Integrating Pattern Mining in Relational Databases

Toon Calders, Bart Goethals, and Adriana Prado

University of Antwerp, Belgium
{toon.calders, bart.goethals, adriana.prado}@ua.ac.be

Abstract. Almost a decade ago, Imielinski and Mannila introduced the notion of *Inductive Databases* to manage KDD applications just as DBMSs successfully manage business applications. The goal is to follow one of the key DBMS paradigms: building optimizing compilers for ad hoc queries. During the past decade, several researchers proposed extensions to the popular relational query language, SQL, in order to express such mining queries. In this paper, we propose a completely different and new approach, which extends the DBMS itself, not the query language, and integrates the mining algorithms into the database query optimizer. To this end, we introduce *virtual mining views*, which can be queried as if they were traditional relational tables (or views). Every time the database system accesses one of these virtual mining views, a mining algorithm is triggered to materialize all tuples needed to answer the query. We show how this can be done effectively for the popular association rule and frequent set mining problems.

1 Introduction

Almost a decade ago, Imielinski and Mannila [9] introduced the concept of an *Inductive Database*, in which a *Knowledge and Data Discovery Management System* (KDDMS) manages KDD applications just as DBMSs successfully manage business applications. Generally speaking, besides allowing the user to query the data, the KDDMS should also give users the ability to query patterns and models extracted from these data. In this context, several researchers proposed extensions to the popular relational query language, SQL, as a natural way to express such mining queries [8, 10, 11].

In our work, we aim at extending the DBMS itself, not the query language. That is, we propose an approach in which the user can query the collection of all possible patterns as if these are stored in relational tables. The main challenge is how this storage can be implemented effectively. After all, the amount of all possible patterns can be extremely large, and impractical to store. For example, in the concrete case of itemsets, an exponential number of itemsets would need to be stored. To resolve this problem, we propose to keep these pattern tables virtual. That is, as far as the user is concerned, all possible patterns are stored, but on the physical layer, no such complete tables exist. Instead, whenever the user queries such a pattern table, or *virtual mining view*, an efficient

data mining algorithm is triggered by the DBMS, which materializes at least those tuples needed to answer the query. Afterwards, the query can be executed as if the patterns had been there before. Of course, this assumes the user poses certain constraints in his query, asking for only a subset of all possible patterns, which should then be detected and exploited by the data mining algorithm. As a first step towards this goal, we propose such a constraint extraction procedure starting from a collection of simple constraints.

Notice that the user can now query mining results by using a standard relational query language, such as SQL. Furthermore, the user does not need to deal with the mining algorithms themselves as these are transparently triggered by the DBMS. We show how this approach can be implemented for the popular association rule and frequent set mining problems.

2 Related Work

The idea to integrate data mining into databases has been addressed in a number of works [8–11]. Some of these works focus on extensions of SQL, as a natural way to give the user the ability to mine the data. For example, the query language *DMQL* was proposed by Han *et. al.* for mining relational databases [8]. *DMQL* adopts an SQL-like syntax for mining different kinds of rules such as classification rules and association rules. Another example, is the *MINE RULE* operator proposed by Meo *et. al.* [11], designed as an extension of the SQL language. This operator was proposed specifically for association rule mining discovery. The *MSQL* language [10], proposed by Imielinski and Virmani, is also focussed on association rules. It extends SQL with the operators *GetRules* and *SelectRules* that can, respectively, generate and query a set of association rules. Other examples of query languages for data mining are $\mathcal{LDL}^{++}$ [13] and ATLaS [14] of Wang and Zaniolo, which are extensions of $\mathcal{LDL}$ and SQL, respectively. A more theoretical study of a data mining query language is the data mining algebra proposed by Calders *et. al.* [3].

3 Virtual Mining Views

An association rule, defined over a set of items $\mathcal{I}$,is an implication of the form $X \Rightarrow Y$, where $X, Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. We say that X is the antecedent and Y the consequent of the rule [2]. Let D be a collection of transactions, where each transaction is a subset of $\mathcal{I}$. Then, the rule $X \Rightarrow Y$ holds in D with support s, if s transactions in D contain $X \cup Y$, and confidence c if $c\%$ of the transactions in D that contain X also contain Y.

The transaction database D can be stored as a binary relational data table. For each transaction, there is a set of tuples of the form $(tid, item)$, where tid is the transaction identifier and $item$ is an item in that transaction. Note that this table can also be implemented as a database view on the real data, as this is typically not represented in such a binary relation. Then, the association rules,

generated from the mining of D, can be represented in the same database where D is stored, by using the following schema.

1. $Sets(sid, item)$: This table represents all itemsets. A unique identifier sid is associated to each itemset and, $item$ is an item in that itemset.
2. $Supports(sid, supp)$: This table represents the supports of the itemsets in the $Sets$ table. For each itemset, there is a tuple where sid is the identifier of the itemset and $supp$ the support.
3. $Rules(rid, sida, sidc, sid, conf)$: This table represents all association rules. For each rule $X \Rightarrow Y$, there is a tuple with an association rule identifier rid, the antecedent identifier $sida$, the consequent identifier $sidc$, the set identifier sid of the complete itemset $(X \cup Y)$, and $conf$ the confidence.

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 set identifiers for an association rule, sid, $sida$ or $sidc$, is redundant, as it can be determined from the other two. Nevertheless, it would also imply the user would have to write much more complicated queries. Essentially, only the data table D and the $Sets$ table are necessary in order to be able to select itemsets or association rules in a single query. Indeed, without the $Sets$ table, this is impossible because such a query explicitly generates the powerset of $\mathcal{I}$, which is well known to be impossible in SQL [1].

It is of course also still possible to add more attributes to these tables in which, for example, also other interestingness measures could be stored.

The main goal of the proposed pattern tables is to give the user the ability to query data mining results in the same way as traditional relational tables are queried. As already explained, however, it is intractable to store all $2^{|\mathcal{I}|}$ itemsets or $3^{|\mathcal{I}|}$ association rules. On the other hand, the entire set of patterns does not always need to be stored, but only the patterns that satisfy the constraints within the user's query (e.g., minimum support, minimum confidence, etc.). Therefore, in our proposal, the pattern tables are actually empty, and after the user has posed his query, the necessary patterns are materialized by the mining algorithm, immediately before the DBMS answers the query. Of course, this means that the DBMS should be able to detect the constraints in the given query. In our approach, we extract such constraints from a relational algebra expression equivalent to the user's query. Note that every SQL-query can easily be transformed into an equivalent relational algebra expression [7]. Such a relational algebra expression has the advantage of being procedural as compared to SQL, which is declarative, making the constraint extraction easier.

4 Extracting Constraints from Queries

For reasons of simplicity, we study a restricted class of constraints: for association rules we have minimal and maximal support, minimal and maximal confidence, plus the constraints that a certain item must be in the antecedent, in the consequent, in the antecedent or the consequent, and Boolean combinations of these.

The stricter the constraints we can extract, the more efficient query evaluation will become, because the number of tuples that needs to be materialized decreases when the extracted constraint is stricter. Note, however, that extracting the best possible constraint, i.e. the most strict, is theoretically impossible, even in the restricted case considered here. Indeed; suppose, for the sake of contradiction, that an algorithm exists that always extracts the best possible constraints. Then, this algorithm allows us to decide whether an SQL-query will always return an empty answer as follows: given a query q, run the assumed algorithm on the following query q': **select** *sid* **from** *Sets* **where** not exists (q); Obviously, on the one hand, if q always returns the empty answer, q' will never return any *sid*. In that case, the most strict constraint is "false"; i.e., no itemsets nor rules should be mined. On the other hand, if q is not empty, q' will produce the *sid* of every itemset, and hence, the constraint must be "true". Therefore, q is non-empty if and only if the assumed algorithm returns the constraint "true". Deciding non-emptiness of SQL-queries, however, is well-known to be undecidable [1]. Therefore, the algorithm proposed in this section is necessarily incomplete. For simple queries, however, it will find strict constraints, as will be illustrated with some examples.

As already explained earlier, the proposed constraint extraction algorithm does not work directly on the SQL-query, but on an equivalent relational algebra expression, which can be easily generated by existing algorithms.

Relational Algebra A relational algebra expression is a sequence of set operations on the relations, resulting in the answer of the query. Consider, for example, the SQL-query shown in Fig. 1. The query asks for the rules and their confidences, that have the item *apple* in the antecedent or consequent of the rule, support of at least 40 and confidence of at least 80%. An equivalent relational algebra expression for this SQL-query is

$$\pi_{R.rid,R.conf}\ \sigma_{S_2.sid=R.sid}\ \big(\sigma_{S_1.sid=S_2.sid}\ (\sigma_{S_1.item=apple}\ \mathit{Sets}) \times (\sigma_{S_2.supp\geq 40}\ \mathit{Supports})\big) \times (\sigma_{R.conf\geq 80\%}\ \mathit{Rules})$$

In this expression, $\sigma_{R.conf\geq 80\%}\ \mathit{Rules}$ expresses that we only select those tuples from the relation *Rules* that satisfy the constraint $R.conf \geq 80\%$, $\times$ constructs the Cartesian product of two relations, i.e., for every tuple of the first relation, and every tuple of the second relation, a new tuple that is the concatenation of the two tuples is in the result relation of $\times$. $\pi_{R.rid,R.conf}$ produces a new relation that has only the attributes $R.rid$ and $R.conf$.

Notice that this expression can also be represented by its syntax tree, as is illustrated in Fig. 1. For the sake of simplicity, we will continue working with such expression trees instead of the relational algebra expressions themselves.

Algorithm Given a query q as input, first, an equivalent tree of relational algebra is constructed. As every leaf node in this tree represents a table or a virtual mining view, the goal of the algorithm is to find which tuples should be present in

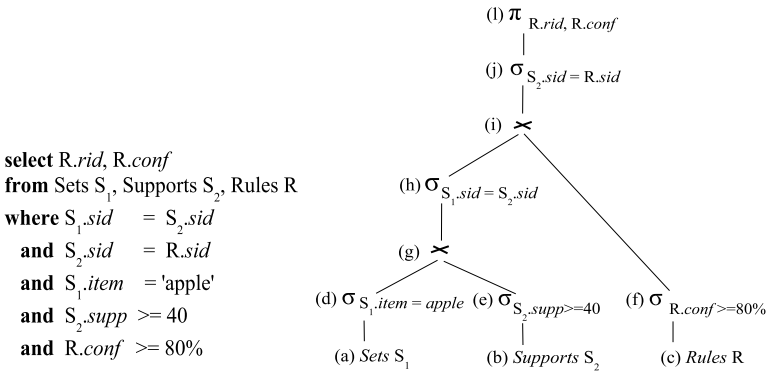


Fig. 1. An example query and its corresponding expression tree of relational algebra.

those nodes representing a virtual mining view, in order to answer to the query. Thus, actually, the algorithm determines, for each of the aforementioned nodes, a constraint. For example, in Fig. 1, there are three leaf nodes that represent virtual mining views: one with the *Sets* view, one with the *Supports* view, and one with the *Rules* view. The goal of the algorithm is then to identify that, in order to answer the query, it suffices to have in (a) and (b) only tuples that come from itemsets having the item *apple* and with support of at least 40, and in (c) only tuples that come from association rules with confidence of at least 80% and generated from the same collection of itemsets present in (a) and (b). We denote the subset of tuples of a virtual mining view V that come from itemsets that satisfy ϕ by $V[\phi]$. E.g., the subset of *Sets* needed to be materialized for node (a) can be denoted $Sets[apple \wedge supp \geq 40]$.

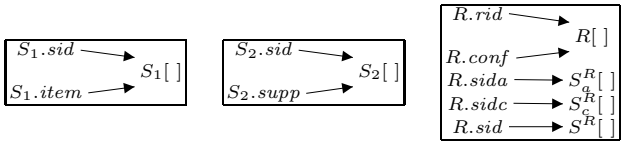
Notice that because leaf nodes always correspond to the tables or virtual mining views in the from-clause of the query, the algorithm finds, for every virtual mining view in the from-clause, a constraint. If one virtual mining view is used multiple times, multiple constraints will be extracted. This is not a problem, as we can easily combine these different constraints by taking the union.

In this context, the procedure to extract the constraints on the virtual mining views is as follows. Starting from the leaves, going bottom-up, the algorithm determines for every node n in the expression tree which tuples should be in the virtual mining views in order to answer the *subquery* associated with that node, that is, the query represented by the subtree rooted at n. To answer the subqueries associated with the leaf nodes themselves, obviously, all tuples should be in, since the subquery is essentially asking for all tuples in the virtual mining view. Going up, however, it becomes clear that, in fact, not the complete virtual mining view is needed. E.g., in Fig. 1, for node (a), all tuples of *Sets* are needed. When going up to node (d), however, we see that only those tuples satisfying ($item = apple$) are needed. Henceforth, to answer the subquery rooted at (d), it

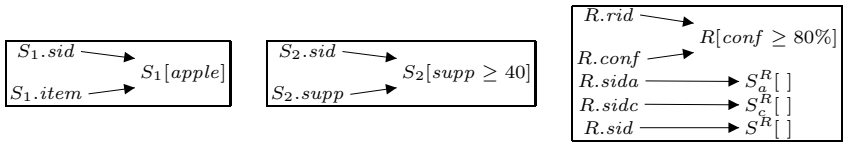
suffices that the virtual mining view *Sets* only contains those tuples that come from itemsets having the item *apple*, that is, only $Sets[apple]$ is needed.

In the computation of the constraints on the virtual mining views, for every node n, we annotate every node with a three-tuple $(\mathcal{A}, \mathcal{V}, \mathcal{O})$. In this three-tuple, $\mathcal{A}$ is the set of attributes $\{A_1, \ldots, A_n\}$ of that node, $\mathcal{V}$ is the set of virtual views with the constraints for n $\{V_1[\phi_1], V_2[\phi_2], \ldots, V_m[\phi_m]\}$, and $\mathcal{O}$ a set of pairs $A_i \longrightarrow V_j$ denoting that the values in attribute A_i originate from the view V_j. Notice that the values in an attribute can originate from more than one view at the same time, namely if two views have been joined on this attribute. We now give rules to compute the annotation of a node in the expression tree, if the annotations of its child nodes have been given. In this section we will restrict the detailed technical explanation to the constructions needed for the example query given in Fig. 1. For the complete explanation, we refer the reader to [4].

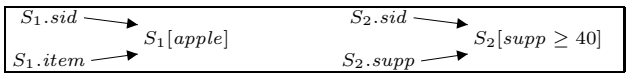
Leaf annotation. The annotations for the leaves *Sets* S_1 (node (a)) and *Supports* S_2 (node (b)) denote that there are two attributes connected to the views, named S_1 and S_2, respectively, both associated to an empty constraint ([ ]). For the leaf *Rules* R (node (c)), the annotation is a bit more complicated, as there are in fact four objects that can be constrained: the antecedent of the rule, the consequent, the union of the two, and the rule itself. Therefore, four variables are introduced that represent respectively these objects: S_a^R, S_c^R, S^R, and R. Thus, the annotations for the leaves (a), (b) and (c) are, respectively:



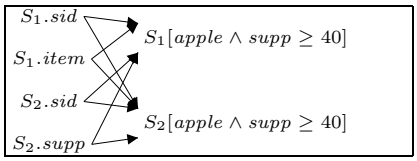
Annotation of internal nodes. In node (d), only those tuples coming from (a) that satisfy the condition ($item = apple$) are selected. From the annotation of node (a), we observe that the attribute $S_1.item$ originates from the view S_1 (*Sets*). Thus, we can actually associate the constraint $[apple]$ to S_1. By similar reasoning, we can associate the constraint $[supp \geq 40]$ to S_2 and the constraint $[conf \geq 80\%]$ to R. The annotation for nodes (d),(e) and (f) are, respectively:



In node (g), a cartesian product is made from the tuples coming from nodes (d) and (e). Therefore, all tuples from nodes (d) and from node (e) should be considered at this node. The annotation for node (g) is then the union of the annotations of its child nodes:

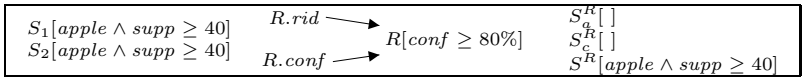


In node (h), only those tuples coming from node (g) that satisfy the condition ($S_1.sid = S_2.sid$) are selected. From the annotation of node (g), we can see that $S_1.sid$ is connected to the view S_1 with constraint $[apple]$ and $S_2.sid$ is connected to the view S_2 with constraint $[supp \geq 40]$. Then, according to the related condition, the tuples really needed to be considered in node (h) are those coming from the itemsets present in both of the views $S_1[apple]$ and $S_2[supp \geq 40]$, that is, the same collection of itemsets having the item *apple* and with support greater than or equal to 40. We can thus associate the constraint $[apple \wedge supp \geq 40]$ to both views. Furthermore, as the itemsets come from the same collection, the set of attributes of S_1 are now connected to the view S_2 and vice versa. The annotation for node (h) is as follows:



The annotation for nodes (i) and (j) are constructed in the same way as the annotations for nodes (g) and (h), respectively.

Finally, root node (l) projects the tuples coming from node (j) over the attributes $R.rid$ and $R.conf$. Its annotation is similar to that one of its child node (j), keeping, however, only the attributes $R.rid$ and $R.conf$ and its connections. The annotation for this node is then:



Observe that the annotation of the root node (l) has all the necessary information we need in order to know exactly which tuples should be materialized by the DBMS: According to the constraint $[apple \wedge supp \geq 40]$ associated to the views S_1 and S_2, we can identify that the views *Sets* and *Supports* should contain all tuples coming from itemsets having the item *apple* and with support of at least 40. According to the constraint associated to the view R, it is easy to deduce that the view *Rules* should contain the tuples coming from association rules with confidence of at least 80%. Moreover, as S^R is also associated to the constraint $[apple \wedge supp \geq 40]$, the association rules should be generated from the same collection of itemsets present in the views *Sets* and *Supports*. Note that the constraints associated to S_a^R and S_c^R are both empty, which means that there are no constraints considering the presence of items in the antecedent nor the consequent of the rules, respectively. With this information, the DBMS can trigger the necessary data mining algorithms with the identified constraints.

5 Conclusion

This paper proposes a different and new approach in which association rule mining results can be queried as if they were stored in traditional relational

tables. This approach is based on the existence of *virtual mining views* that represent mining results. Every time the user queries one of these views, data mining algorithms are triggered by the DBMS in order to materialize, according to the constraints within the given query, the patterns needed to answer it.

From the user's point of view, the virtual mining views will always contain all association rules and itemsets, but, according to our proposal, none of the patterns should be actually stored. In reality, the action of querying a virtual mining view triggers a data mining algorithm or a set of data mining algorithms transparently to the user, which means that the user does not need to know how to use data mining algorithms. Moreover, due to the fact that only the DBMS is extended, the user can query the data by using a standard relational query language. Extensions of query languages are not necessary in our approach.

Acknowledgement This work was partially funded by the EU project "IQ", and the FWO project "Foundations for Inductive Databases". The second author would also like to thank Jan Van den Bussche for many interesting discussions and suggestions resulting in several of the ideas presented in this paper.

References

1. S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases*, Addison-Wesley (1995).
2. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *20th VLDB Conference* (1994) 487–489.
3. T. Calders, L. V.S. Lakshmanan, R. T. Ng and J. Paredaens. Expressive Power of an Algebra for Data Mining. *Manuscript* (2006).
4. T. Calders, B. Goethals and A. Prado. Constraint Extraction from SQL-queries. *Manuscript* (2006).
5. B. Goethals, J. Van den Bussche and K. Vanhoof. Decision Support Queries for the Interpretation of Data Mining Results. *Manuscript* (1998).
6. B. Goethals and J. Van den Bussche On Supporting Interactive Association Rule Mining. In *2nd DaWaK* (2000) 307–316.
7. H. Garcia-Molina, J. D. Ullman and J. Widom. Database System Implementation. Prentice-Hall, Inc. (2000).
8. J. Han, Y. Fu, W. Wang, K. Koperski and O. Zaiane. DMQL: A Data Mining Query Language for Relational Databases. In *SIGMOD DMKD Workshop* (1996).
9. T. Imielinski and H. Mannila. A Database Perspective on Knowledge Discovery. *Communications of the ACM*, Vol. 39 (1996) 58–64.
10. T. Imielinski and A. Virmani. MSQL: A Query Language for Database Mining. *Data Mining and Knowledge Discovery*, Vol. 3 (1999) 373–408.
11. R. Meo, G. Psaila and S. Ceri. An Extension to SQL for Mining Association Rules. *Data Mining and Knowledge Discovery*, Vol. 2 (1998) 195-224.
12. T. Mitchell Machine Learning. McGraw Hill (1997).
13. H. Wang and C. Zaniolo. Nonmonotonic Reasoning in LDL++. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic Publishers (2000) 523–544.
14. H. Wang and C. Zaniolo. ATLaS: A Native Extension of SQL for Data Mining. In *3rd SIAM Conference* (2003) 130–144.