the concatenation of the annotations of nodes (f) and (b).

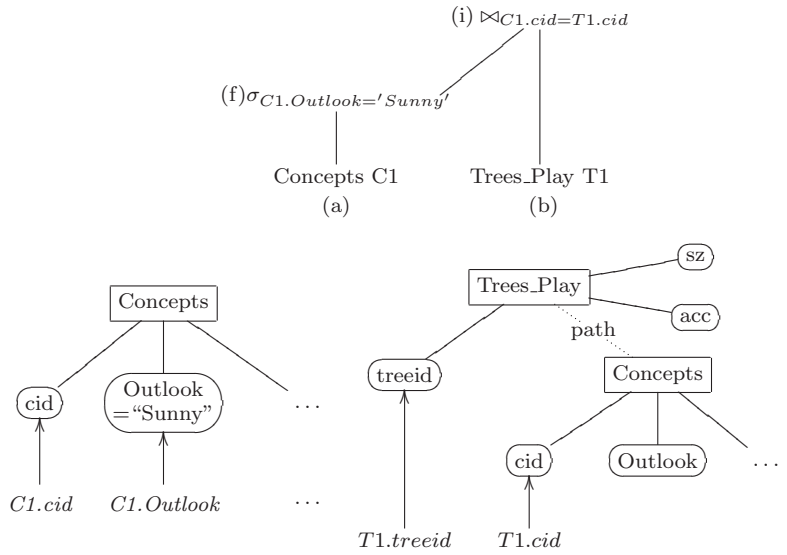


Fig. 20 Annotation for $R_f \times R_b$ (bottom) and node (i) as seen in the example algebra tree (top).

We now examine the construction of the annotation for the selection operation with predicate $\sigma_{Attr_1 \theta Attr_2}$. This procedure is implemented by Algorithm 2, which receives as input the annotation for the corresponding Cartesian product operation discussed above. Consider the selection "C1.cid=T1.cid" in our running example. From the Cartesian product annotation, we observe that "C1.cid" originates from the identifier attribute Cid of the concepts represented in the annotation of node (a), while "T1.cid" originates from the identifier attribute Cid of the concepts in the annotation of node (f). Then, according to the equality defined by the selection predicate, the concepts to be considered for this operation are those represented in both annotations (b) and (f), that is, the same collection of concepts having "Outlook=Sunny" that are also paths of decision trees. As a result, the annotation for this operation is obtained by merging the entities Concepts in these annotations, as depicted in Figure 21. Observe that due to the equi-join operation, the output attributes "C1.cid" and "T1.cid" have now the same originating attribute in this annotation.

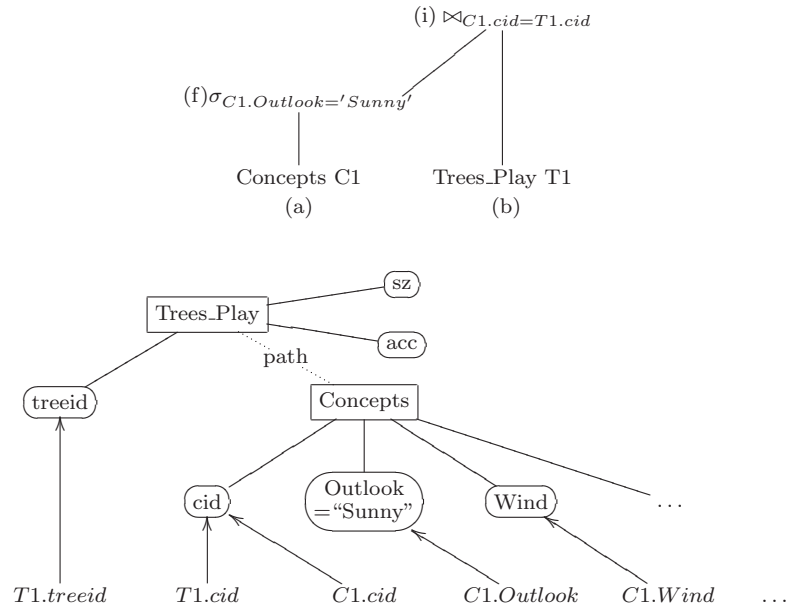


Fig. 21 Annotation for node (i) (bottom) and the corresponding sub-query (top).

Algorithm 2: Annotate_Selection2(annotation for the Cartesian product, Attr_1, Attr_2, θ).

Input: annotation for the Cartesian product, *Ann*,
Attr_1 (an output attribute in *Ann*),
Attr_2 (an output attribute in *Ann*),
$\theta \in \{=, \neq, \leq, \geq, <, >\}$.
Output: annotation for node of type $\sigma_{Attr_1 \theta Attr_2}$.

```
1  begin
2      if θ is '=' then
3          A := originating attribute of Attr1 in Ann;
4          B := originating attribute of Attr2 in Ann;
5          if both A and B are non-identifier attributes with the same
           name then
6              merge_attributes(Ann,A,B);
7          else
8              Let V be the entity such that A belongs to V;
9              Let W be the entity such that B belongs to W;
10             if both A and B are concept identifiers then
11                 merge_concepts(Ann,V,W);
12             else if both A and B are rule identifiers then
13                 merge_rules(Ann,V,W);
14             else if both A and B are identifiers of decision trees
               targeting the same attribute then
15                 merge_trees(Ann,V,W);
16             end
17         end
18     end
19     return Ann;
20 end
```

Algorithm 2 annotates every node representing the operation σ with predicate $Attr_1 \theta Attr_2$. Note that if θ is not '=' (see condition in line 2), then the output annotation is equal to the input annotation, i.e., the annotation for the Cartesian product operation. This is due to the fact that if θ is not '=', all mining objects represented in the annotation of the Cartesian product are necessary to execute such operation. In other words, the number of necessary mining objects cannot be reduced in this case. The same happens when an equi-join is made between attributes with different names (see conditions in lines 5, 10, 12, and 14), or between two identifiers of decision trees targeting different attributes (see condition in line 14).

Let us now consider another example for the join operation, corresponding to node (l) in the example tree in Figure 15. In this case, the process to construct its annotation is very similar to that executed for node (i). Due to the selection predicate "T1.treeid = T2.treeid", however, the method merge_trees

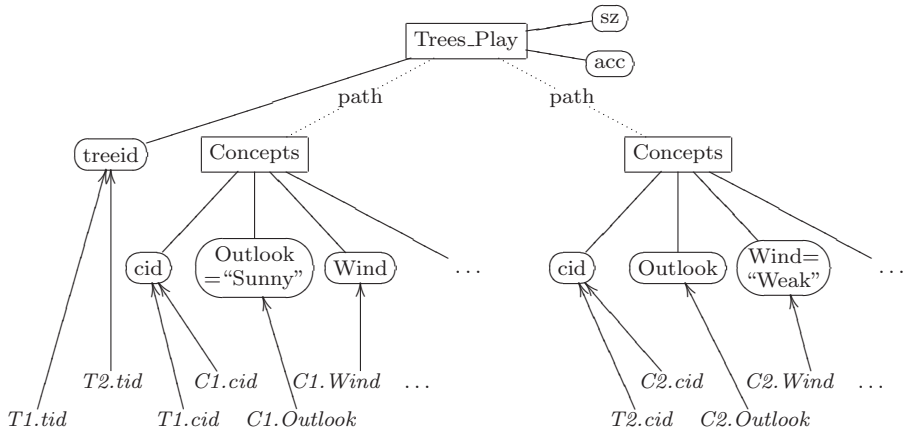


Fig. 22 Annotation for node (l).

will be executed in Algorithm 2. The resulting annotation for node (l) is shown in Figure 22.

Algorithm 2 is constructed based on the types of mining objects that are considered in this work. Next, we briefly describe the methods used by this algorithm.

- **merge_attributes**: this method receives as input the same annotation received by the Algorithm 2 and the two attributes to be merged, A and B. The most important task of this method is to merge the constraints on the attributes A and B, by taking their conjunction. The method then removes all references to attribute B from the input annotation, keeping only attribute A.
- **merge_concepts**: it also receives as input the annotation received by Algorithm 2 and two entities Concepts to be merged, V and W. The method merges the attributes of V and W (using the method merge_attributes) and removes all references to W from the annotation.
- **merge_rules**: this method merges two entities Rules, V and W, similarly to the method merge_concepts. Remember that an entity Rules always appear in an annotation connected to three entities Concepts (see Subsection 3.3.1), which represent, for each rule, its antecedent, consequent, and their union. Since this triplet uniquely identifies a rule, merging two entities Rules implies merging the corresponding triplets as well. Therefore, this method also merges the three entities Concepts, using the method merge_concepts.
- **merge_trees**: this method is also similar to merge_concepts. This time, however, V and W represent two entities Trees_A. If V or W (or both) is connected to one or more entities Concepts (via a dotted line labeled "path"), these are not merged, but are all connected to the remaining entity. This is due to the fact that a path of a decision tree does not

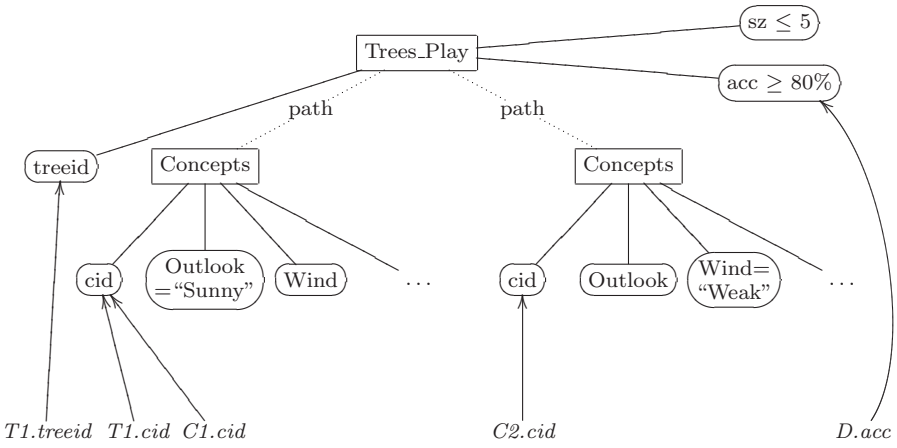


Fig. 23 Annotation for node (n), which is the final annotation.

uniquely identify it, that is, a decision tree may have more than one path (see ER diagram in Figure 16).

Projection with Attribute List $\mathbf{Attr_1}, \ldots, \mathbf{Attr_k}$ Lastly, we describe how the annotation for nodes of type $\pi_{Attr_1,\ldots Attr_k}$ is constructed. It is built by discarding from the annotation of its child node the output attributes that are not present in the projected attribute list $\{Attr_1, \ldots Attr_k\}$. As an example, root node (n) simply projects the tuples coming from node (m) on the attributes $T1.treeid$, $C1.cid$, $C2.cid$, and $D.acc$. Its annotation is the same as that for node (m), keeping, however, only the projected output attributes and their originations. The annotation for this node, which is the final annotation for our example query, is shown in Figure 23.

Remember that an annotation is a particular instantiation of the ER diagram of possible mining objects, the instances of which represent the objects to be stored in the mining views (and under which constraints) to answer the corresponding mining query. In this way, the annotation in Figure 23 indicates that the objects to be stored are decision trees predicting the attribute *Play* (given by the presence of the entity Trees_Play), having an accuracy of at least 80%, size of at most 5, a path containing an attribute test on "Outlook=Sunny", and also a path containing an attribute test on "Wind=Weak" (given by the dotted lines between the entity Trees_Play and the entities Concepts, as well as the constraints on the attributes Acc and Sz of the former).

3.3.4 Operations Set-Difference *and* Union

We conclude Section 3 by discussing the construction of the annotation for nodes of type *Set-Difference*, denoted by $-$, and *Union*, denoted by $\cup$.

The result of the operation R_1-R_2 is a relation obtained by including all tuples from R_1 that do not appear in R_2. For instance, consider the relational algebra query in Figure 24. In this query, R_1 is the relation produced by node (n), while R_2 is the relation produced by node (o).

The query in Figure 24 asks for association rules $X \rightarrow Y$ with support greater than 5 and confidence between 70% and 80% (node (n)) that are not the result of chaining a rule $X \rightarrow Z$ (with the same characteristics of $X \rightarrow Y$) and a correct rule (100% confidence) $Z \rightarrow Y$ (node (o)).

Example 6 Suppose that in the database we have the following rules produced by node (n): $AB \rightarrow C$, $AB \rightarrow D$, $AC \rightarrow B$, $B \rightarrow C$, $B \rightarrow D$, and the following 100% confident rule: $C \rightarrow D$. Then, the rules $AB \rightarrow D$ and $B \rightarrow D$ will be in the result of node (o), since they can be obtained by chaining $AB \rightarrow C$ and $C \rightarrow D$, and $B \rightarrow C$ and $C \rightarrow D$, respectively.

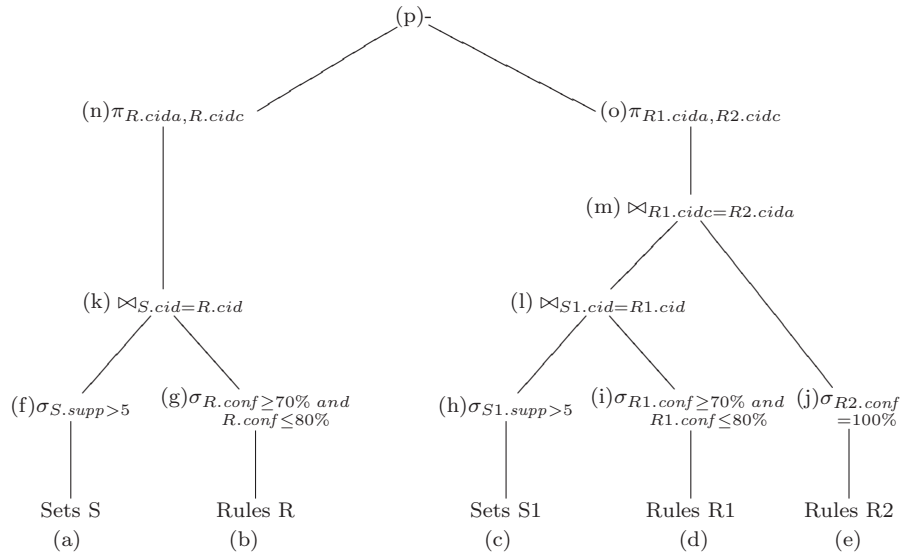


Fig. 24 A relational algebra tree with a node of type set-difference (node (p)).

Suppose now that we keep only the annotation for node (n) as the final annotation for the example query, which means that only the rules coming from node (n) are materialized in the mining view *Rules*. In this way, *Rules* will not contain any 100% rule and, therefore, the relation produced by node (o) will be empty and the query will not be properly answered. For instance, in the case of Example 6, the rules $AB \rightarrow D$ and $B \rightarrow D$ would be produced incorrectly as part of the query's answer. The annotation for node (p) must be such that all rules needed for the correct evaluation of node (o) are present in the mining views.

The annotation for a node of type set-difference is thus defined as being the concatenation of the annotations for both of its child nodes, keeping, however, only the output attributes in the annotation of the left node. In other words, the construction of the annotation for this type of node is the same as for a node of type Cartesian product (described at the beginning of Section 3.3.3) followed by the projection on the output attributes of the left child node. The annotation for the right child node is stripped of its output attributes and added to the annotation for the left child node. The two annotations are disjoint in the resulting one.

In the case of the operation $R_1 \cup R_2$, the result relation is a relation with all tuples that appear in either R_1 or R_2, or in both of them. Therefore, the annotation for this type of nodes is constructed in the same way as for the nodes of type set-difference.

4 Constraint Exploitation

Having presented the steps for constructing the annotation for a given mining query, in this section we discuss how the materialization of the mining views itself is performed by the system, based on an annotation.

The mining objects to be stored in the mining views are first computed by data mining algorithms, which receive as parameters the constraints present in the given annotation. Next, the results are stored as tuples in the corresponding mining views.

Each type of mining object considered in this article is associated with a certain algorithm, as follows:

- For itemsets and association rules, the system is linked to the Apriori-like algorithm by Christian Borgelt[2] and the rule miner implementation by Bart Goethals[3].
- For decision trees, the system is linked to the exhaustive decision tree learner called Clus-EX, described in detail in [3], which searches for all trees satisfying the constraints "mimimum accuracy" and "maximum size". Clus-EX learns decision trees with binary splits only.

As an example, consider the annotation in Figure 23, which indicates that the system should store a subset of decision trees that target the attribute *Play*. As remarked earlier, a decision tree is spread over the mining views *Concepts*, *Trees_A*, and *Treescharac_A*. Therefore, after mining for such decision trees using Clus-EX, the system stores the obtained results in the aforementioned mining views. A minor detail here is that Clus-EX is not yet capable of exploiting constraints on the paths of the decision trees (in this case, the exploited constraints are $acc \geq 80$ and $sz \leq 5$). On the one hand, for this example, the system stores more decision trees than those necessary to answer

[2] `http://www.borgelt.net/software.html`

[3] `http://www.adrem.ua.ac.be/~goethals/software/`

the query. On the other hand, this is not a fundamental problem, since the non-exploited constraints are used anyway by the database management system to filter the results before showing them to the user. Consequently, the efficiency of the system may be affected, but not its correctness.

It is worth noticing, however, that there are often several possible strategies to materialize the required mining views depending on the input annotation. As an example, consider the annotation in Figure 25. Starting from the left of the figure, the annotation indicates that the system should store: (a) association rules with a confidence of at least 80% that are generated from concepts with support of at least 3 (dotted line with label "rule" between the entities Rules and Concepts, and the constraint on the attribute Supp); (b) decision trees for attribute Play, having size of at most 5, and at least one path among the concepts that generate the aforementioned rules (constraint on the attribute Sz, and dotted line between the entities Trees_Play and Concepts).

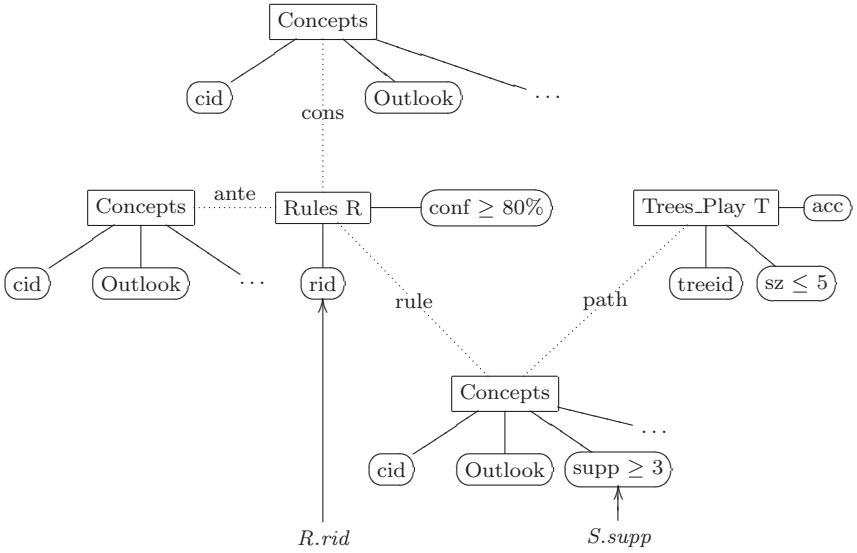


Fig. 25 Example annotation.

Based on this annotation, possible strategies for materializing the mining views are:

1. First, mine association rules having $supp \geq 3$ and $conf \geq 80\%$. Next, mine decision trees predicting attribute *Play*, having $sz \leq 5$, and at least one path among the itemsets previously computed to generate the rules. The association rules are stored in views *Rules*, *Concepts* and *Sets*, while the decision trees are stored in views *Trees_Play*, *Treescharac_Play* and *Concepts*.

2. First, mine decision trees predicting attribute *Play*, having $sz \leq 5$. Then, for every path in the generated decision trees, we compute its support and size. Finally, we mine association rules having $supp \geq 3$ and $conf \geq 80\%$, using, as itemsets, the paths of the decision trees already generated. In this case, all generated decision trees are first stored in views *Trees_Play*, *Treescharac_Play*, *Concepts*, and *Sets*. After that, the view *Rules* and *Concepts* are materialized with the generated association rules.

The choice of strategy depends on the characteristics of the available data mining algorithms in terms of the constraints they can exploit and the type of input they need. Currently, the system always mines itemsets first, followed by association rules and then decision trees. Given this order, our system adopts the strategy 1 to materialize the mining views based on the annotation above.

Some materialization strategies as well as mining algorithms may be more efficient than others and the collection of tuples that are eventually stored may also differ. However, no matter what strategy the system takes, the query will be answered correctly, since all the necessary tuples will certainly be stored.

Another aspect of the materialization is the occurrence of duplicates when, in the final annotation, mining objects of the same type are represented more than once. This is the case of the annotation in Figure 26. If the system simply mines for itemsets twice (once for itemsets with $supp \geq 4$, and once for itemsets with $supp \geq 5$), duplicates will be generated for itemsets having support of at least 5. One way to solve this problem is to take the disjunction of the constraints and put it into *disjoint DNF*, as proposed by Goethals and Van den Bussche in [23]: in disjoint DNF, the conjunction of any two disjuncts is unsatisfiable. The disjoint DNF in this case is $(supp \geq 5) \vee (supp \geq 4 \wedge supp < 5)$. By mining itemsets as many times as the number of disjuncts in the DNF formula, no duplicates are generated. This is the strategy adopted by our system.

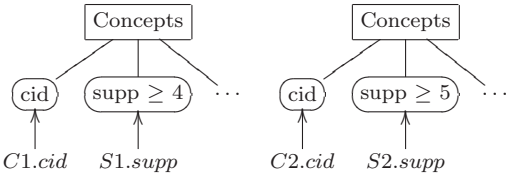


Fig. 26 Example annotation for a case of generation of duplicates.

5 An Illustrative Scenario

In this section, we describe a data mining scenario that explores the interactive and iterative capabilities of our inductive database system. The system

was developed into the well-known open source database system PostgreSQL[4] (written in *C*).

The main steps of the system is as follows: when the user writes an SQL query, the parser module of PostgreSQL is invoked, which generates an SQL parse tree. Then, if such tree refers to one or more mining views, the following steps are carried out:

1. First, the SQL parse tree generated by PostgreSQL is converted into a relational algebra expression tree so as to be used by our constraint extraction algorithm.
2. The constraint extraction algorithm, which was integrated into the code of PostgreSQL, is then invoked. Having the relational algebra tree as input, the algorithm extracts the constraints from it, as described in Section 3.
3. Afterwards, the system triggers the necessary constraint-based data mining algorithms in order to materialize the mining views with the required tuples, as discussed in Section 4. These algorithms were therefore linked to PostgreSQL's code as well.
4. Just after the materialization, the work-flow of the database system continues and the query is executed as if the patterns were there all the time.

In order to have a user-friendly interface to the system, we adapted the web-based administration tool PhpPgAdmin[5]. We refer the reader to [5, 7] for more details on the implementation and efficiency evaluation of the system.

Next, we illustrate an extended data mining scenario with SQL queries over the mining views. Differently to the scenario presented in [6], here we do not learn a classifier, but mine for non-redundant correct association rules.

5.1 Scenario

The scenario presented in this section consists in extracting knowledge from the gene expression data which resulted from a biological experimentation concerning the transcription of *Plasmodium Falciparum* [24] during its reproduction cycle (IDC) within the human blood cells.

The Plasmodium Falciparum is a parasite that causes human malaria. The data gather the expression profiles of 472 genes of this parasite in 46 different biological samples.[6] Each gene is known to belong to a specific biological function. Each sample in turn corresponds to a time point (hour) of the IDC, which lasts for 48 hours. During this period, the merozoite (initial stage of the parasite) evolves to 3 different identified stages: Ring, Trophozoite, and Schizont. In addition, due to reproduction, one merozoite leads to up to 32 new ones during each cycle, after which a new developmental cycle is started. Figure 27 shows the percentage of parasites (y-axis) that are at the Ring (black

[4] `http://www.postgresl.org/`

[5] `http://phppgadmin.sourceforge.net/`

[6] The data is available at http://malaria.ucsf.edu/SupplementalData.php

curve), Trophozoite (light gray curve), or Schizont (dark gray curve) stage, at every time point of the IDC (x-axis).

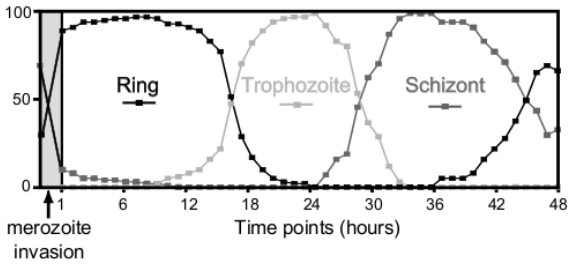


Fig. 27 Major developmental stages of Plasmodium Falciparum parasite (Figure from [24]). The three curves, in different levels of gray, represent the percentage of parasites (y-axis) that are at the Ring (black), Trophozoite (light gray), or Schizont stage (dark gray), at every time point of the IDC (x-axis).

These data were stored into 3 different tables in our system, as illustrated in Figure 28. They are the following:

- GeneFunctions(function_id, function): represents the biological functions. There are in total 12 different functional groups.
- Samples(sample_name, stage): represents the samples themselves. Two data points are missing, namely the 23rd and 29th hours. We added to this table the attribute called *stage*, the values of which are based on the curves illustrated in Figure 27: this new attribute discriminates the samples having at least 75% of the parasites in the Ring (stage=1), Trophozoite (stage=2) or Schizont (stage=3) stage. Samples that contain less than 75% of any parasite stage were assigned to stage 4, a "non-identified" stage. Thus, for our scenario, stage 1 corresponds to time points between 1 and 16 hours; stage 2 corresponds to time points between 18 and 28 hours; and stage 3 gathers time points between 32 and 43 hours.
- Plasmodium(gene_id, function_id, tp_1, tp_2,. . . , tp_48): represents, for each of the genes, its corresponding function and its expression profile. As proposed in [24], we take the logarithm to the base 2 of the raw expression values.

Having presented the data, we are now ready to describe the goal of our scenario. In gene expression analysis, a gene is said to be highly expressed, according to a biological sample, if there are many RNA transcripts in the considered sample. These RNA transcripts can be translated into proteins, which can, in turn, influence the expression of other genes. In other words, it can make other genes also highly expressed. This process is called *gene regulation* [24].

In this context, analogously to what the biologists have studied in [24], we want to characterize the parasite's different stages by identifying the genes

Plasmodium

gene_id	function_id	tp_1	tp_2	...	tp_48
1	12	-0.13	0.12	...	0.11
2	12	0.24	0.48	...	-0.03
...	...	...	...	...	...
472	5	1.2	0.86	...	1.15

GeneFunctions

function_id	function
1	Actin_myosin_mobility
2	Cytoplasmic_translation_machinery
...	...
12	Transcription_machinery

Samples

sample_name	stage
tp_1	1
tp_2	1
...	...
tp_48	4

Fig. 28 The Plasmodium data.

that are active during each stage. More precisely, we want to identify, for each different stage, the functional groups whose genes have an unusual high level of expression or, as the biologists say, are overexpressed in the corresponding set of samples. By considering the samples corresponding to a specific stage and the genes that are overexpressed within those samples, we might have insights into the regulation processes that occur during the development of the parasite. As pointed out in [24], understanding these regulation processes would provide the foundation for future drug and vaccine development efforts toward eradication of the malaria.

Observe that decision trees are not appropriate for the analysis we want to perform; they are most suited for predicting, which is not our intention here. Therefore, in our scenario we mine for association rules. A couple of pre-processing steps have to be performed initially, such as the discretization of the expression values. These steps are described in detail in the first 3 subsequent subsections. The remaining subsections show how the desired rules can be extracted from the data.

5.1.1 Step 1: Pre-processing 1

Since our intention is to characterize the parasite's stages by means of the functional groups and not of the individual genes themselves, first, we create a view on the data that groups the genes by the function they belong to. The corresponding pre-process query is shown below:[7]

[7] For the sake of readability, ellipsis were added to some of the SQL queries presented in this section and in the following ones, which represent sequences of attribute names, attribute values, clauses etc.

```
1. create view PlasmodiumAvg as
2. select G.function,
3.         avg(p.tp_1) as tp_1,
4.         ...,
5.         avg(p.tp_48) as tp_48
6. from Plasmodium P, GeneFunctions G
7. where P.function_id = G.function_id
8. group by G.function
```

The view called "PlasmodiumAvg" calculates, for every different functional group, the average expression profile (arithmetic mean) over all time points (see lines from 2 to 5).

5.1.2 Step 2: Pre-processing 2

Since we want the functional groups as components of the desired rules (antecedents and/or consequents), it is therefore necessary to transpose the view PlasmodiumAvg, which was created in the previous step. In other words, we need a new view in which the gene functional groups are the columns and the expression profiles are the rows. To this end, we use the PostgreSQL function called *crosstab*[8]. As *crosstab* requires the data to be listed down the page (not across the page), we first create a view on PlasmodiumAvg, called "PlasmodiumAvgTmp", which lists data in such format. The corresponding queries are shown below.

```
 1. create view PlasmodiumAvgTmp as
 2. select function as tid, 'tp_1' as item, tp_1 as val
 3. from PlasmodiumAvg
 4. union
 5. ...
 6. union
 7. select function as tid, 'tp_48' as item, tp_48 as val
 8. from PlasmodiumAvg



 9. create view PlasmodiumTranspose as
10. select * from crosstab
11. ('select item, tid, val from PlasmodiumAvgTmp order by item',
    'select distinct tid from PlasmodiumAvgTmp order by item')
12. as (sample_name text, Actin_myosin_mobility real,...,
                              Transcription_machinery real)
```

[8] We refer the reader to `http://www.postgresql.org/docs/current/static/tablefunc.html` for more details on the *crosstab* function.

5.1.3 Step 3: Pre-processing 3

Having created the transposed view PlasmodiumTranspose, the third and last pre-processing step is to discretize the gene expression values so as to encode the expression property of each functional group of genes.

In gene expression data analysis, a gene is considered to be overexpressed if its expression value is high with respect to its expression profile. One approach to identify the level of expression of a gene is the method called *x% cut-off*, which was proven to be successful in [25]: a gene is considered overexpressed if its expression value is among the x% highest values of its expression profile, and underexpressed otherwise. With x=50, a gene is tagged as overexpressed if its expression value is above the median value of its profile.

As in this scenario the data are log transformed (very high expression values are deemphasized), the distribution of the data is symmetrical and, therefore, median expression values are very similar to mean values. As computing the mean value is straightforward in SQL and as we are not dealing with genes independently, but with groups of genes, we use a slight adaptation of the *50% cut-off* method: we encode the overexpression property by comparing it to the mean value observed for each group, rather than the median. We first create a view, called "PlasmodiumTranspAvg", which calculates, for every group of genes, its mean expression value. This computation is performed by the following query:

```
1. create view PlasmodiumTranspAvg as
2. select avg(Actin_myosin_mobility) as avg_Actin_mm,
3. ...
4. avg(Transcription_machinery) as avg_Transcription_m
5. from PlasmodiumTranspose
```

Afterwards, we create the new table named "PlasmodiumSamples" applying the aforementioned discretization rule. The query that performs this discretization step is shown below. Notice that the attribute *stage* is also added to the new table PlasmodiumSamples (see line 2).

```
 1. create table PlasmodiumSamples as
 2. select P.sample_name, S.stage,
 3. case when P.Actin_myosin_mobility > avg_Actin_mm then
 4.    'overexpressed'
 5.  else
 6.    'underexpressed'
 7.  end as Actin_myosin_mobility,
 8.  ...
 9.  case when P.Transcription_machinery > avg_Transcript_m then
10.     'overexpressed'
11.  else
12.     'underexpressed'
13.  end as Transcription_machinery
14.  from PlasmodiumTranspose P, PlasmodiumTranspAvg, Samples S
15.  where P.sample_name = S.sample_name
16.  order by S.stage
```

5.1.4 Step 2: Mining over Association Rules

After creating the table PlasmodiumSamples, in this new step, we look for the desired rules. The corresponding query is shown below:

```
1. create table RulesStage as
2. select R.rid, S.sz, S.supp, R.conf,
          CAnt.stage as stage_antecedent
          CCon.*
3. from PlasmodiumSamples_Sets S,
PlasmodiumSamples_Sets SAnt,
         PlasmodiumSamples_Concepts CAnt,
         PlasmodiumSamples_Concepts CCon,
         PlasmodiumSamples_Rules R
4. where R.cid = S.cid
5.   and CAnt.cid = R.cida
6.   and CCon.cid = R.cidc
7.   and S.supp >= 10
8.   and R.conf = 100
9.   and R.cida =  SAnt.cid
10.  and SAnt.sz = 1
11.  and CAnt.stage <> '?'
12.  order by Ant.stage
```

As we want to characterize the parasite's stages themselves by means of the gene functions, we look for rules having only the attribute *stage* as the antecedent (see lines 9, 10 and 11) and gene function(s) in the consequent. Additionally, since we want to characterize the stages without any uncertainty,

we only look for correct association rules, that is, rules with a confidence of 100% (see line 8). Finally, as the shortest stage is composed of 10 time points in total (not considering the dummy stage), we set 10 as the minimum support (line 7). The 381 resultant rules are eventually stored in the table called "RulesStage" (see line 1).

5.1.5 Step 3: Post-processing

The previous query has generated many redundant rules [26]: for each different antecedent, all rules have the same support and 100% confidence. Notice, however, that as we are looking for all gene groups that are overexpressed according to a given stage, it suffices to analyze, for each different stage (antecedent of the rules), only the rule that has maximal consequent. Given this, all one has to do is to select, for each different stage, the longest rule. The corresponding query is presented below. The sub-query, in lines from 3 to 5, computes, for every antecedent (stage), the maximal consequent size.

```
1. select R.*
2. from RulesStage R,
3. (select max(sz) as max_sz, stage_antecedent
4. from RulesStage
5. group by stage_antecedent) R1
6. where R.sz = R1.max_sz
7.   and R.stage_antecedent = R1.stage_antecedent
```

antecedent (stage)	consequent	
	overexpressed	underexpressed
Ring	Early ring transcripts Transcription machinery	Deoxynucleotide synthesis DNA replication machine Plastid genome **TCA cycle**
Trophozoite	Glycolytic pathway Ribonucleotide synthesis Deoxynucleotide synthesis DNA replication Proteasome	Actin myosin motors Early ring transcripts Merozoite invasion
Schizont	Plastid genome Merozoite invasion Actin myosin mobility	**Cytoplasmic translation machinery** Ribonucleotide synthesis Transcription machinery

Fig. 29 Correct association rules with maximum consequent for the second scenario.

The 3 rules output by the last query are presented in Figure 29. As shown in Figure 30, they are consistent with the conclusion drawn in the corresponding biological article [24]. Each graph in Figure 30, from B to M, corresponds to the average expression profile of the genes of a specific functional group (the names of the functions are shown at the bottom of the figure). The functions

are ordered, from left to right, with respect to the time point when there is a peak in their expression profiles (the peak value is shown in parentheses) and they are assigned to the parasite's stage during which this peak occurs (the name of the stages are presented at the top of the figure). Observe that, according to Figure 30, the functions *Early ring transcripts* and *Transcription machinery* are related to the early Ring and Ring stages, which is in fact indicated by the first extracted rule. The *Glycolytic pathway*, *Ribonucleotide synthesis*, *Deoxynucleotide synthesis*, *DNA replication*, and *Proteasome* are related to the early Trophozoite and the Trophozoite stages, which is also consistent with the second extracted rule. Finally, *Plastid genome*, *Merozoite Invasion*, and *Actin myosin mobility* have been associated to the Schizont stage by the biologists, which is indeed consistent with the third rule.

Notice that *TCA cycle* and *Cytoplasmic translation machinery* (in bold in Figure 29) appear in the extracted rules only as underexpressed, meaning that they are never overexpressed during an entire stage. *TCA cycle*, for example, is underexpressed during the whole Ring stage and becomes overexpressed from the second half of the Trophozoite until the beginning of Schizont stage.

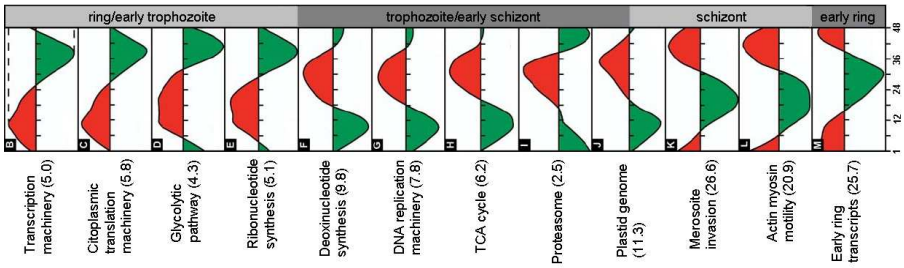


Fig. 30 The temporal ordering of functional groups of genes (an adapted figure from [24]). Each graph, from B to M, corresponds to the average expression profile of the genes of a specific functional group. The biologists of [24] have assigned each functional group to the parasite's stage in which it achieves its highest expression value.

6 Related Work

There already exist several proposals for developing an inductive database following the framework introduced in [1]. Below, we list some of the best known examples.

SQL-based proposals. The system SINDBAD (structured inductive database development), developed by Wicker et al. [14], processes queries written in SIQL (structured inductive query language). SIQL is an extension of SQL that offers a wide range of query primitives for feature selection, discretization, pattern mining, clustering, instance-based learning and rule induction. Another extension of SQL has been proposed by Bonchi et al. [16] which is

called SPQL (simple pattern query language). SPQL provides great support to pre-processing and supports a very large set of different constraints. A system called ConQueSt has also been developed, which is equipped with SPQL and a user-friendly interface. In both SIMBAD and ConQueSt, the number of constraints that the user can possibly pose within their mining queries is fixed. In our system, the users have the flexibility of ad-hoc querying: as the data mining query language is traditional SQL, the users can think of new constraints not available at implementation time. Moreover, new combinations of patterns are also possible in our framework.

In *Microsoft's Data Mining extension (DMX)* of SQL server [13], a classification model M can be created and used afterwards to give predictions via the so-called prediction joins. Although the philosophy behind the prediction join is somewhat related to what we propose, our work goes much further: in [6], for example, we present a scenario in which we learn a classifier having the maximum accuracy among those with a certain size. DMX would not be appropriate to accomplish this task, since it does not provide any other operations for manipulating models, other than browsing and prediction.

Programming language-based approaches. An acronym for "Aggregate & Table Language and System" [12], ATLaS is a Turing-complete programming language based on SQL that enables data mining operations on top of relational databases. An important aspect of this approach is that, in order to mine data, one needs to implement in this language the appropriate mining algorithm. For instance, in [12], the authors show the code for the Apriori algorithm, which becomes considerably complex when implemented with this language. In addition, specific code must be written to manipulate the mining results, since these should be encoded by the user into the database. Similarly, found patterns need to be decoded and encoded back into the database so as to be used in subsequent queries.

In [27], an algebraic framework for data mining is presented, which allows the mining results as well as ordinary data to be manipulated. The framework is based on the 3W model of Johnson et al. [28]. In short, "3W" stands for the "Three Worlds" for data mining: the D(ata)-world, the I(ntensional)-world, and the E(xtensional)-world. Ordinary data are manipulated in the D-world, regions representing mining results are manipulated in the I-world, and extensional representations of regions are manipulated in the E-world.

Since the 3W model relies on black box mining operators, a first contribution of the work in [27] is to extend the 3W model by "opening up" these black boxes, using generic operators in a data mining algebra. Two key operators in this algebra are regionize κ, which creates regions (or models) from data tuples, and a restricted form of looping called mining loop λ, which is meant to manipulate regions. The resulting data mining algebra, which is called $\mathcal{MA}$, is studied and properties concerning the expressive power and complexity are established. As ATLaS, $\mathcal{MA}$ can also be seen as a programming language based on those key operators. These programming languages are much less declarative, making them less attractive for query optimization.

7 Conclusions

In this article, we described a practical inductive database based on virtual mining views. The development of the system has been motivated by the need to provide an intuitive framework that covers a wide variety of models in a uniform way, and enables to easily define meaningful operations, such as prediction of new examples. We show the benefits of its implementation through an illustrative data mining scenario on real-word biological data. In summary, the main advantages of our system are the following:

- **Satisfaction of the closure principle**: since, in the proposed system, the data mining query language is standard SQL, the closure principle is clearly satisfied.
- **Flexibility to specify different kinds of patterns**: our system provides a very clear separation between the patterns it currently represents, which in turn can be queried in a very declarative way (SQL queries). In addition to itemsets, association rules and decision trees, the flexibility of ad hoc querying allows the user to think of new types of patterns which may be derived from those currently available. For example, in [7] we show how frequent closed itemsets [29] can be extracted from a given table T with an SQL query over the available mining views $T\_Concepts$ and T_Sets.
- **Flexibility to specify ad hoc constraints**: the proposed system is meant to offer exactly this flexibility: by virtue of a full-fledged query language that allows of ad hoc querying, the user can think of new constraints that were not considered at the time of implementation.
- **Intuitive way of representing mining results**: in our system, patterns are all represented as sets of concepts, which makes the mining views framework as generic as possible, not to mention that the patterns are easily interpretable.
- **Support for post-processing of mining results**: again, thanks to the flexibility of ad hoc querying, post-processing of mining results is clearly feasible in the proposed inductive database system.

We identify three directions for further work: currently, the mining views are in fact empty and only materialized upon request. Therefore, inspired by the work of Harinarayan et al. [30], the first direction for further research is to investigate which mining views (or which parts of them) could actually be materialized in advance, as it is too expensive to materialize all of them. This would speed up query evaluation. Second, the constraint extraction could be improved so as to incorporate a larger variety of constraints. For example, constraints such as "the maximum accuracy" are, at the current time, not recognized and hence not filtered out. The main reason for this is that the constructions for expressing these constraints is non-trivial in SQL, requiring sub-queries and aggregations. One direction we want to explore in this perspective is dynamic optimization, where the result on one part of the query or of a sub-query can be used to constrain another part. Finally, the system developed so far covers only itemset mining, association rules and decision trees.

A direction for further work is to extend it with other models, considering the exhaustiveness nature of the queries the users are allowed to write.

8 Acknowledgements

This work has been partially supported by the projects IQ (IST-FET FP6-516169) 2005/8, GOA 2003/8 "Inductive Knowledge bases", FWO "Foundations for inductive databases", and BINGO2 (ANR-07-MDCO 014-02). When this research was performed, Hendrik Blockeel was a post-doctoral fellow of the Research Foundation - Flanders (FWO-Vlaanderen), Élisa Fromont was working at the Katholieke Universteit Leuven, and Adriana Prado was working at the University of Antwerp.

References

1. Imielinski, T., Mannila, H.: A database perspective on knowledge discovery. Communications of the ACM **39** (1996) 58–64
2. Calders, T., Goethals, B., Prado, A.: Integrating pattern mining in relational databases. In: Proc. ECML-PKDD European Conf. on Machine Learning and Principles and Practice of Knowledge Discovery in Databases. (2006) 454–461
3. Fromont, E., Blockeel, H., Struyf, J.: Integrating decision tree learning into inductive databases. In: ECML-PKDD Workshop on Knowledge Discovery in Inductive Databases (KDID) (Revised selected papers). (2007) 81–96
4. Blockeel, H., Calders, T., Fromont, E., Goethals, B., Prado, A.: Mining views: Database views for data mining. In: ECML-PKDD Workshop on Constraint-based Mining and Learning (CMILE). (2007)
5. Blockeel, H., Calders, T., Fromont, E., Goethals, B., Prado, A.: Mining views: Database views for data mining. In: Proc. IEEE ICDE Int. Conf. on Data Engineering. (2008)
6. Blockeel, H., Calders, T., Fromont, E., Goethals, B., Prado, A.: An inductive database prototype based on virtual mining views. In: Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery in Databases. (2008)
7. Prado, A.: An Inductive Database System Based on Virtual Mining Views. PhD thesis, University of Antwerp, Belgium (December 2009)
8. Han, J., Fu, Y., Wang, W., Koperski, K., Zaiane, O.: DMQL: A data mining query language for relational databases. In: ACM SIGMOD Workshop on Data Mining and Knowledge Discovery (DMKD). (1996)
9. Meo, R., Psaila, G., Ceri, S.: An extension to sql for mining association rules. Data Mining and Knowledge Discovery **2**(2) (1998) 195–224
10. Imielinski, T., Virmani, A.: Msql: A query language for database mining. Data Mining Knowledge Discovery **3**(4) (1999) 373–408
11. Wang, H., Zaniolo, C.: Nonmonotonic reasoning in ldl++. Logic-based artificial intelligence (2001) 523–544
12. Wang, H., Zaniolo, C.: Atlas: A native extension of sql for data mining. In: Proc. SIAM Int. Conf. on Data Mining. (2003) 130–144
13. Tang, Z.H., MacLennan, J.: Data Mining with SQL Server 2005. John Wiley & Sons (2005)
14. Wicker, J., Richter, L., Kessler, K., Kramer, S.: Sinbad and siql: An inductive databse and query language in the relational model. In: Proc. ECML-PKDD European Conf. on Machine Learning and Principles and Practice of Knowledge Discovery in Databases. (2008) 690–694
15. Nijssen, S., Raedt, L.D.: Iql: A proposal for an inductive query language. In: ECML-PKDD Workshop on Knowledge Discovery in Inductive Databases (KDID) (Revised selected papers). (2007) 189–207

16. Bonchi, F., Giannotti, F., Lucchese, C., Orlando, S., Perego, R., Trasarti, R.: A constraint-based querying system for exploratory pattern discovery information systems. Information System (2008) Accepted for publication.
17. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proc. VLDB Int. Conf. on Very Large Data Bases. (1994) 487–499
18. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
19. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. Data Mining and Knowledge Discovery (1996) 152–159
20. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
21. Chen, P.P.: The entity-relationship model: Toward a unified view of data. ACM Transactions on Database Systems **1** (1976) 9–36
22. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems. Addisson Wesley (2006)
23. Goethals, B., Bussche, J.V.D.: On supporting interactive association rule mining. In: Proc. DAWAK Int. Conf. on Data Warehousing and Knowledge Discovery. (2000) 307–316
24. Bozdech, Z., Llinás, M., Pulliam, B.L., Wong, E.D., Zhu, J., DeRisi, J.L.: The transcriptome of the intraerythrocytic developmental cycle of plasmodium falciparum. PLoS Biology **1**(1) (2003) 1–16
25. Becquet, C., Blachon, S., Jeudy, B., Boulicaut, J.F., Gandrillon, O.: Strong association rule mining for large-scale gene-expression data analysis: a case study on human SAGE data. Genome Biology **12** (2002)
26. Zaki, M.J.: Generating non-redundant association rules. In: Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery in Databases. (2000) 34–43
27. Calders, T., Lakshmanan, L.V.S., Ng, R.T., Paredaens, J.: Expressive power of an algebra for data mining. ACM Transactions on Database Systems **31**(4) (2006) 1169–1214
28. Johnson, T., Lakshmanan, L.V.S., Ng, R.T.: The 3w model and algebra for unified data mining. In: Proc. VLDB Int. Conf. on Very Large Data Bases, Morgan Kaufmann (2000) 21–32
29. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering frequent closed itemsets for association rules. In: Proc. ICDT Int. Conf. on Database Theory. (1999) 398–416
30. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing data cubes efficiently. In: Proc. ACM SIGMOD Int. Conf. on Management of Data. (1996) 205–216