A Tightly-Coupled Architecture for Data Mining

Rosa Meo(1) Giuseppe Psaila(1) Stefano Ceri(2)

(1)Dip di Automatica e Informatica
Politecnico di Torino
C.so Duca degli Abruzzi 24
10129 Torino, Italy

(2)Dip. di Elettronica e Informazione
Politecnico di Milano
P.za Leonardo da VInci 32
20133 Milano, Italy

rosimeo@polito.it, psaila@elet.polimi.it, ceri@elet.polimi.it

Abstract

Current approaches to data mining are based on the use of a decoupled architecture, where data are first extracted from a database and then processed by a specialized data mining engine. This paper proposes instead a tightly coupled architecture, where data mining is integrated within a classical SQL server. The premise of this work is a SQL-like operator, called `MINE RULE`*, introduced in a previous paper. We show how the various syntactic features of the operator can be managed by either a SQL engine or a classical data mining engine; our main objective is to identify the border between typical relational processing, executed by the relational server, and data mining processing, executed by a specialized component. The resulting architecture exhibits portability at the SQL level and integration of inputs and outputs of the data mining operator with the database, and provides the guidelines for promoting the integration of other data mining techniques and systems with SQL servers.*

1 Introduction

Data mining is a recent technique for data analysis, aiming at the discovery of regularities among data which can reveal business-critical information. The extraction of *association rules* has emerged as the most important paradigm for data mining; thus, it has been deeply investigated by researchers and at the same time supported by many products. Several papers have considered this problem from an algorithmic point of view, showing the practicability of data mining on very large databases.

The approach to data mining followed by several products is based on a *decoupled architecture*; a data mining tool executes on data which is previously extracted from the database and transformed into a suitable format. We believe that the decoupled approach has several inconveniences:

- It requires the data analyst to perform a long preparation for extracting data (by means of queries) and preparing data (by means of explicit encoding), so as to achieve the format required by the tool.
- It offers a limited data mining paradigm, that sometimes matches well with specific requirements, but does not offer sufficient generality.
- Once extracted, rules are contained in the data mining tool, and it is quite hard to combine the information embedded into them with the data in the database.

Some systems [9] are able to retrieve data from SQL server, thus the data access problem is partially solved.

In this paper, we propose a *tightly-coupled* approach to data mining; we specify data mining operations by means of a SQL-like operator for data mining, called `MINE RULE` ([11]), which extends the semantics of association rules in order to cover a wide class of problems (e.g. traditional rules, sequential patterns-like rules, etc.). SQL is used in the extraction of the source data (by means of an unrestricted query on the database) and in the definition of generic association conditions (based on the *grouping* and further *clustering* of attributes) and mining conditions (selection predicates on rules). These SQL expressions, embedded in the `MINE RULE` operator, guarantee the specification of different semantics for association rules.

In this paper, we specify how the `MINE RULE` operator can be implemented on top of a SQL server; thus, this paper concentrates on the development of an integrated, tightly-coupled architecture.

Previous literature on the mining of association rules has established that rule extraction is efficiently performed by an iterative process [1], with ad hoc memory-resident data structures. In particular, references [3, 12, 13] have shown how the algorithm's efficiency can be improved by means of variants in the data structures, and that the number of input-output operations, when data does not fit into main memory, can be reduced to a minimum (more than one but less than two) by means of suitable sampling [7]. The outcome of this work gives evidence that the implementation of `MINE RULE` cannot be fully relational (e.g., by means of SQL query expressions) even in the context of a tightly-coupled architecture; the core part of data mining is the responsibility of a non-SQL procedure, that we call *core operator*, which receives data retrieved by the SQL sever, extracts rules, and returns them in the format of database relations.

Nevertheless, many features of the `MINE RULE` operator (like sub-query evaluation, grouping, etc.) are better managed in SQL, outside of the core operator. Thus, one of the main problems in designing such an architecture (not only from a research viewpoint, but also from and industrial viewpoint), is to identify the border between typical *relational processing*, to be executed by the SQL server, and *mining processing*, to be executed by the core operator, such that each component executes the tasks for which it is best suited. This work os influenced by our wishes of:

• Ensuring portability, in order that the implementation does not depend on a particular DBMS.
• Exploiting the use of SQL queries in order to obtain a reasonable efficiency.

In the rest of the paper, we discuss the data mining architecture in detail: we show its components, the process flow between SQL server and core operator, and the relevant aspects of a translation required in order to generate the proper SQL queries from the data mining operator. The core operator is described in functional terms (i.e., in terms of its inputs and outputs); its innovative features have required several adaptations of the well known algorithms proposed in literature, briefly described in Section 3.

Observe that similar work has been done in [2, 6, 8]. However, we need to follow a different approach, due to the semantic richness of our operator.

The paper is organized as follows: Section 2 summarizes the most important features of the `MINE RULE` operator and gives its syntax; Section 3 presents the global tighty-coupled architecture and summarizes the features of the core operator. Section 4 describes the architectural components in detail, focusing on the interfaces among them. Section 5 draws the conclusions.

2 The Mine Rule Operator

We summarize the main features of the `MINE RULE` operator by showing it "at work" on the most classical data mining problem, consisting of mining association rules relative to regularities in purchases in store.

The database contains data about customers, products, and purchases. Each customer buys a set of products (also called *items*); the whole purchase is referred to as a *transaction* having a unique identifier, a date and a customer code. Each transaction contains the set of bought items with the purchased quantity and the price. These data are stored in the (nonnormalized) table `Purchase`, depicted in Figure 1.

Suppose that we are interested in finding out when 1995 purchases of items by a given customer, with

tr.	*cust.*	*item*	*date*	*price*	*q.ty*
1	$cust_1$	ski_pants	12/17/95	140	1
1	$cust_1$	hiking_boots	12/17/95	180	1
2	$cust_2$	col_shirts	12/18/95	25	2
2	$cust_2$	brown_boots	12/18/95	150	1
2	$cust_2$	jackets	12/18/95	300	1
3	$cust_1$	jackets	12/18/95	300	1
4	$cust_2$	col_shirts	12/19/95	25	3
4	$cust_2$	jackets	12/19/95	300	2

Figure 1: The `Purchase` table for a big-store.

price greater than or equal to \$100 (the "premise" of rule), are followed by purchases of items having price less than \$100 by the same customer (the "consequence"), such that the second purchases occur in the same date, successive to the date of the premise. In order not to extract too many rules, we further impose that extracted correlation rule be present in 20% of the customers, and that it occurs at least 30% of the times when the premise of the rule occurs.

The above problem specifies a rather complex association rule, which is characterized by the pair of items representing the premises and consequences of the rule. The premise is called the *body* of the rule, while the consequence is called the *head* of the rule.

This problem is specified as follows.

```
MINE RULE FilteredOrderedSets AS
SELECT DISTINCT 1..n item AS BODY,
   1..n item AS HEAD, SUPPORT, CONFIDENCE
   WHERE BODY.price>=100 AND HEAD.price<100
FROM Purchase
   WHERE date BETWEEN 1/1/95 AND 12/31/95
GROUP BY customer
CLUSTER BY date HAVING BODY.date<HEAD.date
EXTRACTING RULES WITH SUPPORT: 0.2,
   CONFIDENCE: 0.3
```

Next, we sketch the "operational semantics" of the operator; such semantics is essential for understanding the meaning of each clause in the operator. A complete characterization of the `MINE RULE` operator, together with the specification of the semantics in a suitable extended relational algebra, is discussed in [11]. The operator performs the following steps.

1. `FROM .. WHERE` clause. Extracts the relevant tuples of the `Purchase` table (those relative to 1995 purchases); the result of this clause is the actual input table of the data mining process.
2. `GROUP BY` clause. Logically divides the `Purchase` table into disjoint groups based on the attribute `customer`; rules are extracted from within groups, and express regularities among them.
3. `CLUSTER BY` clause. Partitions groups into subgroups called *clusters* on the attribute `date` (Figure 2.a). Each cluster can be mapped to both the body and the head of a given rule; we indicate the former case as *body cluster* and the latter as *head cluster*. If the cluster clause is not present, all tuples within a group are considered as both the body group and the head group.
4. `HAVING` clause. All body and head clusters are paired, and only pairs such that the body cluster has a date less than the head cluster are considered for extracting items. If the `HAVING` clause is not present, all pairs survive.
5. `SELECT ... WHERE` clause, `EXTRACTING` clause. For each pair of body and head clusters surviving the `HAVING` clause, items are extracted to form a rule, so that they satisfy the mining condition `WHERE BODY.price >= 100 AND HEAD.price < 100`; rules must have sufficient support and confidence. All cluster pairs contribute to the evaluation of support (the number of groups containing the rule on the total number of groups), while all body clusters are

used for computing confidence (the number of groups containing the rule on the number of groups containing the body). In this case, the minimum support is `0.2` and the minimum confidence is `0.3`.

6. Finally, the resulting rules are stored into a SQL3 table called `FilteredOrderedSets`, shown in Figure 2.b; observe the representation for association rules is, e.g., $\{$*brown_boots,jackets*$\}\Rightarrow\{$*col_shirts*$\}$.

cust	*date*	*item*	*tr.*	*price*	*q.ty*
cust_1	12/17/95	ski_pants	1	140	1
		hiking_boots	1	180	1
	12/18/95	jackets	3	300	1
cust_2	12/18/95	col_shirts	2	25	2
		brown_boots	2	150	1
		jackets	2	300	1
	12/19/95	col_shirts	4	25	3
		jackets	4	300	2

a)

BODY	*HEAD*	*S.*	*C.*
{brown_boots}	{col_shirts}	0.5	1
{jackets}	{col_shirts}	0.5	0.5
{brown_boots,jackets}	{col_shirts}	0.5	1

b)

Figure 2: a) The `Purchase` table grouped by `customer` and clustered by `date`. b) The output table `FilteredOrderedSets`.

3 Architecture

In this section, we illustrate the architecture of a *data mining system*, i.e. a system whose fundamental task is to support `MINE RULE` requests on top of a relational database server. The design of this architecture aims at the following objectives:

1. *Relational Interoperability.* The architecture includes a relational server supporting SQL92.
2. *Algorithm Interoperability.* For mining both conventional and general association rules, the architecture is independent of the particular mining algorithm.
3. *Ease of use.* Unexperienced users which are not familiar with the SQL-like style of the operator are supported by a user-friendly interface.
4. *Ease of view.* The user support provides facilities for viewing rules generated by the analysis process.

Mining of simple association rules can be done with an arbitrary mining algorithm based on an arbitrary approach, e.g. partitioning-based approaches (e.g. [13]) or sampling-based approaches ([7, 13]); typically each of them has better performance under specific assumptions about data and rule distribution. Mining of complex rules supported by the `MINE RULE` operator (with advanced semantic features such as clusters, mining condition, etc.) requires an extension of those algorithms, because these semantic features are not considered by known algorithms [1, 3, 7, 12, 13].

In order to guarantee algorithm interoperability, the real source data are hidden to the algorithm and made fully uniform by means of their preliminary encoding. This encoding requires several relational transformations which are effectively and efficiently evaluated by the SQL server itself, rather than being implemented ad-hoc. Given that such transformations refer to the actual schema of the database, the corresponding SQL clauses are evaluated before encoding and standardizing the source data. Consequently, the system includes a significant preprocessing of input data into a suitable input format for the algorithms, in a way which is totally transparent to the user.

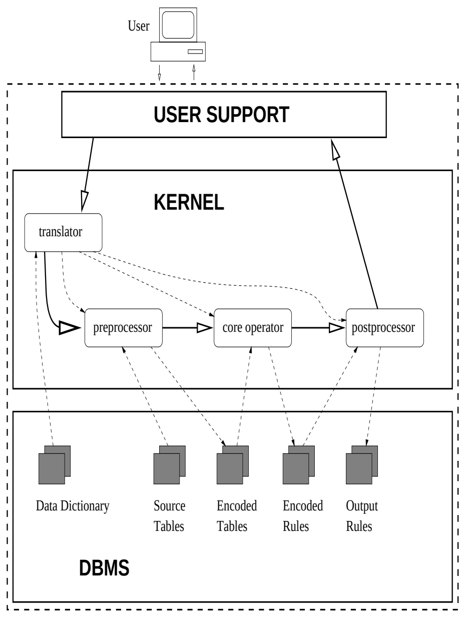
The previous considerations led us to define the architecture shown in Figure 3; in the figure, thick lines denote process flow, dashed lines denote data/information flow. The system is divided in three fundamental modules:

- **User Support**, which has the responsibility of interacting with the user;
- **Kernel**, which interprets the mining statement and perform rule extraction;
- **DBMS**, which stores source data, output rules and intermediate results.

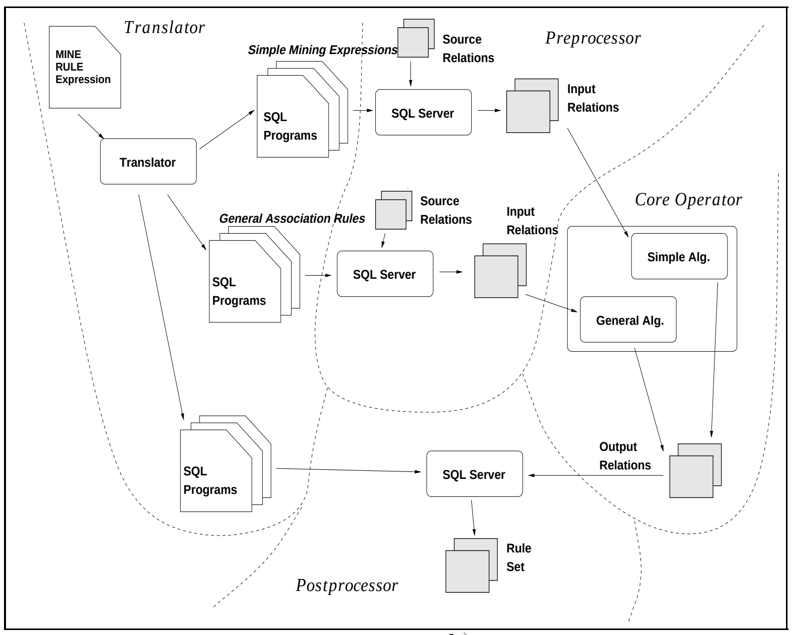
In this paper, we focus on the kernel of the mining system; user support is described in a related paper [4]. The main kernel components are in Figure 3.a):

1. *Translator.* The translator interpretes a `MINE RULE` statement, checks the correctness of the statement by accessing the DBMS *Data Dictionary*, and produces translation programs used by the preprocessor and postprocessor. It is invoked when the user submits a mining statement.
2. *Preprocessor.* The preprocessor retrieves source data, evaluates the mining, grouping and cluster conditions of the mining statement, and encodes data that will appear in rules; it produces a set of *Encoded Tables*, stored again into the DBMS. Note that the same preprocessing could be in common to the execution of several data mining queries, thus saving its cost.
3. *Core Operator.* The core operator performs the actual generation of association rules. It works on the *Encoded Tables*, prepared by the preprocessor, and produces the set of *Encoded Rules*, i.e. association rules whose elements are still encoded. This component is invoked after the preprocessor, and uses directives from the translator to decide the mining technique to apply: in fact, depending on the particular mining statement, the mining process changes to capture the semantics of the statement. As a result, the core operator can be constituted of a pool of mining algorithms.
4. *Postprocessor.* At the end of the chain, the postprocessor decodes the rules, producing the relations containing the desired rule set in a normalized, user-readable form; results take a tabular form (*Output Rules*) and are stored into the database. It is invoked when the core operator terminates, receiving directives from the translator about the final format of rules. It ends the mining process, and the control passes again to the user support.

This architecture guarantees algorithm interoperability because the core operator is decoupled from



a)



b)

Figure 3: a) Architecture of the Data Mining System. b) Translations required by the mining system.

the other components of the kernel, and the only interaction is through the encoded source data and the encoded rules, stored in the DBMS, and the directives received by the translator. Consequently, algorithms are completely hidden to the rest of the system.

The preprocessor is implemented by means of a pool of SQL programs generating the input relations for the core operator; these programs are in turn generated by the translator; depending on the class of mining statements, different sets of programs are necessary (see Section 4.2 for a detailed discussion).

Figure 3.b) outlines as well that the envisioned architecture has two versions of the core process, called *simple*, and *general*; each of them corresponds to a precise class of `MINE RULE` expressions.

1. The first one includes the *simple association rules*, in which the head and body of the rule concern the same attribute(s) , and neither the `CLUSTER BY` clause nor the mining condition are specified;
2. The second one includes the *general association rules*, i.e. statements for which the above restrictions do not hold; thus, statements extracting general association rules may contain the `CLUSTER BY` clause or the mining condition, and the body and head of rules may be defined on two different sets of attributes.

For each one of the two classes, the core operator applies a variant of the algorithm, designed to take advantage by the peculiar features of the class.

In the rest of this paper, we concentrate on the components of the kernel, which are specific of our architecture; these are the translator, preprocessor, and postprocessor of Figure 3.a).

4 The Architectural Components

This section presents each architectural component.

4.1 The Translator

The translator checks the correctness of the mining statement, and classifies it so that the subsequent modules activate only those computations which are required for its processing. The classification is made through a syntactic analysis of `MINE RULE` statements.

The `MINE RULE` operator uses the following syntax, where symbols enclosed by < > denote nonterminals and square brackets denote optionality. Symbols like `<group attr list>`, `<body schema>` and `<head schema` are list of attributes, while simbols like `<group cond>` are search conditions.

```
<mine rule op> ::=
 MINE RULE <output table name> AS
 SELECT DISTINCT <body descr>,<head descr>
 [,SUPPORT] [,CONFIDENCE]
 [ WHERE [<mining cond>]
 FROM <from list> [WHERE <source cond>]
 GROUP BY <group attr list> [HAVING <group cond>]
 [ CLUSTER BY <cluster attr list>
  [HAVING <cluster cond>] ]
 EXTRACTING RULES WITH SUPPORT:<uns number>,
                   CONFIDENCE:<uns number>
<output table name>::= <identifier>
<body descr>::= [<cardspec>] <body schema>
  AS BODY /* default card: 1..n */
<head descr>::= [<cardspec>] <head schema>
  AS HEAD /* default card: 1..1 */
<card spec>::= <uns int> .. (<uns int> | n
<from list>::= <table ref> {, <table ref>}
```

In the rest of the paper, we will use symbols of the grammar to refer to the actual substrings in `MINE RULE` specifications. For instance, considering the example of Section 2, `<mining cond>` refers to the string `BODY.price >= 100 AND HEAD.price < 100`.

The translator performs the following semantic checking, in order to ensure correctness of expressions:

1. All attribute lists must be defined on the schema of source tables.
2. Grouping attributes (`<group attr list>`) and clustering attributes (`<cluster attr list>`) must be disjoint sets of attributes, and attribute sets constituting the body schema (`<body schema>`) and the head schema (`<head schema>`) must be disjoint from grouping and clustering attributes.
3. The `HAVING` clause for grouping (clustering) can refer only to grouping (clustering) attributes.
4. The mining condition (`<mining cond>`) can refer to every attribute but the grouping and clustering ones.

We classify the `MINE RULE` expressions on the basis of their syntactic clauses by defining several boolean variables; these are used as directives for the three components performing the mining process (i.e. preprocessor, core operator and postprocessor).

The following boolean variables are orthogonal, i.e. each value does not depend on the other ones.

• H: true when the body and head are relative to different attributes.
• W: true in presence of the `<source cond>` (and of the corresponding tables in the `<from list>`).
• M: true in presence of the `<mining cond>`.
• G: true in presence of the `<group cond>`.
• C: true in presence of the `CLUSTER BY` clause.

Instead, the following boolean variables depend on the above ones.

• K: true in presence of the `<cluster cond>`. Observe that the `<cluster cond>` needs the presence of the `CLUSTER BY` clause; thus $K \Rightarrow C$.
• F: true in presence of aggregate functions in the `<cluster cond>`; thus $F \Rightarrow K$.
• R: true in presence of aggregate functions in the `<group cond>`; thus $R \Rightarrow G$.

4.2 The Preprocessor

The preprocessing phase prepares the input data for the core operator, which are different depending on the class of mining statements. In particular, in the case of simple association rules, only the encoding of items and groups is necessary; in contrast, in the case of general association rules, additional information is required, concerning clusters and mining conditions.

4.2.1 Preprocessing for Simple Rules.

In Figure 4.a), we illustrate the transformations required for encoding of the source data when we aim at producing simple association rules. The encoding produces tables containing all the large items (i.e. items with sufficient support); this is the input for all algorithms known in the literature.

• At first, it is necessary to retrieve the source data by computing the query in the `FROM ... WHERE` clause; this is done by query Q_0, which materializes a view `Source`, containing the actual data to analize. Observe that this query is skipped when W is false, i.e. only one table is present in the clause, and no condition is specified.
• Next, it is necessary to count the total number of groups inside the source data; this counter is necessary to evaluate support of items and rules, and is performed by query Q_1.
• In presence of the `HAVING` predicate in the `GROUP BY` clause (when G is true), a selection of groups must be done; this is done by query Q_2, which also encodes groups, substituting the set of attributes which identify groups with a unique numerical identifier.
• Next, each item candidate to appear in rules is encoded: this encoding step is required to make the core operator unaware of the real item structure. As an effect, query Q_3 generates table `Bset`, in which each item (identified by attributes in `Battlist`), is associated a unique identifier `Bid`.
• Finally, the encoded version of table `Source` is generated, in such a way that each tuple in the source data is described by the identifier of the group to which the tuple belongs (`Gid`), and the identifier of the item that can be extracted from the tuple (`Bid`). This work is done by query Q_4, that generates table `CodedSource` in such a way only tuples containing large items are actually encoded, since tuples with no large items cannot participate to the rule generation process.

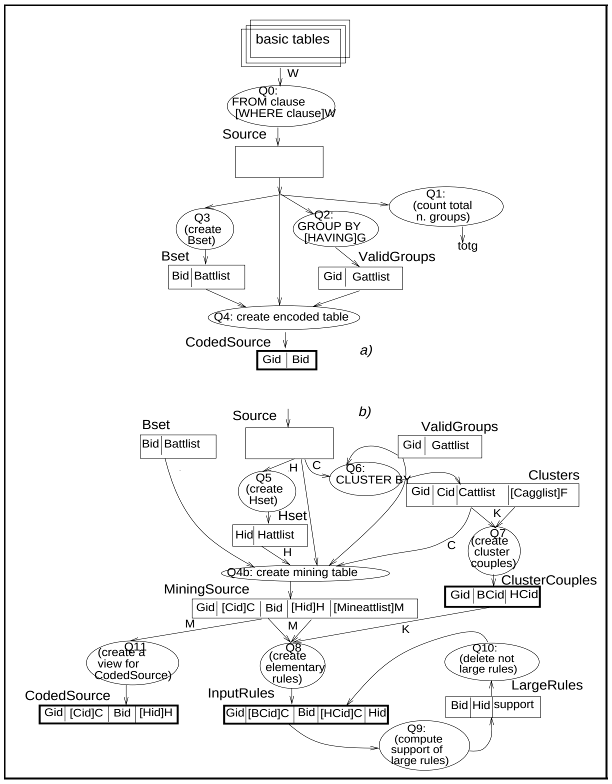


Figure 4: Structure of the preprocessor.

Appendix A describes the SQL queries which are needed by this transformation; they are the most significant for supporting data mining on a commercial relational server, and at the same time formally specify the preprocessor task (more details are in [5]).

4.2.2 Preprocessing for General Rules.

Figure 4.b) shows the additional encoding required by general association rules, i.e., rules with either clusters, or mining conditions, or body and head having different schema. The challenge of this translation consists in determining a suitable borderline between the preprocessor and the core operator, in order to obtain algorithm interoperability in this novel context.

Part of the preprocessing for simple association rules is repeated in the general case (queries Q_0, Q_1, Q_2 and Q_3). However, if the body and head schema are different (when H is true), two distinct encodings for items for body and head must be performed. Assuming that items for body are encoded into table `Bset`, query Q_5 encodes items for head, generating table `Hset`.

In presence of the `CLUSTER BY` clause (when the boolean veriable C is true) it is necessary to encode attributes identifying clusters as well. This is performed by query Q_6, which produces the cluster encoding into the temporary table `Clusters`. In this table, group identifier (`Gid`) and cluster identifier (`Cid`) uniquely identify a cluster.

Finally, a different version of the `CodedSource` table is generated by queries Q_4b and Q_{11}. We have designed the encoding of the source table as a natural extension of the encoding for simple association rules; each input tuple in the source data is described in terms of group identifier `Gid`, cluster identifier `Cid` (if C is true), body item identifier `Bid`, head item identifier `Hid` (if H is true). Notice that the generation of the `CodedSource` table proceeds through the generation of the intermediate `MiningSource` table by query Q_4b: this table associates to each encoded tuple the set `Mineattlist` of attributes appearing in the mining condition (if M is true), that will be used in a successive query; then, query Q_{11} defines `CodedSource` as a not materialized view of `MiningSource` (thus there is no computation). Table `CodedSource` is the actual input to the core operator; `MiningSource` is hidden to the core operator. Observe that the schema of tables `CodedSource` and `MiningSource` is not fixed, but changes depending on which of C, H and M is set to true; if a variable is not true, the corresponding portion of the schema is not present.

Let us now discuss the remaining tasks of the preprocessing: these concern selection of cluster pairs and evaluation of the mining condition.

Recall from Section 2 that in presence of the `HAVING` predicate in the `CLUSTER BY` clause (K is true), not all pairs of clusters inside a group are valid for rule extraction. In order to make the core operator unaware of attributes and predicates in this clause, the selection of valid cluster pairs is performed by query Q_7, which may use the results (associated to each cluster in table `Clusters`) of the evaluation of aggregate functions possibly present in the cluster condition, performed by query Q_6; valid cluster pairs are stored into the input table `ClusterCouples`.

When specified, the mining condition must be evaluated on the mining attributes in order to form elementary rules with one item in the body and one item in the head; in such a way, the predicate is evaluated by the SQL, and no information concerning the attributes involved in the predicate arrives to the core operator. An elementary rule is described in terms of group identifier, cluster identifiers for the body and the head, identifiers of the body and head items. Elementary rules are generated by query Q_8, which uses table `MiningSource` and possibly table `ClusterCouples`, to generate the first instance of table `InputRules`. Elementary rules with not sufficient support are discarded: query Q_9 computes support of each rule, generating the temporary table `LargeRules`, then query Q_{10} discards rules without sufficient support, and generates the final instance of table `InputRules`. Notice that the confidence of rules is not considered here, because it involves the evaluation of support for bodies, built by the core operator.

Depending on the class of mining statement, not all the input tables are generated; in particular, if the mining condition is not specified, table `InputRules` is not computed; analogously, if the cluster condition is not specified, table `ClusterCouples` is not generated. In contrast, the input table `CodedSource` is always produced. In Figure 4, input tables for core processing have a thick border. Queries Q_0 to Q_{11} are reported in [5].

4.3 The Core Operator

The core operator performs the actual discovery of the association rules that satisfy the mining request; it incorporates all those computations which cannot efficiently be programmed as SQL queries.

4.3.1 Core Processing of Simple Rules

The simple core processing algorithm is one of the traditional data mining algorithm, described in [1, 3, 12], whose general approach is summarized below. The algorithm incrementally builds the so-called *large itemsets*, i.e. sets of items (elements that will appear in final rules) with sufficient support, moving up from singleton itemsets to itemsets of generic cardinality by adding one new item to already computed large itemsets. The fixpoint of the process is reached when no new large itemsets are generated. Support of an itemset is evaluated by counting elements in an associated list that contains identifiers of groups in which the itemset is present; the list is computed when the new itemset is generated. Then, rules are built from large itemsets by extracting subsets of items: indicating with L a large itemset and with $H \subset L$ a subset, we form the rule $(L - H) \Rightarrow H$ when it has suitable confidence.

4.3.2 Core Processing of General Rules

With general association rules, the core operator starts from the initial set of large elementary rules, and proceeds discovering rules with bodies and heads of arbitrary cardinality from the initial set of rules.

The discovery of rules of arbitrary cardinality proceeds as follows: given the set of rules $m \times n$, where m is the cardinality of the body and n is the cardinality of the head, the algorithm computes the set of rules $(m+1) \times n$ and the set of rules $m \times (n+1)$, from which rules with insufficient support are pruned. This process determines a lattice: at the top, we find the set of elementary rules; given a rule set $m \times n$, the left child contains rules $(m+1) \times n$, while the right child contains rules $m \times (n+1)$.

The process of rule discovery starts by considering all the rules in the set at the top of the lattice; then, it generates rules for the rule sets in the subsequent layer, and so on, until a layer with empty rule sets is produced. Observe that rule sets $m \times n$, with m and n both greater than 1, can be obtained in two alternative ways: both from the rule set $(m-1) \times n$, and from the rule set $m \times (n-1)$. The efficiency of the algorithm is maximized if, at each step, we start from the set with lower cardinality. A detailed description of the algorithm is outside the scope of this paper and can be found in [5, 10].

Observe that the generation of elementary rules is performed by the preprocessor (in SQL) in presence of mining condition, because it involves the evaluation of a SQL predicate. In this case, elementary rules are retrieved by the core operator from table `InputRules`. In the other cases, the core operator itself performs the precomputation of elementary rules, which conceptually requires the building of the cartesian product of the source tuples belonging to the same group. The resulting elementary rules are associated with the identifier of the groups and possibly the clusters from which they are extracted; the cluster condition selects over this cartesian product only certain cluster pairs, reported in table `ClusterCouples`. The cartesian product is not materialized: it is described by pointers between source tuples. For this reason, this computation is part of the core operator instead of being done in SQL.

4.4 The Postprocessor

The last component of the architecture produces the output rules. Conceptually, the core operator produces rules as associations between two itemsets, the first one for the body, the second one for the head, where each itemset is a set of item identifiers. Thus, the output of the `MINE RULE` operator should include attributes built by the set type constructor; however, this feature is not standardized and not yet available on most relational systems.

In order to avoid any portability problem (and to achieve algorithm independence also from the viewpoint of the produced outputs), the core operator generates a normalized form based on the use of three tables. In the first one, each tuple corresponds to a rule, and has schema `(BodyId, HeadId, SUPPORT, CONFIDENCE)`; attributes `BodyId` and `HeadId` are identifiers of encoded bodies and heads, returned by the core operator in two auxiliary tables `OutputBodies` (with schema `(BodyId, Bid)`) and `OutputHeads` (with schema `(HeadId, Hid)`), respectively.

Each item in these tables is referred to by means of its identifier. Thus, while table `<output table name>` is already in the user format, tables `OutputBodies` and `OutputHeads` must be translated by the postprocessor into the corresponding, user readable, `<output table name>_Bodies` and `<output table name>_Heads`, by means of the temporary tables `Bset` (and `Hset` too when H is true).

5 Conclusions

The integration of mining of association rules with relational processing is a step towards the development of tightly integrated environments for on-line analytical processing. We believe that the current diversity of data mining products represents just the initial step in the natural evolution of this market sector, and that sooner or later standardization will become a must also for data mining products. Then, integration with the SQL language and processing will be two key success factors.

The main contribution of this paper is to identify an architecture for the seamless integration of the data mining process with a SQL server. We show that the integration is feasible, provided that the "core processing" of data mining is performed efficiently by specialized algorithms. We propose a "borderline" between relational and data mining processing, and we demonstrate that with this choice of borderline the translation process for an arbitrary data mining operation can be done in a rather easy and efficient way.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc.ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993. British Columbia.

[2] R. Agrawal and K. Shim. Developing tightly-coupled data mining applications on a relational database system. *KDD-96*, pages 287–290, 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, September 1994.

[4] E. Baralis, S. Ceri, R. Meo, G. Psaila, M. Richeldi, and P. Risimini. Amore: An integrated environment for database mining. In *Proceedings of SEBD 97 Sistemi Evoluti per Basi di Dati*, Verona, Italy, June 1997.

[5] E. Baralis, R. Meo, G. Psaila, and M. Richeldi. Research report number 5. *Project 101196 CSELT - Politecnico di Torino*, June 1997.

[6] J. Han et al. Dbminer: A system for mining knowledge in large relational databases. *KDD-96*, pages 250–255, 1996.

[7] H.Toivonen. Sampling large databases for association rules. In *Proceedings of the 22nd VLDB Conference*, Bombay (Mumbai), India, 1996.

[8] T. Imielinski, A. Virmani, and A. Abdoulghani. Datamine: Application programming interface and query language for database mining. *KDD-96*, pages 256–260, 1996.

[9] Silicon Graphics Incorporated. Mineset. *http:www.sgi.com/Products/software/MineSet.*

[10] R. Meo. *Un Linguaggio e Nuove Soluzioni per il Data Mining.* Ph.D. Thesis, in Italian, 1997.

[11] R. Meo, G. Psaila, and S. Ceri. A new sql-like operator for mining association rules. In *Proceedings of the 22st VLDB Conference*, Bombay, India, September 1996.

[12] J. S. Park, M. Shen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, San Jose, California, May 1995.

[13] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st VLDB Conference*, Zurich, Swizerland, 1995.

Appendix A: SQL Queries

As introduced in Section 4.1, we report the SQL programs performing the preprocessing for the case of simple association rules. We refer to Figure 4. In the following, given a boolean condition C, the notation $[expression]C$ indicates an expression that is actually present in the final query if condition C is true; we use as conditions the same boolean variables which describe the presence of seven syntactic components in a `MINE RULE` operation, as defined in Section 4.1. AR stands for Association Rules.

Query Q_0: Basic tables listed in the `<from list>` are joint, possibly using the `<source cond>`, if specified (denoted by the boolean condition W; it produces table `Source`, i.e. the table on which AR extraction is performed. `<needed attr list>` indicates the set of attributes that are actually useful in AR extraction obtained as the union of the attribute lists `<body schema>`, `<head schema>`, `<group attr list>`, `<cluster attr list>` and `<mine attr list>` (attributes refered to by the `<mining cond>`) plus the attributes possibly referred by aggregate functions in `<group cond>` and `<cluster cond>`.

```
INSERT INTO Source (SELECT <needed attr list>
 FROM <from list>[WHERE <source cond>]W
```

Query Q_1: It calculates the number of groups in the `Source`. The result is stored into variable `:totg`.

```
SELECT COUNT(*  INTO :totg FROM
(SELECT DISTINCT <group attr list> FROM Source
```

Query Q_2: First, The `Source` is partitioned into groups on attributes in `<group attr list>`, applying the `<group cond>` if present (boolean condition G); the view `ValidGroupsView` is produced.

```
CREATE VIEW ValidGroupsView AS
(SELECT <group attr list> FROM Source
GROUP BY <group attr list> [HAVING <group cond>]G
```

Second, the table `ValidGroups` is generated. An identifier, called `Gid`, is associated for each valid group, and is the numerical identifier for groups.

In Oracle numerical identifiers are obtained by sequences that are defined by the following instruction:

```
        CREATE SEQUENCE <sequence-name>
```

A value of the sequence is generated by a call to the function: `<sequence-name>.NEXTVAL`. The identifier `Gid` will be generated in the following instruction by means of a sequence called `Gidsequence`. Each valid group taken from the view `ValidGroupsView` is inserted with a unique value of the sequence into table `ValidGroups`.

```
INSERT INTO ValidGroups
(SELECT Gidsequence.NEXTVAL
 AS Gid, V.* FROM ValidGroupsView AS V
```

This kind of generation of identifiers applies also to queries Q_3, Q_4 and Q_5. For brevity we simply indicate this operation with `<sequence-name>.NEXTVAL`, omitting the rest of details.

Query Q_3: The encoding of items is performed by query Q_3. At first, the temporary table `DistinctGroupsInBody` is generated by the following query. This table represents for each body element the distinct valid groups containing it.

```
INSERT INTO DistinctGroupsInBody
(SELECT DISTINCT <body schema>,<group attr list>
 FROM Source
```

Then, the following query generates the `Bset` table containing for each element an attribute for the identifier (`Bid`), the attributes in the `<body schema>` and the number of valid groups that contain the element (this number is later used to compute confidence of elementary rules if the general core processing algorithm is adopted); in particular, only elements which appear in a sufficient number of groups are selected by the `HAVING` clause.

```
INSERT INTO Bset
(SELECT Bidsequence.NEXTVAL AS Bid,<body schema>
 FROM  DistinctGroupsInBody GROUP BY <body schema>
 HAVING COUNT(*  > :mingroups
```

Query Q_4: Finally, the `CodedSource` table is produced, by means of a join between tables `Source`, `ValidGroups` and the `Bset`.

```
INSERT INTO CodedSource
(SELECT DISTINCT V.Gid, B.Bid
 FROM Source S, ValidGroups AS V, Bset B
 WHERE S.<group attr list> = V.<group attr list>
   AND S.<body schema> = B.<body schema>
```

To conclude, the postprocessing query for bodies are reported (for simple association rules).

```
INSERT INTO <output table name>_Bodies
(SELECT BodyId, <body schema>
 FROM OutputBodies, Bset
 WHERE OutputBodies.Bid = Bset.Bid
```