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 that does not rely on a new data mining query language, but on plain SQL. 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 data table. 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 SQL using a unifying framework. As a proof of concept, we illustrate a complete data mining scenario with SQL queries over the mining views, which is executed in our system.

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

1 Introduction

Data mining is an interactive process in which different tasks may be performed sequentially and the output of these tasks may be combined to be used as input for subsequent ones. In order to effectively support this knowledge discovery process, the integration of data mining into database management systems has become necessary. By integrating data mining more closely into a database system, separate steps such as data pre-processing, data mining, and post-processing of the results, can all be handled using one query language. The concept of *inductive database systems* has been proposed so as to achieve such integration [1].

In order to tackle the ambitious task of building an inductive database system, 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.

Although SQL is the language of choice for database querying, it is generally acknowledged that it does not provide enough features for data mining processes. Indeed, SQL offers no true data mining facilities for, e.g., the discovery of frequent itemsets. Therefore, several researchers have proposed new query languages, which are extensions of SQL, as a natural way to provide an inductive database system [2–10]. As we show in [11,12], however, these languages have some limitations: For example, i) there is little attention to the closure principle; the output of a mining query cannot or can only very difficultly be used as the input of another query, ii) if the user wants to express a constraint that was not explicitly foreseen by the developer of the language, he or she will have to do so with a post-processing query, if possible at all, and iii) data mining results are often offered as static objects that can only be browsed or in a way that does not allow for easy post-processing.

With these limitations in mind, we describe in this article an inductive database system that is implemented by extending the structure of the database itself, which can be queried using standard SQL, rather than relying on a new query language for data mining. More precisely, we propose a system in which the user can query the collection of all possible patterns as if they were stored in traditional relational tables. Since the number of all possible patterns can be extremely high and impractical to store, the main challenge here is how this storage can be implemented efficiently. For example, in the concrete case of itemsets, an exponential number of itemsets would need to be stored. To solve this problem, we introduced the so-called *virtual mining views*, as presented in [12–16]. The mining views are relational tables that virtually contain the complete output of data mining tasks executed over a given data table. For example, for itemset mining, there is a table called *Sets* virtually storing all frequent patterns. Whenever the user queries such a table, or virtual mining view, an efficient data mining algorithm (e.g., Apriori [17]) is triggered by the database system, which materializes, that is, stores in this table at least those tuples needed to answer the query. Afterwards, the query can be executed as if the patterns were there all the time.

The proposed 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 Section 2. We show how such special tables can be developed for three popular data mining tasks, namely frequent itemset mining, association rule discovery and decision tree learning. Such framework, initially discussed in the European project IQ ("Inductive Queries for Mining Patterns and Models", IST FET FP6-516169, 2005-2008) [12], is a reevaluation of the works proposed in [13, 14], leading to a unified framework that is more elegant and simpler than the originally proposed frameworks.

Note that this querying approach assumes the user uses certain constraints in his or her query, asking for only a subset of all possible patterns. As an example, the user may query from the mining view *Sets* all itemsets with a certain frequency. Therefore, the entire set of patterns does not always need to be stored in the mining views, but only those that satisfy the constraints imposed by the user. In Section 3, we fully describe for the first time an algorithm to extract constraints from SQL queries over the mining views.

Once the constraints are detected and extracted by the system, they are exploited by data mining algorithms, the results of which are stored in the required mining views just before the actual execution of the query. There are often several possible strategies to fill the required mining views based on the extracted constraints, an issue discussed in Section 4.

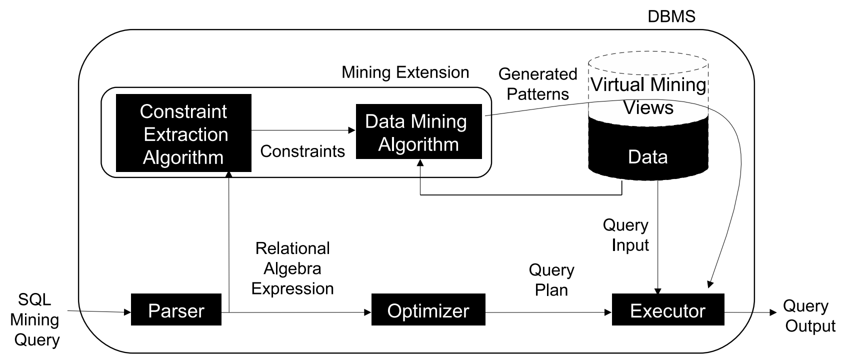


Fig. 1 An inductive database system based on virtual mining views.

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 complete model as was implemented is illustrated in Figure 1 and the implementation is available upon request[2]. In our inductive database system, 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.

In the remainder of this article, we illustrate the interactive and iterative capabilities of our system with a data mining scenario in Section 5 (for other example scenarios we refer the reader to [12, 15, 16]). Section 6 makes an overview of related work, Section 7 presents a discussion on the limitations of the system as well as on how to extend it, and the article is concluded in Section 8.

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 executed over a given data table. 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 Entity-Relationship Model

Figure 2 presents the entity-relationship model (ER model) [18] which is eventually translated into the relational tables (or mining views) presented in this article. The entities and relationships of the ER model are described in the following.

2.1.1 Concepts

We begin by describing the entity Concepts. We assume to be working on a database that contains the table $T(A_1, \ldots, A_n)$, having only categorical attributes. In our proposed framework, the output of data mining tasks executed over table T are generically represented by what we call *concepts*. We denote a concept as a conjunction of attribute-value pairs that is definable over T. Therefore, the entity Concepts represents all such concepts that are definable

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

[2] E-mail to Adriana Prado (`adriana.bechara.prado@gmail.com`).

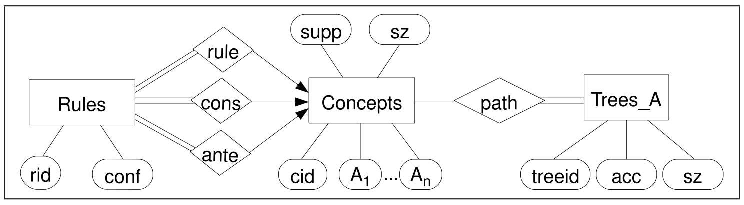


Fig. 2 The entity-relationship model which is eventually translated into the mining views described in this article.

over T. We assume that these concepts can be sorted in lexicographic order and that an identifier, represented here by the attribute cid, can unambiguously be given to each concept. In addition, the attribute supp (from "support") gives the number of tuples in T satisfied by the concept and sz is its size, in number of attribute-value pairs.

Example 1 Consider T the classical relational table Playtennis(*Day*, *Outlook*, *Temperature*, *Humidity*, *Wind*, *Play*) [19], which is illustrated in Figure 3. The concept below is an example concept that is defined over this table:

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

Since it is satisfied by 3 different tuples in table Playtennis (highlighted in Figure 3), its support is 3. Its size is also 3, as it is composed by 3 attribute-value pairs.

□

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. 3 The data table Playtennis.

Next, we explain how itemsets, association rules and decision trees extracted from table T are represented in our ER model by the concepts in entity Concepts.

2.1.2 Itemsets

As itemsets in a relational database are conjunctions of attribute-value pairs, they are represented here as concepts. Note that the absolute frequency and size of the itemsets are given by the attributes Supp and Sz, respectively.

2.1.3 Association Rules

The entity Rules represents the entire collection of association rules that can be extracted from T. Since association rules are built on top of itemsets and itemsets are in fact concepts in the ER model, rules are represented here as a triplet of concepts. More specifically, the entity Rules has 3 relationships with the entity Concepts, namely "ante" (from "antecedent"), "cons" (from "consequent") and the union of these 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. We assume that a unique identifier, attribute Rid, can be given to each rule, and the attribute Conf is its confidence.

2.1.4 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, 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 is represented in our ER model by a set of concepts, where each concept represents one path from the root to a leaf of the tree.

Example 2 The tree in Figure 4 can be seen as the following set of concepts (paths):

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

$$(Outlook = \text{‘Sunny’} \wedge Humidity = \text{‘Normal’} \wedge Play = \text{‘Yes’})$$

$$(Outlook = \text{‘Overcast’} \wedge Play = \text{‘Yes’})$$

$$(Outlook = \text{‘Rain’} \wedge Wind = \text{‘Strong’} \wedge Play = \text{‘No’})$$

$$(Outlook = \text{‘Rain’} \wedge Wind = \text{‘Weak’} \wedge Play = \text{‘Yes’})$$

□

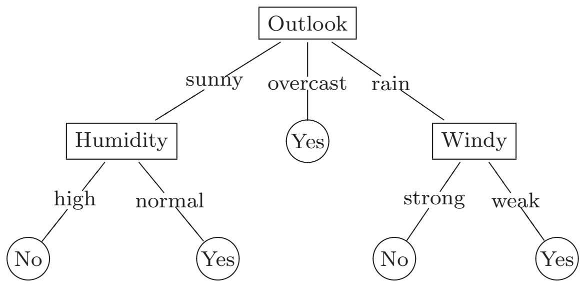


Fig. 4 An example decision tree.

The collection of all decision trees predicting a particular target attribute A is therefore represented by the entity Trees_A. This entity has attributes Treeid (identifier of the trees), Acc (accuracy of the tree), and Sz (size of the tree in number of nodes). In addition, it has a relationship, called “path” with the entity Concepts, which represents the fact that all of its instances have at least one concept as a path.

Note that the entity Trees_A represents the decision trees semantically, not syntactically. That is: the predictive semantics of a decision tree is determined entirely by describing its leaves (using the conjunction of attribute-value pairs from root to leaf), which is exactly what is represented by this entity. The structure of a tree, however, is not determined uniquely by its leaves: sometimes different trees may have exactly the same set of leaves. Such trees always represent exactly the same predictive function, though.

2.2 The Relational Model

We now define the virtual mining views themselves, which are a particular translation of the ER model above into relational tables.

2.2.1 The Mining View T_Concepts

Consider again table $T(A_1, \ldots, A_n)$ with 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 order 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 in Example 1, which is defined over table Playtennis in Figure 3, is represented by the tuple:

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

We are now ready to define the mining view $T\_Concepts(cid, A_1, \ldots, A_n)$. This view virtually contains all concepts that are definable over table T and it is a translation of the entity Concepts with attributes cid, $A_1, \ldots, A_n$.

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 5 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* [20] built from table T, with the difference that the wildcard value "ALL" introduced in [20] is replaced by the value '?'. By following the syntax introduced in [20], the mining view *T_Concepts* would be created with the SQL query shown in Figure 6 (consider adding the identifier *cid* after its creation).

In the remainder of this section, we consider T the table Playtennis and use the concepts in Figure 7 for the illustrative examples (where an identifier has been given to each of the concepts).

2.2.2 The Mining Views T_Sets and T_Rules

All itemsets extracted from table T are represented in our framework by the mining view $T\_Sets(cid, supp, sz)$, which is a translation of the entity Concepts with attributes cid, supp, and sz. This view 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 mining view $T\_Rules(rid, cida, cidc, cid, conf)$. This view is the translation of the entity Rules along with its relationships "ante", "cons", and "rule", and is described by the definition below:

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. 5 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. 6 The data cube that represents the contents of the mining view *T_Concepts*.

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. 7 A sample of the mining view *Playtennis_Concepts*, which is used for the illustrative examples in Subsections 2.2.2 and 2.2.3.

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. The attribute *rid* is the rule identifier, *cida* is the identifier of the concept representing its antecedent, *cidc* is the identifier of the concept representing its consequent, *cid* is the identifier of the union of those two concepts, and *conf* is the confidence of the rule.

Figure 8 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 7. The rule has identification number 1.

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. 8 Mining views for representing itemsets and association rules. The attributes *cida*, *cidc*, and *cid* refer to concepts given in Figure 7.

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 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 3.

Example 3 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 9 shows how this task is performed in the proposed framework. Without the mining views *T_Concepts* or *T_Sets*, this task would be executed with a much more complicated query, as given in query (B) in Figure 9. □

Example 4 In Figure 10, query (C) is another example query over itemsets, while query (D) is an example query over association rules.

Query (C) represents the task of finding itemsets with large area. The area of an itemset corresponds to the size of the tile, which is formed by the attribute-value pairs in the itemset in the tuples that support it. The mining of large tiles, i.e., itemsets with a high area, is useful in constructing small summaries of a database [21].

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

(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. 9 Example queries over itemsets with (query (A)) and without (query (B)) the mining views *T_Concepts* and *T_Sets*.

same time allowing their composition, as association rules are built on top of frequent itemsets.

2.2.3 The Mining Views T_Trees_A and T_Treescharac_A

The collection of all decision trees predicting a particular target attribute A_i is represented by the mining view $T\_Trees\_A_i(treeid, cid)$, which is the translation of the relationship "path" in the ER model. We formally define it as follows:

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

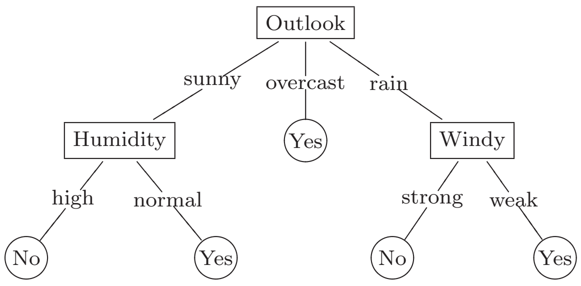
(C)

```
select C.*, S.supp, S.sz,
       S.supp * S.sz as area
from T_Concepts C, T_Sets S
where C.cid = S.cid
  and ((S.supp * S.sz > 60)
      or (S.supp > 10))
```

(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. 10 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. 11 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 7.

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, the view $T\_Treescharac\_A_i$(*treeid*,*acc*,*sz*), representing several characteristics of a tree learned for one specific target attribute A_i, is defined as the translation of the entity Trees_A as presented next:

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 11 shows how the example decision tree in Figure 4 is represented in the mining views *T_Trees_Play* and *T_Treescharac_Play* by using the concepts in Figure 7. The example decision tree predicts the attribute *Play* of table *T* and has identification number 1.

(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. 12 Example queries over decision trees.

Example 5 In Figure 12, we present two 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.

Example 6 Consider the table BestTrees created after the execution of query (E), in Figure 12. Figure 13 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. 13 An example prediction query.

2.3 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.

Example 7 Consider query (H) in Figure 14. 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. 14 Example query combining patterns.

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

- *T_Concepts*(*cid*,A_1,...,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.

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 expressiveness of these queries is the same as that of queries over traditional relational tables.

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., concepts, 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 8, the entire set of objects does not always need to be stored, but only those satisfying the constraints in the given SQL query.

Example 8 Consider the SQL query in Figure 15. 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. □

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). It is worth noticing that this extraction process is more involved than just selecting the conditions that relate to the mining views in the where-clause of the query (i.e., Outlook = 'Sunny', Wind = 'Weak', sz $\leq$ 5, and acc $\geq$ 80 for the query in Figure 15), as the following example shows.

```
select T1.treeid, 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. 15 Example query over decision trees.

Example 9 Consider the query in Figure 16. This query has exactly the same constants as the query in Figure 15 (w.r.t. the mining views in its where-clause), but has a different semantics; it asks for all decision trees in which the conditions Outlook = ‘Sunny’ and Wind = ‘Weak’ occur *in the same path* of the decision tree. □

```
select T1.treeid, 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. 16 Another example query over decision trees.

As the extracted constraints cannot be presented as a list of constraints, we will introduce a more advanced structure, the so-called *annotation*. In summary, the processing of a given SQL query over the mining views will proceed as follows:

1. The constraints are extracted from the query and represented in the form of an *annotation*. Annotations are introduced in Subsection 3.1 and the algorithm to extract them in Subsection 3.2.
2. Based on the annotation, one or more constraint-based data mining algorithms are executed. The selection of the algorithms and the order in which they are executed is not determined by the annotation and different strategies are possible. The execution step is discussed in Section 4.
3. The output of the mining algorithms is used to fill the virtual tables. From here on the tables are no longer virtual.

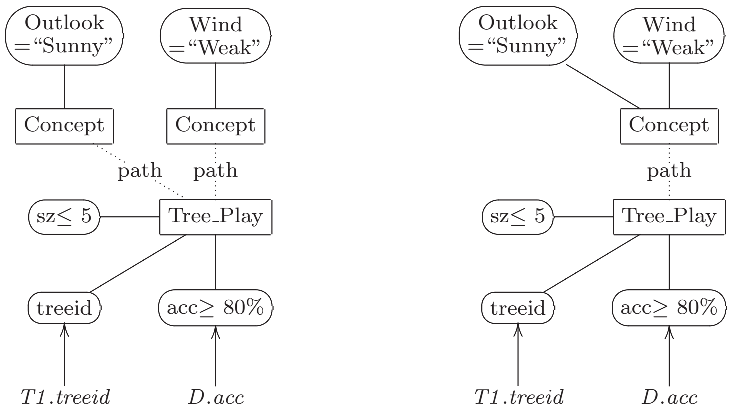


Fig. 17 Annotation for the query in Figure 15 (left) and Figure 16 (right), respectively.

4. The SQL query is executed by an underlying relational database system, using normal SQL query optimization and execution.
5. The tables are then emptied and become virtual again. Notice that in this step there is room for improvement by caching some of the results for subsequent queries. Such optimizations are, however, beyond the scope of this paper.

3.1 Annotations

An annotation of an SQL query can be seen as an instantiation of the ER model of the mining objects in Figure 2. It represents the following three types of information about the query:

1. Which mining objects are involved in the query and how do they relate to each other? (e.g., a concept representing a path in a tree)
2. From which mining objects do the attributes in the query result come? (e.g., a tree identifier, an attribute of a concept, the identifier of the consequent of an association rule)
3. Which atomic constraints hold on the attributes of the mining objects?

More formally, an annotation is defined as follows:

Definition 7 (Annotation) An annotation for a query q is a three-tuple (I, M, C), where I is an abstract instantiation of the ER model, i.e., a set of objects for which the attribute values have not been specified, and relations between them, respecting the cardinality constraints in the ER model. M is a mapping from the attributes of q to a set of attributes in I, which are called the *originating attributes*. C is a partial function mapping attributes in I to constraints. The constraints can be any Boolean combination of attribute-value comparisons.

Furthermore, the annotation should describe a sufficient set of mining objects to compute the answer of the query. That is, for any database, if we fill the virtual tables with the mining objects (concepts, sets, rules, or trees) that can be mined from this database and that satisfy all constraints in C and relations in I, then the query should return the correct answer.

Example 10 Figure 17 contains the annotations for the queries in Figures 15 and 16, respectively. The objects in the instantiation I are depicted by rectangles containing the type of the object. Attributes are in rounded boxes. The relations are depicted by dotted lines. The attributes at the bottom are the attributes in the query. The arrows pointing out from them indicate the mapping M. The constraints are included into the attribute they refer to. To avoid visual clutter, only those attributes that are in the image of the mapping M or in the range of C and objects that have such attributes are visually represented.

The left annotation, e.g., describes the following set of mining objects: (i) all trees with an accuracy of at least 80% and a size of maximal 5 such that there is a path in the tree with "Wind=Weak" and a path with "Outlook=Sunny", (ii) all concepts with "Wind=Weak" that participate in such a tree, and (iii) all concepts with "Outlook=Sunny" that participate in such a tree. "Filling the virtual tables with these mining objects" implies that all such trees are mined. For each such tree we (i) add to the mining view *Trees* as many tuples as there are concepts that describe a path in this tree, (ii) add a tuple in the mining view *Treescharac_Play*, and (iii) for all concepts that describe a path in this tree, a tuple is added in the mining view *Concepts*. If we fill the tables in this way, the query will clearly be answered correctly.

□

To summarize, an annotation for a given query describes the mining objects necessary to form the tuples in the result of this query. Notice that the definition does not insist on minimality, because the problem of creating a minimal annotation is incomputable (see [13]). This situation is very similar to the inability to find the most optimal query execution plan for relational queries; even deciding whether a relational query will always return an empty answer is undecidable (see, e.g., [22]). The algorithm in the next subsection will hence describe how to find a correct, but not necessarily tight annotation.

3.2 Bottom-up Construction of an Annotation

We will consider in this section SQL queries that can be translated into relational algebra expressions [22]. 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.

3.2.1 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 15. An equivalent relational algebra expression for this SQL query is given in Figure 18. 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. The expression is given by its syntax tree to ease the explanation of the further parts. Although all mining views have prefix T, in the remainder of this section, we will omit it for ease of presentation. The leaf nodes of the tree are the mining views or normal relations. The internal nodes represent intermediate results in the computation of the query by applying one of the operations $\times$, σ, $\cup$, $\cap$, π, $\bowtie$, or $-$ on the intermediate results represented by its children. For example, the *selection* operation

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

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

$$\pi_{T1.treeid,D.acc}$$

of node (n) produces a new relation that has only the attributes T1.*treeid* and D.*acc*. For a complete description of the relational algebra, we refer the reader to [23].

3.2.2 Algorithm

We will compute the annotation for the query in a bottom-up fashion. We will start with basic annotations in the leaf nodes. Next, we will form annotations for the sub-queries corresponding to the intermediate results in the internal nodes. This will be done by combining the annotations of their children, based on the operation they represent. Some of the operations $\times$, σ, $\cup$, $\cap$, π, $\bowtie$, and $-$ are redundant. These operations will be rewritten using the other operations. We will now discuss all operations and illustrate our algorithm with the relational algebra expression in Figure 18. For a fully formal description of the algorithm, we refer the reader to [24].

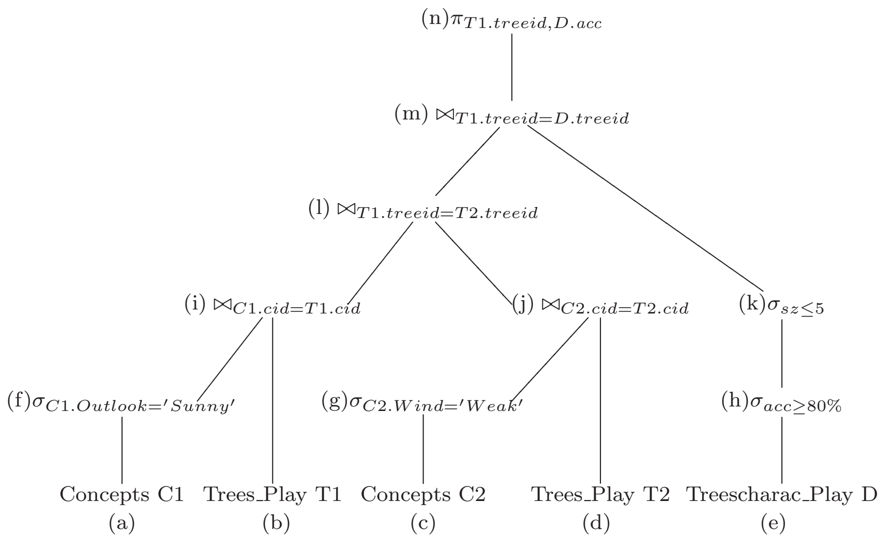

(n)$\pi_{T1.treeid,D.acc}$
(m) $\bowtie_{T1.treeid=D.treeid}$
(l) $\bowtie_{T1.treeid=T2.treeid}$
(i) $\bowtie_{C1.cid=T1.cid}$
(j) $\bowtie_{C2.cid=T2.cid}$
(k)$\sigma_{sz\leq 5}$
(f)$\sigma_{C1.Outlook='Sunny'}$
(g)$\sigma_{C2.Wind='Weak'}$
(h)$\sigma_{acc\geq 80\%}$
Concepts C1
(a)
Trees_Play T1
(b)
Concepts C2
(c)
Trees_Play T2
(d)
Treescharac_Play D
(e)

Fig. 18 An equivalent relational algebra tree for the query in Figure 15.

3.2.3 The redundant operations and the union

The join operation $Q_1 \bowtie_\theta Q_2$ is rewritten as $\sigma_\theta(Q_1 \times Q_2)$, and the intersection $Q_1 \cap Q_2$ as $\pi_{A_1,\ldots,A_n}\sigma_{Q_1.A_1=Q_2.A_1,\ldots,Q_1.A_n=Q_2.A_n}(Q_1 \times Q_2)$, where $A_1,\ldots,A_n$ are the attributes of both queries Q_1 and Q_2.

The union will be handled as follows: every relational algebra expression can be rewritten as $Q_1 \cup \ldots \cup Q_k$, where none of the Q_i contains a union operation. The algorithm will compute annotations for all Q_i separately, and join the tuples needed to answer Q_i together for all $i = 1 \ldots k$.

Example 11 Query Q below is an example query with the union and join operators. It asks for all concepts C, such that either rules $C \rightarrow$ (*Play*=‘Yes’) have confidence of at least 70% or rules $C \rightarrow$ (*Play*=‘No’) have confidence of at least 60%. Also, in both cases, the support of the rules should be at least 5.

$$\begin{aligned}\mathbf{Q:}\ &\pi_{R.cida}((\sigma_{supp\geq 5} Sets S) \bowtie_{S.cid=R.cid}\\ &\quad((\sigma_{conf\geq 60} Rule\ R) \bowtie_{R.cidc=C.cid} (\sigma_{Play='No'} Concepts\ C)\\ &\qquad \cup (\sigma_{conf\geq 70} Rule\ R) \bowtie_{R.cidc=C.cid} (\sigma_{Play='Yes'} Concepts\ C)))\end{aligned}$$

To compute the annotation for Q, the proposed algorithm will first rewrite Q as the following queries:

$$\begin{aligned}\mathbf{Q_1:}\ &\pi_{R.cida}(\sigma_{S.cid=R.cid}((\sigma_{supp\geq 5} Sets\ S)\times\\ &\quad(\sigma_{R.cidc=C.cid}((\sigma_{conf\geq 60} Rule\ R) \times (\sigma_{Play='No'} Concepts\ C)))))\end{aligned}$$

$$\mathbf{Q_2}\text{:}\ \pi_{R.cida}(\sigma_{S.cid=R.cid}((\sigma_{supp\geq 5}Sets\ S)\times (\sigma_{R.cidc=C.cid}((\sigma_{conf\geq 70}Rule\ R)\times(\sigma_{Play='Yes'}Concepts\ C)))))$$

Afterwards, it will compute the annotations for Q_1 and Q_2. Finally, the annotation for Q will be the union of the annotations for Q_1 and Q_2.

$\square$

One important point we would like to stress here is that the rewritings that we apply to the query to reduce the number of operations needed to be translated has no effect afterwards, i.e., on the actual computation of the query in the relational database system; the extracted annotation is used only to fill the virtual tables. After that, the original SQL query is presented to the normal SQL query processor, which will rewrite the query for optimal execution. Such rewriting may be different from the one applied when constructing the annotation.

3.2.4 Leaf Nodes

The sub-query associated with 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 example tree (Figure 18) asks for all tuples from the mining view *Concepts*. Therefore, the annotation for a leaf node is simply the representation of the type of object necessary to form the tuples to be stored in the mining view being queried. Figure 19 shows the annotations for all possible types of leaf nodes that represent mining views. Every annotation also includes a mapping from the attributes of the sub-query associated with the node itself to the originating attributes.

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.

3.2.5 Selection with Predicate Attrθa

This node type will include a constraint into the originating attributes of *Attr* in the annotation of the child node. Consider, e.g., node (f) in the example tree. 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 concepts that contain "Outlook=Sunny". The annotation for this node is therefore constructed by simply including the constraint '="Sunny"' into the originating attribute of C1.Outlook in the annotation of node (a), which is reproduced at the top of Figure 20. The annotation for node (f) is shown at the bottom of the same figure.

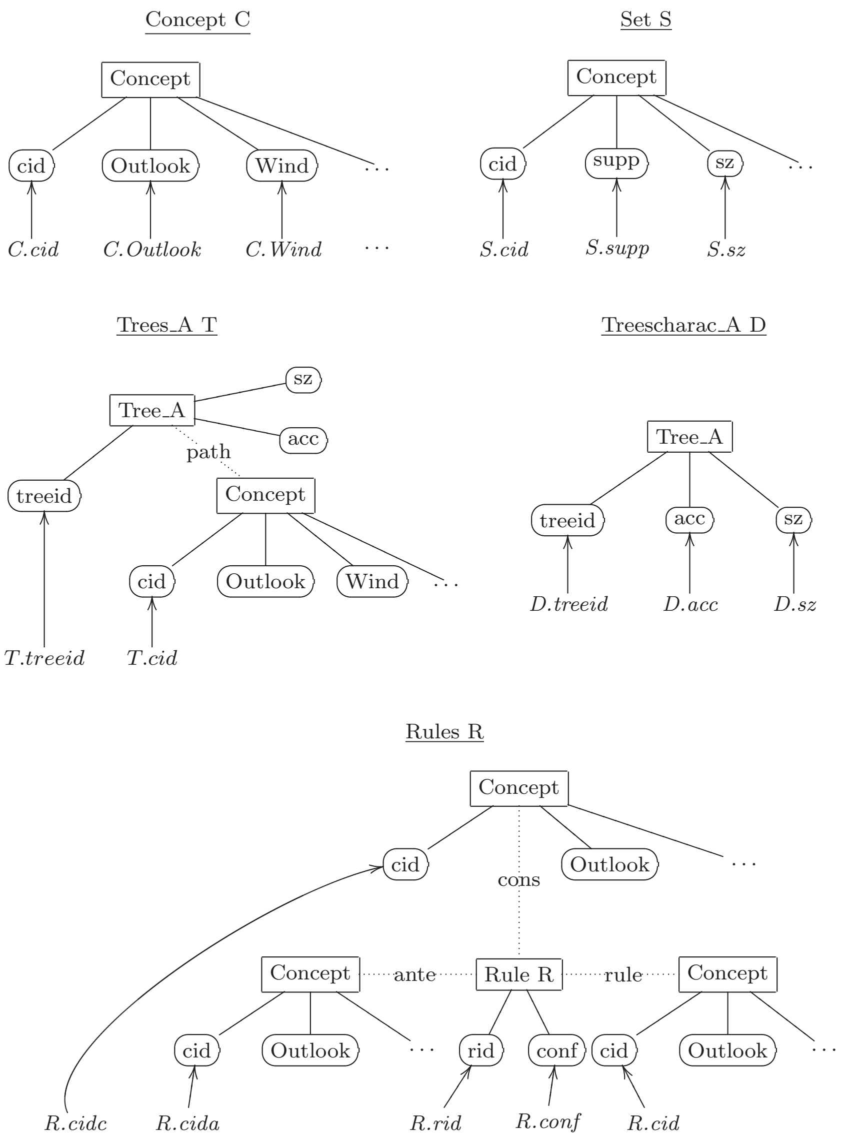


Fig. 19 Annotations for the leaf nodes representing the proposed mining views.

3.2.6 The Cartesian Product

The construction of the annotation for a node of type × consists in simply taking the union of the annotations of its children. This is because to answer

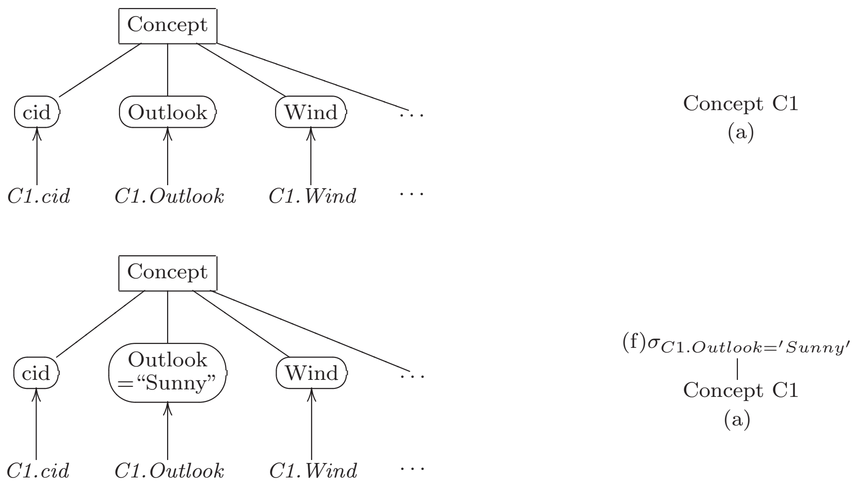


Fig. 20 Annotation and corresponding sub-query of node (a) (top), and the annotation and sub-query of node (f) (bottom).

the query $Q_1 \times Q_2$ correctly, we need everything necessary to compute Q_1 as well as Q_2.

This operation occurs, e.g., when rewriting the join in node (i) in the example tree. 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 for the Cartesian product, illustrated at Figure 21, 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.

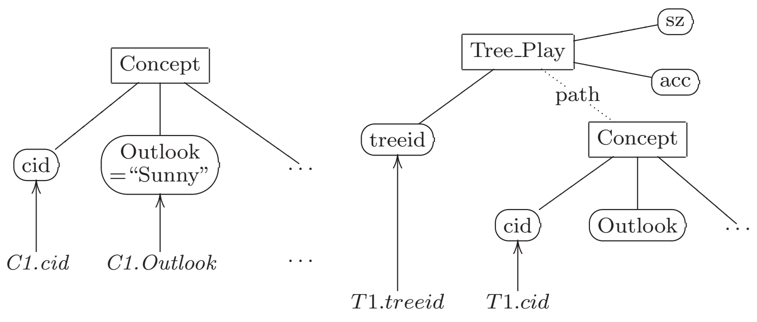


Fig. 21 Annotation for $R_f \times R_b$.

3.2.7 The Selection $\sigma_{Attr_1 \theta Attr_2}$

This is the most involved operation, as it may result in the merge of objects (represented by the rectangles) in the annotation, e.g., when two cid's are made equal. Such a merge may cause a cascade of merges, such as when two rules are merged, resulting in the merge of the antecedents and consequents of the rules as well. Merging objects will result in merging constraints as well.

Consider the selection "C1.cid=T1.cid" in our running example, following the Cartesian product in the rewriting of the join in node (i). From the Cartesian product annotation (Figure 21), 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 Concept objects in these annotations, as depicted in Figure 22. Observe that due to the equi-join operation, the query attributes "C1.cid" and "T1.cid" have now the same originating attribute in this annotation.

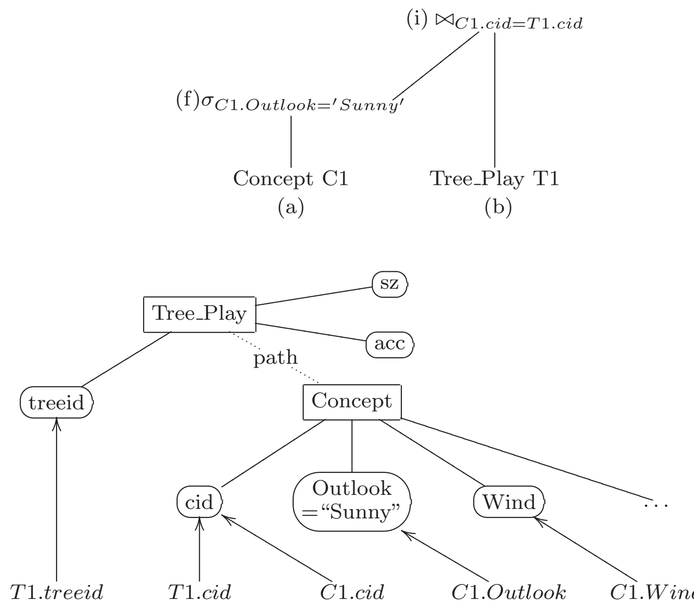


Fig. 22 Annotation for node (i) (bottom) and the corresponding sub-query (top).

Note that if θ is not '=', then the annotation is equal to 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 will be

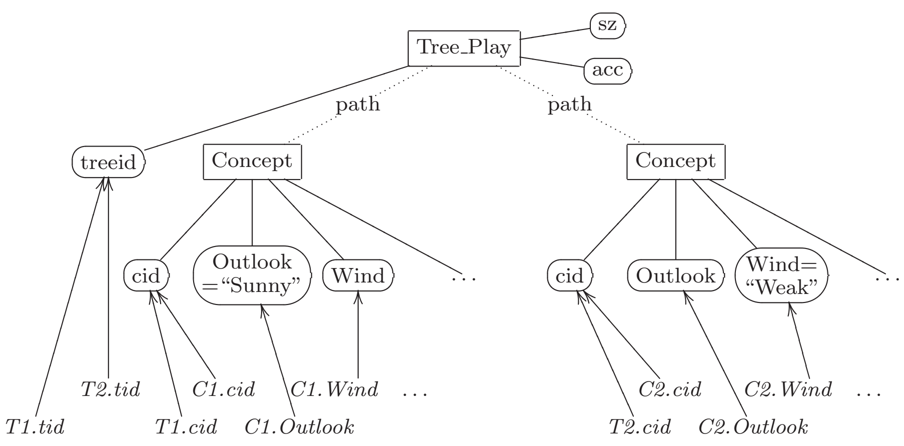


Fig. 23 Annotation for node (l).

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.

Let us now consider another example for the join operation, corresponding to node (l) in the example tree in Figure 18. 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 two tree objects will be merged. The resulting annotation for node (l) is shown in Figure 23. Since the relation between tree and concept is one-to-many, the concepts need not to be merged but become both related to the same tree object. When two rule objects need to be joined, however, their join would result into a join of their respective antecedent, consequent, and rule concepts as well, since the relation between rule and, e.g., antecedent is one-to-one. So, if the two rules turn out to become the same because of a join, their antecedents, consequents, and rule concepts will implicitly become the same as well.

3.2.8 Projection with Attribute List $\mathbf{Attr_1}, \ldots, \mathbf{Attr_k}$

The annotation for the projection operator is trivially constructed from that of the child node; the instantiation does not change as we obviously still need exactly the same mining objects. Only the attribute mapping will change as some attributes are removed. As an example, root node (n) simply projects the tuples coming from node (m) on the attributes $T1.treeid$ and $D.acc$. Its annotation is the same as that for node (m), keeping, however, only the projected query attributes and their originations. The annotation for this node, which is the final annotation for our example query, is shown in Figure 24.

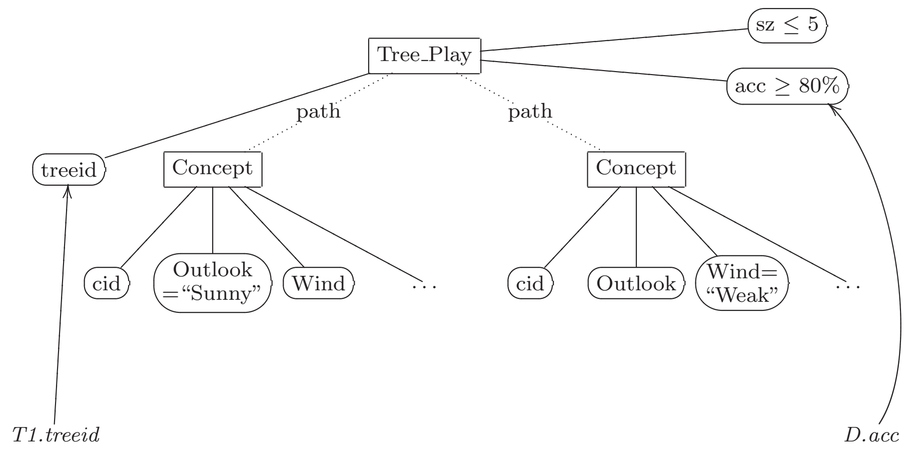


Fig. 24 Annotation for node (n), which is the final annotation for the query in Figure 18.

3.2.9 Operation Set-Difference

The treatment of the set-difference operation is not very involved, yet is somewhat counterintuitive. The reason for this is the non-monotonic nature of the difference operation; sometimes we need to put some objects in the mining views not because they will be part of the output, but rather to prevent other objects from appearing in the output.

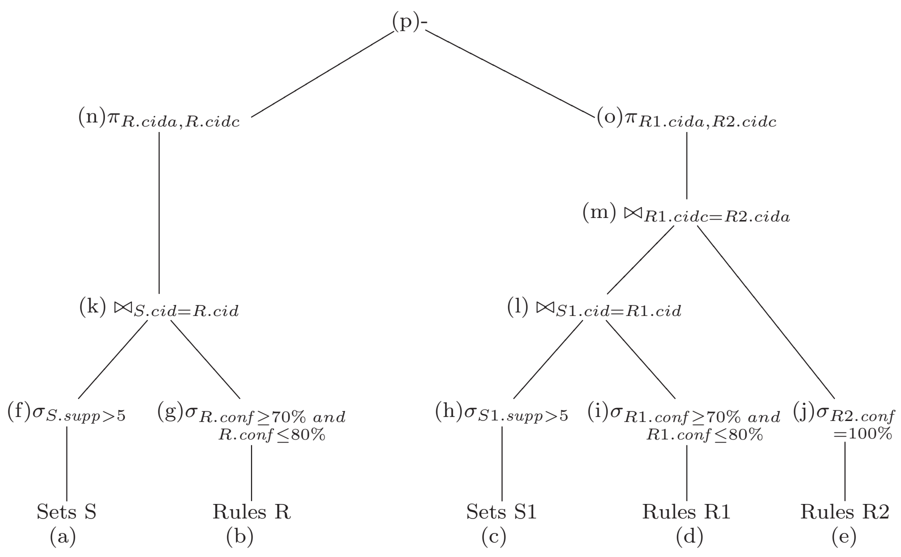
The result of the operation R_1-R_2 is a relation obtained by including all tuples from R_1 that do not appear in R_2.

Example 12 Consider the relational algebra query in Figure 25. In this query, R_1 is the relation produced by node (n), while R_2 is the relation produced by node (o). The query asks for association rules $X \to Y$ with support greater than 5 and confidence between 70% and 80% (node (n)) that are not the result of chaining a rule $X \to Z$ (with the same characteristics of $X \to Y$) and a correct rule (100% confidence) $Z \to Y$ (node (o)).

□

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

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 12, the rules $AB \to D$ and $B \to D$ would be produced incorrectly as part of the query's answer. The annotation for node (p) must



(p)-
(n)$\pi_{R.cida,R.cidc}$
(o)$\pi_{R1.cida,R2.cidc}$
(m) $\bowtie_{R1.cidc=R2.cida}$
(k) $\bowtie_{S.cid=R.cid}$
(l) $\bowtie_{S1.cid=R1.cid}$
(f)$\sigma_{S.supp>5}$
(g)$\sigma_{R.conf\geq 70\%\ and\ R.conf\leq 80\%}$
(h)$\sigma_{S1.supp>5}$
(i)$\sigma_{R1.conf\geq 70\%\ and\ R1.conf\leq 80\%}$
(j)$\sigma_{R2.conf=100\%}$
Sets S (a)
Rules R (b)
Sets S1 (c)
Rules R1 (d)
Rules R2 (e)

Fig. 25 A relational algebra tree with a node of type set-difference (node (p)).

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 query 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 3.2.6) followed by the projection on the query attributes of the left child node. The annotation for the right child node is stripped of its query attributes and added to the annotation for the left child node. The two annotations are disjoint in the resulting one.

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[3] and the rule miner implementation by Bart Goethals[4].
– For decision trees, the system is linked to the exhaustive decision tree learner called Clus-EX, described in detail in [14], which searches for all trees satisfying the constraints "minimum accuracy" and "maximum size". Clus-EX learns decision trees with binary splits only.

As an example, consider the annotation in Figure 24, 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.

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 26. 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).

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

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

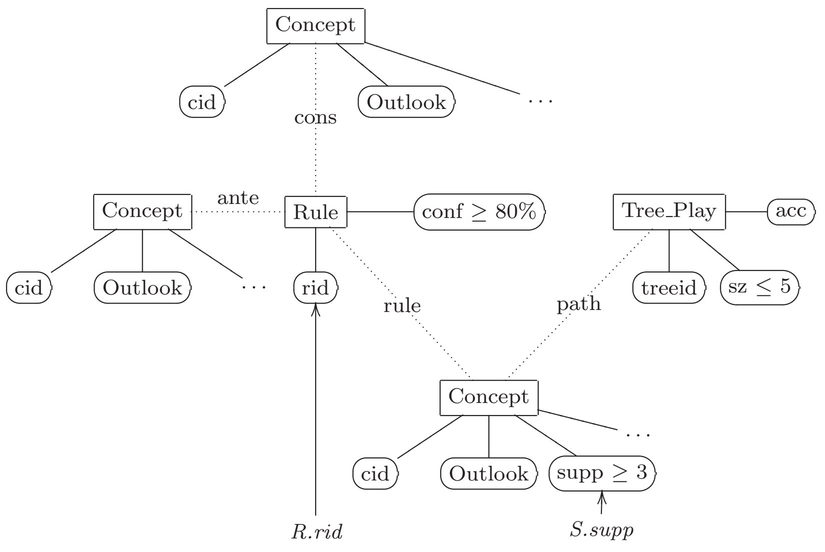


Fig. 26 Example annotation.

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 27. 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 [25]: 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.

5 An Illustrative Scenario

In this section, we describe a data mining scenario that explores the main advantages of our inductive database system. We use the Adult dataset, from the UCI Machine Learning Repository [26], which has census information, and

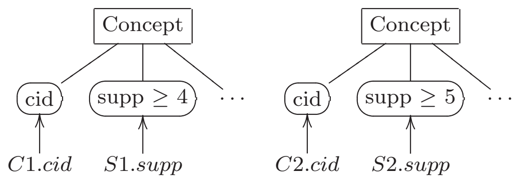


Fig. 27 Example annotation for a case of generation of duplicates.

assume it is already stored in our system. It contains 32,561 tuples, 6 numerical attributes and 8 categorical attributes. An extra attribute called 'class' discriminates people from having a low ('≤50K') or high income ('>50K') a year, while the other attributes describe features such as age, sex, and marital status. The scenario consists in finding an accurate model to predict the attribute 'class' of tuples that refer only to women.

5.1 Step 1: Discretizing Numerical Attributes

We start by discretizing the numerical attributes 'age', 'capital_gain', 'capital_loss', and 'hours_per_week'. For this task, we use the SQL CASE expression, which is available in a variety of database systems (e.g., PostgreSQL, MySQL, and Oracle). As an example of how to use this expression, consider the query in Figure 28. It creates a table called ADULT_CATEG, where the attribute 'age' in table ADULT_FEMALE is discretized into the categories 'Young', 'Middle_aged', 'Senior' and 'Old'.

```
create table adult_categ as
 select case
    when age <= 25 then 'Young'
    when age between 26 and 45 then 'Middle_aged'
    when age between 46 and 65 then 'Senior'
    else 'Old'
  end as age,
  from adult_female
```

Fig. 28 Example discretization query.

We then create table ADULT_CATEG for our scenario, where not only the attribute 'age', but all aforementioned numerical attributes are also discretized (with the SQL CASE expression in Figure 28). The categories used for each attribute are shown in Figure 29 and are inspired by those described in [27]. We also only select a subset of the attributes of the original table: we removed the attributes 'fnlwgt', since it only contains information about the data collection process, and the attribute 'education_num', as it is just a numeric representation of the attribute 'education'.

Attribute	Categories
age	'Young' (0-25), 'Middle_aged' (25-45), 'Senior' (45-65)' and 'Old' (66+)
capital_loss	'None' (0), 'Low' (< median of values > 0), 'High' (≥ median of values > 0)
capital_gain	'None' (0), 'Low' (< median of values > 0), 'High' (≥ median of values > 0)
hours_per_week	'Part_time' (0-25), 'Full_time' (25-40), 'Over_time' (40-60) and 'Too_much' (60+)

Fig. 29 Discretization categories.

5.2 Step 2: Selecting Transactions

Since we want to classify only women, we now create a new table called FEMALE, having only those tuples from table ADULT_CATEG with attribute sex='Female'. Figure 30 shows the corresponding SQL query. In the end, table FEMALE has 10,771 tuples, being 1,179 women with high income and 9,592 women with low income.

```
create table female as
 select age, work_class, education,
        marital_status, occupation,
        relationship, race,
        capital_gain, capital_loss,
        hours_per_week, country,
        class
 from adult_categ
 where sex = 'Female'
```

Fig. 30 Pre-processing query.

5.3 Step 3: Mining Decision Trees with Maximum Accuracy

We now look for decision trees over the new table FEMALE, targeting the attribute 'class' and with maximum accuracy among those trees of size ≤ 5. The query is shown in Figure 31. [5]

Eventually, 27 trees are stored in table BEST_TREES, all of them with accuracy of 91%.

5.4 Step 4: Choosing the Smallest Tree

Having learned the trees with maximum accuracy in the previous step, we now want to evaluate the predictive capacity of these trees. Since all trees have the same accuracy, we choose the smallest one, which is depicted in Figure 32. This tree has *treeid* equal to 238 and the corresponding query for this task is shown in Figure 33.

[5] For the sake of readability, ellipsis were added to some of the SQL queries presented in this section, which represent sequences of attribute names, joins, etc.

```
create table best_trees as
 select t.treeid,
        c.age, ..., c.class
        d.acc, d.sz
from female_trees_class t,
     female_treescharac_class d,
     female_concepts c
where t.cid = c.cid
  and t.treeid = d.treeid
  and d.sz <= 5
  and d.acc=
      (select max(acc)
       from female_treescharac_class
       where sz <=5 )
  order by t.treeid, c.cid
```

Fig. 31 Query over decision trees.

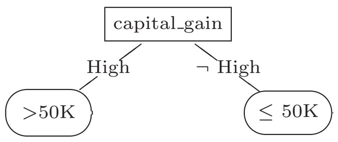


Fig. 32 Decision tree (¬ High = {None, Low})

```
select treeid
from best_trees
where sz = (select min(sz)
            from best_trees)
```

Fig. 33 Query that selects the smallest tree.

5.5 Step 5: Testing the Chosen Decision Tree

We now check the performance of the chosen decision tree. For this, we use the test set from the UCI repository. We start by creating table FEMALE_TEST by applying the same discretization rules and selecting only the female examples (5,421 in total). Next, we check, for each class, the number of misclassified examples w.r.t the selected decision tree. This task is achieved with the query in Figure 34. Here, the equi-join, as explained in Section 2, is executed between tables FEMALE_TEST and BEST_TREES, from which the tree with *treeid* = 238 is selected. The misclassified tuples are those for which the predicted class is different from the real class.

The result of this query reveals that 481 (out of 590) of women with high income are misclassified, while only 9 (out of 4,831) of those with low income are misclassified (490 women are misclassified in total).

Note that the obtained accuracy (91%) is not far from the theoretical baseline value (89%), which is obtained by assigning all tuples to the majority

```
select F.class, count(*)
from female_test F,
     best_trees T
where (F.age = T.age or T.age  = '?')
and (F.work_class = T.work_class or T.work_class  = '?')
and (F.education  = T.education or T.education  = '?')
and ...
and (F.country  = T.country or T.country  = '?')
and F.class <> T.class //misclassified
and T.treeid = 238
group by F.class
```

Fig. 34 Query to compute, for each class, the number of misclassified tuples w.r.t the tree in Figure 32.

class, '≤50K'. In the following, we consider the use of *emerging patterns* [28] to obtain a classifier with a higher accuracy.

5.6 Step 6: Mining Emerging Patterns

In this new step, we evaluate whether emerging patterns would better discriminate the two classes. As initially introduced in [28], emerging patterns (EPs) are patterns whose frequency increases significantly from one class to another and, as such, can be used to build classifiers.

Since a significant number of women with high income are misclassified with the decision tree, we now look for EPs within this class. We start by mining frequent itemsets having support of at least 117 (10%) and keep them in table FEMALE_HI, as shown in Figure 35. We obtain 1,439 itemsets.

```
create table female_hi as
 select C.*, S.supp
 from female_Sets S,
      female_Concepts C
  where C.cid  = S.cid
    and S.supp  >= 117
   and C.class = '>50K'
```

Fig. 35 Query over frequent itemsets within class '>50K' (high income).

Then, we compute the support of each of these itemsets in the other class, that is, among women with low income. The corresponding query is depicted in Figure 36: for each itemset i in table FEMALE_HI, it computes the number of tuples coming from women with low income (attribute 'class' = '≤50K') that satisfies i.

Finally, we store in table EMERGING_PATTERNS only the EPs, that is, those itemsets whose relative support within class '>50K' is at least 15 times higher

```
create table female_li as
 select HI.age, ..., HI.native_country,
        F.class, count(*) as supp
 from female_hi HI,
      female F
 where F.class = '<=50K'
   and (F.age = HI.age or HI.age = '?')
   and (F.work_class = HI.work_class or HI.work_class = '?')
   and (F.education  = HI.education  or HI.education = '?')
   ...
   and (F.country = HI.country or HI.country = '?')
  group by HI.age, ..., HI.country, F.class;
```

Fig. 36 Computing the support of itemsets selected with the query in Figure 35 within class '≤50K'.

than within class '≤50K'. This task is accomplished with the query in Figure 37. In total, we find 196 EPs.

```
create table emerging_patterns as
 select HI.*
 from female_hi HI, female_li LI
 where (HI.age = LI.age)
   and (HI.work_class = LI.work_class)
   and (HI.education  = LI.education)
   ...
   and (HI.country = LI.country)
   and ((HI.supp/LI.supp) * (9592/1179)) >= 15
```

Fig. 37 Selecting the emerging patterns with respect to class '>50K' (high income).

5.7 Step 7: Classification based on EPs

In this last step, we evaluate the predictive capacity of the EPs in table EMERGING_PATTERNS. The idea is to assign each woman to the low income class ('≤50K'), except those that satisfy at least one EP. The classification accuracy is thus computed as follows: women with high income in table FEMALE_TEST that satisfy at least one EP in table EMERGING_PATTERNS are considered well-classified. Conversely, women with low income that satisfy at least one EP are considered misclassified. The query in Figure 38 computes the number of well-classified (query before the 'union') and misclassified (query after the 'union') women according to this idea.

The result of this query reveals that 300 women (out of 590) with high income are well-classified and that 121 with low income are misclassified. We therefore conclude that with the EPs 121 + (590 - 300) = 411 women are misclassified in total, while the selected decision tree leads to 490 misclassifications.

```
alter table female add column id serial;

select count (distinct F.id)
from female_test F,
     emerging_patterns EP
where (F.age = HI.age or EP.age = '?')
   and (F.work_class = EP.work_class or EP.work_class = '?')
   and (F.education = EP.education  or EP.education = '?')
   ...
   and (F.country = EP.country or EP.country = '?')
   and F.class = EP.class //well-classified
union
select count (distinct F.id)
from female_test F,
     emerging_patterns EP
where (F.age = HI.age or EP.age = '?')
   and (F.work_class = EP.work_class or EP.work_class = '?')
   and (F.education = EP.education  or EP.education = '?')
   ...
   and (F.country = EP.country or EP.country = '?')
   and F.class <> EP.class //misclassified
```

Fig. 38 Computing the number of well-classified and misclassified women w.r.t the selected EPs.

This ends the data mining scenario. We demonstrated that the main defining principles of an inductive database are fully supported by the mining views framework. It is clear, for instance, that the closure principle is straightforwardly supported, as shown by the queries that create new tables with the results of mining queries of previous steps. The new tables can then be further queried with the same language used to perform the original mining tasks, i.e., SQL. The support for ad-hoc constraints was demonstrated in all queries, in which no pre-planning was necessary. In other words, the user can simply think of a new query in SQL and immediately write it using the mining views or the new tables that were created as a result of the mining operations.

5.8 Performance Counters

We now investigate the efficiency and effectiveness of our inductive database system. We conducted a set of experiments over 2 SQL queries executed for the scenario presented above. The chosen queries are those illustrated in Figure 39.

Query (I) asks for decision trees with maximum accuracy among those trees of size of at most 5, while query (J) asks for frequent itemsets. It is worth noticing that the source data for both queries had in total 10,771 tuples and 12 attributes (including the attribute 'class').

(I)

```
create table best_trees as
 select t.treeid,
        c.age, ..., c.class
        d.acc, d.sz
 from female_trees_class t,
      female_treescharac_class d,
      female_concepts c
 where t.cid = c.cid
   and t.treeid = d.treeid
   and d.sz <= 5
   and d.acc=
      (select max(acc)
       from female_treescharac_class
       where sz <=5 )
 order by t.treeid, c.cid
```

(J)

```
create table female_hi as
  select C.*, S.supp
  from female_Sets S,
       female_Concepts C
  where C.cid  = S.cid
    and S.supp  >= 117
    and C.class = '>50K'
```

Fig. 39 The queries chosen for the experiments.

5.8.1 Intermediate Results

We begin by evaluating the intermediate results that were computed by the mining algorithms along with the results that were produced as output to the user. In Table 1, we show the number of intermediately generated concepts, itemsets (when applicable), decision trees (when applicable), and the size of the output table (in tuples) for each of the example queries.

Query	#Concepts	#Itemsets	#Trees	Output (tuples)
(I) - decision trees	5,100	n/a	241	559
(J) - frequent itemsets	1,439	1,439	n/a	1,439

Table 1 Number of concepts, itemsets or decision trees computed by queries (I) and (J) in Figure 39, along with the number of tuples in their output.

Decision Trees Observe that query (I) has a nested query (sub-query) in its where-clause, which computes the maximum accuracy of the decision trees predicting the attribute 'class' with size of at most 5. Since this type of queries can not be translated into relational algebra [29], to extract constraints from this query, the system firstly decomposed it into two *query blocks* (SQL queries with no nesting), in the same way as typical query optimizers do [29]. Afterwards, each query block was translated into a relational algebra tree. The algorithm described in Section 3 was then applied to both expression trees in isolation, and, in the end, the final annotation was the union of the annotations that were constructed for both expression trees.

From the final annotation, the constraint (sz $\leq$ 5) on decision trees was extracted. Afterwards, the mining views *Female_Trees_Class*, *Female_Treescharac_Class*, and *Female_Concepts* were materialized with the results obtained by the algorithm Clus-EX, which exploited the constraint above. The results of the data mining phase consisted of 241 trees (having 5, 100 concepts in total) with a size of at most 5. Notice that, due to the decomposition of the query, the constraint "max(acc)" itself was not extracted by the constraint extraction algorithm and thus not exploited by any data mining algorithm. Nevertheless, the query was correctly computed, since the DBMS considered it anyway to filter the results before showing them to the user. Among the 241 trees generated, only the 27 trees with maximal accuracy (91%) were stored into table BEST_TREES. They had 559 concepts in total.

Frequent Itemsets Regarding the query over association rules, (J), the constraint (supp $\geq$ 117 $\wedge$ class = '>50K') on itemsets was extracted by the constraint extraction algorithm and exploited by the Apriori-like implementation. *Female_Sets* and *Female_Concepts* were then materialized with the obtained results, as well as table FEMALE_HI.

5.8.2 Execution Times

Finally, we also evaluate the total execution time for both queries. Table 2 presents, for each query, its total execution time (in seconds), the time spent by the constraint extraction algorithm (in seconds), and the time consumed by the data mining algorithms plus the time for the materialization of the required mining views (in seconds). The times shown are the average of 3 executions of the queries.

Query	Total Time (in sec.)	CEA (in sec.)	Mining + materialization (in sec.)
(I) - decision trees	23.86	0.01	21.63
(J) - frequent itemsets	1.51	0.01	0.70

Table 2 Execution times for queries (I) and (J) in Figure 39.

Observe that the time spent by the constraint extraction algorithm was negligible for both queries. The total execution time consists mostly of the time consumed by the data mining algorithms along with the materialization of the required virtual mining views. The difference between the total execution time and the mining time was due to the time spent by the system to produce the results as output to the user.

As can be seen, the total execution times were low for both queries, which shows the usefulness and elegance of the proposed inductive database system. Additionally, the experiments demonstrated that the constraint extraction algorithm does not add any extra cost, in terms of execution time, to the query evaluation process. Indeed, the constraint extraction algorithm can be performed in time linearly proportional to the size of the query, whereas query evaluation can take exponential time w.r.t. the size of the query (and polynomial time w.r.t. the size of the database).

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.

6.1 SQL-based proposals

The system SINDBAD (structured inductive database development), developed by Wicker et al. [8], 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. [10] 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 the particular case of SIQL, the attention is not focused on the use of constraints. As we show in [11], the minimum support constraint is not used within the queries themselves, but needs to be configured beforehand with the use of the so-called configure-clause. Constraints are therefore more closely related to a function parameter than to a constraint itself. Additionally, the number of such parameters is limited to the number foreseen at the time of implementation. For example, in the case of frequent itemset mining, the minimum support is the only constraint considered in their system.

Regarding SPQL, the number of constraints that the user can use within their mining queries is also fixed. Consider the task of finding itemsets with large area, discussed in Example 4. Based on its description in [10], this task cannot be expressed in SPQL (nor in SIQL), while being naturally expressed with the mining views. Even though this natural expressiveness is achieved thanks to the addition of the attributes *sz* and *supp* to the mining views *T_Sets*, without these attributes, the constraint could still be expressed in plain SQL (at the cost of writing more complicated queries, as shown in Example 3). With these other languages, this would only be possible with the extension of the language itself, which is considered a drawback here, or with post-processing queries, if possible at all.

In *Microsoft's Data Mining extension (DMX)* of SQL server [7], a classification model 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 [16], for example, we present a scenario in which we learn a classifier having the maximum accuracy and, afterwards, we look for correct association rules, which describes the misclassified examples w.r.t this classifier. 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. Furthermore, there is no notion of composing mining operations in their framework.

For a more detailed comparison between other data mining query languages and the mining views approach, we refer the reader to [11,12].

6.2 Programming language-based approaches

An acronym for "Aggregate & Table Language and System" [6], 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, the authors show the code for the Apriori algorithm, which becomes considerably complex when implemented with the proposed 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.

Another approach with the same line of reasoning is that presented in [30]. This language, however, is not focused on relational databases, but on deductive databases.

In [31], 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. [32]. 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 [31] 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 Discussion

7.1 Expressiveness of the Mining Queries

The expressiveness of queries over the mining views is the same as the expressiveness of querying any relational table with SQL. Despite not being able to express everything (e.g., recursive queries), SQL is a full-fledged query language, which allows ad-hoc constraints and guarantees the closure principle. As specified in [1], these are the main characteristics an inductive database should have. [1] also lists the reasons why this is important. Indeed, the scenario in Section 5 shows the benefit of such characteristics with SQL queries over the mining views.

The mining views framework can express all fundamental constructions of the other data mining query languages. Some types of queries are not as naturally expressed (with the mining views proposed in this paper) as in other languages, but they can easily be handled with the addition of new mining views or even by just pre-processing the data being mined. For example, if the user wants to mine a market basket dataset, where the transactions do not necessarily have the same size, one would need to firstly pre-process the dataset, by creating a new table in which each transaction is represented as a tuple with as many binary attributes (e.g., 'true' or 'false') as are the possible items that can be bought by a customer.

Queries on the structure of the decision trees are not easily expressed either. Indeed, the framework proposed in this article focuses on the representation of the semantics of the trees (the function they represent) rather than on their structure, as in [13,14] [6]. One advantage of the proposed framework in comparison with those in [13,14] is that certain operations, such as using the decision tree for prediction, become straightforward (it is just a join operation). Conversely, queries about the structure, such as asking which attribute is at the root of a decision tree, become cumbersome. An alternative would be to add to the framework separate mining views describing the structure of the decision trees, according to the preferences of the user. Including characteristics of the patterns as attributes may simplify the formulation of constraints as queries (e.g., Examples 3 and 4, Section 2). The schema presented in this article is in fact one possible instantiation of the proposed framework; the mining views with the characteristics of the patterns can always be extended with other characteristics, even if redundant.

[6] Although, through the use of wild-cards, some information about the model structure is still available; for example, the attribute at the root would never have a wild-card value.

7.2 Extraction of Constraints

The proposed constraint extraction algorithm is currently able to extract the following types of constraints:

– **Structural**: minimal and maximal size of itemsets, association rules as well as their components (antecedent and consequent), or decision trees.
– **Syntactic**: a certain attribute-value pair must appear in an itemset, the antecedent of a rule, consequent of a rule, or a decision tree.
– **Interestingness measures**: minimum or maximum support, minimum or maximum confidence, and minimum or maximum accuracy.
– Conjunctions of the constraints detailed above.

Note, however, that any constraint expressible in SQL can be used with our system, even if it is not explicitly extracted by the constraint extraction algorithm. Those that are not yet recognized may affect the efficiency of the system, but not its correctness, since the DBMS will consider them anyway to filter the results before showing them to the user. An example is the "maximum accuracy" used in Example 5, Section 2. The main reason for not recognizing such constraint is that the constructions for expressing it are non-trivial in SQL, requiring sub-queries and aggregations. In any case, Section 8 briefly discusses a possible strategy to extract constraints such as the "maximum accuracy".

There could also be the situation in which constraints extracted by the system are not yet exploited by the current available data mining algorithms, e.g., the algorithm Clus-EX, which is not capable of exploiting syntactic constraints. In this case, the system may store more decision trees than those necessary to answer a given query. Again, this is not a fundamental problem, since the non-exploited constraints are used by the DBMS to filter the query's results.

7.3 Extending the Mining Views Framework With Other Types of Patterns

Some data mining tasks that can be performed in SIQL and DMX, such as clustering, cannot currently be executed with the proposed mining views. On the other hand, note that one could always extend the framework by defining new mining views representing different patterns. More specifically, to extend the proposed system with a new kind of pattern, we first need to specify the schema of the relational tables (or mining views) that will represent the complete output of such a pattern mining task.

In addition to the tables, the pattern mining algorithm to be triggered by the DBMS must also be specified, considering the exhaustiveness nature of the queries the users are allowed to write. Such algorithm should also be, in the best case, able to explore as many constraints as possible. However, as already pointed out in this section, the non-exploitation of constraints does not affect the effectiveness of the system. Once the tables and data mining algorithm are defined, the constraint extraction algorithm needs to be adapted to consider any new structural or syntactic constraints, as well as new interestingness measures.

8 Conclusions

In this article, we described a practical inductive database system 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. 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, as we showed with the scenario in Section 5.
– **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 Section 5 we showed how *emerging patterns* [28] can be extracted from a given table T with an SQL query over the available mining views *Concepts* and *Sets*.
– **Flexibility to specify ad-hoc constraints**: the proposed system is meant to offer exactly this flexibility, i.e., by virtue of a full-fledged query language that allows ad-hoc querying, the user can think of new constraints that were not considered at the time of implementation. A simple example is the constraint *area*, which can naturally be computed with the framework.
– **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 four 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. [33], 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. One direction we want to explore in this perspective is dynamic optimization, where the result of one part of the query or of a sub-query can be used to constrain another part. Third, to mine datasets in the context of, e.g., market basket using the current system, one would need to first pre-process the dataset that is to be mined, by changing its representation. Since this pre-processing step may be laborious, an interesting direction for future work would then be to investigate how this type of datasets could be treated in a easier way by the system. Finally, the system developed so far covers only itemset mining, association rules and decision trees. Another direction for further work is to extend it with other models, considering the exhaustiveness nature of the queries the users are allowed to write.

9 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. 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)
3. Meo, R., Psaila, G., Ceri, S.: An extension to sql for mining association rules. Data Mining and Knowledge Discovery **2**(2) (1998) 195–224
4. Imielinski, T., Virmani, A.: Msql: A query language for database mining. Data Mining Knowledge Discovery **3**(4) (1999) 373–408
5. Wang, H., Zaniolo, C.: Nonmonotonic reasoning in ldl++. Logic-based artificial intelligence (2001) 523–544
6. Wang, H., Zaniolo, C.: Atlas: A native extension of sql for data mining. In: Proc. SIAM Int. Conf. on Data Mining. (2003) 130–144
7. Tang, Z.H., MacLennan, J.: Data Mining with SQL Server 2005. John Wiley & Sons (2005)
8. 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
9. 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
10. Bonchi, F., Giannotti, F., Lucchese, C., Orlando, S., Perego, R., Trasarti, R.: A constraint-based querying system for exploratory pattern discovery. Information Systems **34**(1) (2009) 3–27

11. Blockeel, H., Calders, T., Fromont, E., Goethals, B., Prado, A., Robardet, C.: Practical comparative study of data mining query languages. In: Inductive Databases and Queries: Constraint-based Data Mining. Volume in press. Springer (2010)
12. Blockeel, H., Calders, T., Fromont, E., Goethals, B., Prado, A., Robardet, C.: Inductive querying with virtual mining views. In: Inductive Databases and Queries: Constraint-based Data Mining. Volume in press. Springer (2010)
13. 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
14. 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
15. 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) 1608–1611
16. 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)
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. Chen, P.P.: The entity-relationship model: Toward a unified view of data. ACM Transactions on Database Systems **1** (1976) 9–36
19. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
20. 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
21. Geerts, F., Goethals, B., Mielikäinen, T.: Tiling databases. In: Discovery Science. Volume 3245., Springer (2004) 278–289
22. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
23. Garcia-Molina, H., Widom, J., Ullman, J.D.: Database System Implementation. Prentice-Hall, Inc. (1999)
24. Prado, A.: An Inductive Database System Based on Virtual Mining Views. PhD thesis, University of Antwerp, Belgium (December 2009)
25. 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
26. Newman, D., Hettich, S., Blake, C., Merz, C.: UCI repository of machine learning databases (1998)
27. Hahsler, M., Grün, B., Hornik, K.: arules: Mining association rules and frequent itemsets. SIGKDD Explorations **2** (2007) 0–4
28. Dong, G., Li, J.: Efficient mining of emerging patterns: Discovering trends and differences. In: Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery in Databases. (1999) 43–52
29. Ramakrishnan, R., Gehrke, J.: Database Management Systems. 3 edn. McGraw-Hill Science/Engineering/Math (2002)
30. Giannotti, F., Manco, G., Turini, F.: Specifying mining algorithms with iterative user-defined aggregates. IEEE Transactions on Knowledge and Data Engeneering **16** (2004) 1232–1246
31. 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
32. 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
33. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing data cubes efficiently. In: Proc. ACM SIGMOD Int. Conf. on Management of Data. (1996) 205–216