# Towards a Language for Pattern Manipulation and Querying*

Elisa Bertino[1], Barbara Catania[2], and Anna Maddalena[2]

[1] Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano (Italy)
[2] Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova (Italy)

**Abstract.** Patterns are concise, but rich in semantic, representation of data. The approaches proposed in the literature to cope with pattern management problems usually deal with a single type of knowledge artifact and mainly concern pattern extraction issues. Little emphasis has been posed in defining an overall environment to represent and efficiently manage different types of patterns. The first general approach to deal with patterns has been proposed in the context of the PANDA project [1]. In this paper, we discuss some basic requirements for pattern manipulation and retrieval, represented according to the PANDA model. The proposed languages extend previous proposals and represent the basis for the development of an efficient pattern query processor.

## 1 Introduction

Patterns are concise, but rich in semantic, representation of data. There are many different applicational contexts from which different types of patterns can be generated (e.g. market-basket analysis/association rules, click-stream analysis/click sequences, image recognition/image features, etc.). Moreover, due to the diffusion of the (Semantic) Web, the ability to manipulate different types of patterns is becoming a fundamental issue for any "intelligent" data-intensive and distributed application, where systems must be able to handle and analyze multisite and multiowner data repositories.

Even if several applicational contexts call for a systematic approach to pattern management, the problem of directly storing and querying pattern-bases has received very limited attention so far in the commercial world and the database community. Several approaches have been proposed in the literature to cope with knowledge extraction and management problems. However, they usually deal with a single type of knowledge artifact. As far as we know, the first attempt to introduce a general approach for patterns handling has been proposed in the context of the PANDA project [9, 1]. In the PANDA approach, in order

* This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-33058 PANDA project.

to ensure an efficient handling of both raw data and patterns, two dedicated systems are considered: a traditional DataBase Management System (DBMS) is used for raw data while patterns are managed by a specific Pattern Based Management System (PBMS). The proposed model provides the representation of patterns in terms of their structure, data source, some measures qualifying the quality of the achieved representation, and an expression which relates patterns with the subset of the data source from which they have been generated.

The difference in semantics between patterns and raw data discourages from adopting the same query language for both. Since raw data are managed by using traditional DBMSs, they can be queried by using traditional query languages. On the other side, to retrieve patterns from the PBMS a specific *pattern query language* (PQL) has to be defined, capable of capitalizing on the peculiar semantics of patterns as defined in the model. In particular, besides basic operators to retrieve patterns, the pattern query language has to support operations binding patterns with raw data (*cross-over queries*). Moreover, there is also the need for a *pattern manipulation language* (PML), by which patterns can be generated from raw data, inserted from scratch, deleted, and updated.

The aim of this paper is to present a manipulation language and a query language for patterns represented according to the PANDA model [1]. We first identify basic pattern manipulation operators and provide examples of their usage. To this purpose, traditional manipulation operations, such as insertion, deletion and update, are re-interpreted in the context of the PBMS, leading to the definition of specific operations, such as extraction, recomputation and synchronization. Then, we present an algebraic PQL. The proposed operators extend all previous proposals. In particular, they extend traditional relational operators to cope with patterns of arbitrary type and support cross-over queries. Due to their formal semantics and generality, they can be used as the basis for the development of optimization techniques for pattern query processing.

This paper is organized as follows. In Section 2, the architecture and pattern model defined in the context of PANDA are briefly introduced. Some preliminary notions, useful in defining the pattern manipulation and query language, are then provided in Section 3. Sections 4 and 5 introduce the PML and the PQL, respectively. In Section 6 related work are discussed and, finally, in Section 7 we outline some conclusions and discuss some future work directions.

## 2 Pattern Management

A *Pattern-Base Management System* (PBMS) [9] is a system for handling (storing/processing/retrieving) patterns defined over raw data. Due to their different characteristics, raw data and patterns cannot be managed by using a single management system. According to the PANDA proposal [9], we assume that raw data reside in the raw data layer and are managed by their native systems (for example, a DBMS or just files); on the other side, patterns generated from raw data or inserted from scratch are stored and managed within the PBMS. Within the PBMS, it is worth to distinguish three different layers. The *pattern*

*layer* is populated with patterns. The *type layer* holds built-in and user-defined pattern types. The *class layer* holds definitions of pattern classes, i.e., collections of semantically related patterns.

The basic concepts of the pattern model are therefore: pattern types, patterns, and classes [1, 9]. A *pattern type* represents the intensional form of patterns, giving a formal description of their structure and relationship with source data. It is defined as a record with five elements: (i) the *structure schema* $ss$, which defines the pattern space by describing the structure of the instances of the pattern type; (ii) the *source schema* $ds$, which defines the related source space by describing the dataset from which patterns, instances of the pattern type being defined, are constructed (when no otherwise stated, we assume that data sources are intensionally described); (iii) the *measure schema* $ms$, which describes the measures which quantify the quality of the source data representation achieved by the pattern; (iv) the *formula* $f$, which describes the relationship between the source space and the pattern space, thus carrying the semantics of the pattern. Inside $f$, attributes are interpreted as free variables ranging over the components of either the source or the pattern space. Note that, though in some particular domains $f$ may exactly express the inter-space relationship, in most cases it will describe it only approximatively. The model is parametric with respect to the language adopted for formulas.

*Patterns* are instances of a specific pattern type. Thus, they are record values with identifiers. Each record contains the following elements: (i) a structure that positions the pattern within the pattern space defined by its pattern type; (ii) a source that identifies the specific dataset the pattern relates to; (iii) a measure that estimates the quality of the raw data representation achieved by the pattern; (iv) an expression which relates the pattern to the source data. In particular, the expression is obtained by the formula $f$ in the pattern type by (i) instantiating each attribute appearing in $ss$ with the corresponding value, and (ii) letting the attributes appearing in $ds$ range over the source space. Dot notation and path expressions can be used to denote pattern components.

A *class* is a set of semantically related patterns and constitutes the key concept in defining a pattern query language. A class is defined for a given pattern type and contains only patterns of that type. A pattern may belong to any number of classes. If it does not belong to any class, it cannot be queried.

*Example 1.* Given a domain $D$ of values and a set of product sales transactions, a possible pattern type for modeling association rules is the following:

n: AssociationRule
ss: TUPLE(head: SET(STRING), body: SET(STRING))
ds: BAG(transaction: SET(STRING))
ms: TUPLE(confidence: REAL, support: REAL)
f: $\forall x(x \in \mathsf{head} \vee \mathsf{x} \in \mathsf{body} \Rightarrow \mathsf{x} \in \mathsf{transaction})$

The structure schema is a tuple modeling the head and the body as strings representing products . The source schema specifies that association rules are constructed from a bag of transactions, each defined as a set of products. The measure schema includes two common measures to assess the relevance of a rule: its confidence (what percentage of transactions including the head also include

the body) and its support (what percentage of the whole set of transactions include both the head and the body). Finally, the formula represents (exactly, in this case) the pattern/dataset relationship by associating each rule with the set of transactions which support it.
Now suppose that raw data include a relational database containing a table sales which stores data related to sales transactions in a sport shop with scheme: (transactionId, article, quantity). Using an extended SQL syntax to denote the dataset, an example of an instance of AssociationRule is:

pid: 512
s: (head={'Boots'}, body={'Socks','Hat'})
d: 'SELECT SETOF(article) AS transaction FROM sales GROUP BY transactionId'
m: (confidence=0.75, support=0.55)
e: $\{\text{transaction} : \forall x(x \in \{'\text{Boots}', '\text{Socks}', '\text{Hat}'\} \Rightarrow x \in \text{transaction})\}$ □

To increase the expressivity of the logical model some interesting relationships between patterns have also been introduced [1]. Among them, we recall: *specialization*, a sort of *IS-A* relationship (e.g. clusters of integer points can be seen as a specialization of generic cluster of points); *composition*, between a pattern and those used to define its structure; *refinement*, between a pattern and those belonging to its source component. Composition and refinement support the definition of complex patterns.

*Example 2.* The following pattern type models clusters of association rules:

n: ClusterOfRules
ss: representative: AssociationRule
ds: SET(rule: AssociationRule)
ms: TUPLE(deviationOnConfidence: REAL, deviationOnSupport: REAL)
f: rule.ss.head=representative.ss.head

There is a refinement relationship between ClusterOfRules and AssociationRule, since the data source of a cluster of rules is a set of association rules. In addition, since the representative of each cluster is an association rule, there exists a composition relationship between those two pattern types. □

## 3 Preliminaries

In the following, some preliminary notions useful in the definition of pattern languages are introduced. We introduce the concept of *mining function*, i.e. a function used to generate patterns from raw data, and we discuss how information concerning mining functions can be stored in catalogs within the PBMS.
**Mining function.** Given a pattern type $pt$, a *mining function* $\mu$ for $pt$ takes as input a data source, applies a certain computation to it, and returns a set of patterns, instances of $pt$. Note that a mining function generates not only the structure but also the measures associated with patterns, according to the chosen pattern type. However, sometimes, given a pattern, it is important to be able to recompute its associated measures against a certain dataset, which may either be the one over which the pattern has been generated or another one. It is useful to define a mining function starting from two additional functions: (i) a

*structure function*, computing the structure pattern components; (ii) a *measure function*, computing the measure pattern components. For formal details see [1].
**Catalogs over patterns.** In order to be able to effectively use information concerning mining functions during the pattern generation process, we must store somewhere information concerning them. In general, given a pattern type, several structure and measure functions (thus, several mining functions) can be defined. We assume there exist some libraries of structure and measure functions for each pattern type, stored in some PBMS catalog, since such information is useful from an operational point of view but it is not queried by final users.

Catalogs are created for both the pattern type layer and the pattern layer. Function libraries constitute the content of the pattern type layer catalog. We assume there exists one library for each pattern type. On the other hand, the catalog for the pattern layer must contain, for each pattern, the mining function used to generate it. Thus, the catalog is a set of tuples, specifying: (i) the pattern identifier (pid); (ii) the mining function used to generate the pattern; (iii) the measure functions, if different from those applied by the mining function; (iv) the pattern transaction time, i.e. the instant of time at which the pattern instance has been generated and inserted in the system.

## 4 Pattern Manipulation Language

A *pattern manipulation language* (PML) must support primitives to generate patterns from raw data and to insert them in the PBMS, to delete and to update patterns. In the following, we discuss each PML operator in details. Their formal definitions are provided in [1]. For the sake of simplicity, all the PML operations are defined assuming to deal with a single pattern at a time. All the proposed operations can of course be extended to deal with sets of patterns.

### 4.1 Insertion operators

Three different types of insertion are supported. The first one (extraction) generates new patterns starting from raw data, by applying a mining function. The second one (direct insertion) allows one to insert in the PBMS patterns from scratch. Finally, the third one (recomputation) generate new patterns from existing ones, by recomputing their measure over a different source dataset.
**Extraction $\mathcal{E}$.** The extraction operator $\mathcal{E}(pt, d, cond, \mu)$ allows the extraction of patterns, of a specific pattern type $pt$, from a raw dataset $d$, by applying a specific mining function $\mu$, and their insertion in the pattern layer if they satisfy a specified condition $cond$. Note that, once the mining function $\mu$ has been fixed, the pattern type $pt$ of the resulting patterns is determined by the function signature. The data sources of the generated patterns correspond to the source description specified as input for the extraction operation. As a side effect, new tuples are inserted in the catalog of the pattern layer. Finally, note that the extraction operator does not directly insert resulting patterns in any class, but only in the pattern layer.

**Direct Insertion $\mathcal{I}$.** The direct insertion operator $\mathcal{I}(pt, d, s, m)$ allows one to insert a user-defined pattern of a certain pattern type in the pattern layer. It takes as input a pattern type $pt$, a source $d$, a structure $s$, and a measure $m$ value and inserts the pattern generated from those data in the pattern layer. As a side effect, new tuples are inserted in the catalog of the pattern layer. Since they have not been generated by using a mining function, all these tuples contain no mining function information.
**Recomputation $\mathcal{R}$.** In a pattern-based system, given some pattern $p$, the user may be interested in establishing whether it holds for a specified dataset $d$, different from the one the pattern has been generated from, and computing the new measures, accordingly to some measure functions $\mu_m$ specified as input. This operation is denoted by $\mathcal{R}(p, d, \mu_m)$. In this case, the resulting patterns have the same structure than the input ones but the data source and the measures are different. As a side effect, a new tuple is inserted in the pattern layer catalog. The mining function for the new pattern is the same used for the one in input.

### 4.2 Deletion and update operators

In the PBMS context, the delete operation has the same meaning than in a traditional DBMS context, i.e. patterns satisfying specific conditions are selected and removed from the pattern layer. However, since a pattern can belong to different classes, the following two types of deletion operation have been introduced.
**Deletion_ Restricted $\delta_R$.** It allows the user to remove a pattern $p$ from the pattern layer only if it is contained in no classes (denoted by $\delta_R(p)$).
**Deletion_Extended $\delta_E$.** It allows the user to remove a pattern $p$ from the pattern layer and from all the classes it belongs to (denoted by $\delta_E(p)$).
**Synchronize $\mathcal{S}$.** The synchronize operator $\mathcal{S}(p, \mu_m)$ is an update operator which allows the user to re-compute the measure values associated with a pattern $p$, in order to reflect modifications that occurred in its data source, by using a measure function $\mu_m$ specified as input. Note that only measures are modified, the pid, the structure, the data source, and the expression do not change. Thus, it can be seen as a special case of recomputation, applied against the pattern data source. As a side effect of the synchronization operation, the measure functions specified in the tuple corresponding to the input pattern are updated.

### 4.3 Operators for class

Since, according to the pattern model, a class is a set of semantically related patterns sharing the same pattern type (see Section 2), each pattern can be inserted in an arbitrary number of classes. However, it must be inserted in at least one class in order to be queried. Therefore, two PML operations supporting the insertion and the removal of a pattern into or from a class are provided.
**Insertion Into Class $\mathcal{I}_\mathcal{C}$.** It allows one to insert a fixed pattern $p$ into a specific class $c$ (denoted by $\mathcal{I}_\mathcal{C}(p, c)$).
**Deletion From Class $\mathcal{D}_\mathcal{C}$.** It allows one to delete a specific pattern $p$ from a class $c$ it belongs to (denoted by $\mathcal{D}_\mathcal{C}(p, c)$).

*Example 3.* Let $Q_{Rome}$ and $Q_{Milan}$ be two intensional descriptions of two different datasets concerning sales in stores in Rome and Milan. Let $\mu_{aPriori}$ be the mining function corresponding to the *A-Priori* algorithm for computing association rules and let $\mu_M$ be a measure function for pattern type AssociationRule. Let $AR_R$, $AR_M$, and $AR$ classes containing association rules. The following are some examples of interesting PML operations:

1. A set of association rules ($SP$) about sales in a department store in Rome are extracted and inserted into $AR_R$:
   - $SP = \mathcal{E}(AssociationRules, Q_{Rome}, \emptyset, \mu_{aPriori})$
   - $\mathcal{I}_C(SP, AR_R)$
2. New patterns $SP'$ are generated by recomputing $SP$ over the dataset concerning sales in a department store in Milan. Then, they are inserted into classes $AR_M$ and $AR$. Finally, patterns $SP$ are removed from the system.
   - $SP' = \mathcal{R}(SP, Q_{Milan}, \mu_M)$
   - $\mathcal{I}_C(SP', AR_M)$
   - $\mathcal{I}_C(SP', AR)$
   - $\delta_E(SP)$
3. The patterns $SP$ can be synchronized to reflect changes in the data source:
   - $SP = \mathcal{S}(SP, \mu_M)$ □

## 5 Pattern Query Language

The Pattern Query Language supports the retrieval of patterns from the pattern layer. In order to identify the main operations that could be useful in pattern retrieval, we present a pattern query algebra. Algebraic operators take as input classes, thus sets of patterns, and return a new set of patterns.

Besides operators manipulating patterns, for real application purposes it is also important to support operations binding patterns with raw data. Such operations are known as *cross-over queries* since for their execution two different systems, the PBMS and the system where raw data rely, have to be used.

### 5.1 Basic Pattern Operators

**Traditional relational operators**. Since classes are sets, usual relational operators such as union, difference, and intersection are defined for pairs of classes over the same pattern type.

**Projection**. The projection operator allows one to reduce the structure and the measures of the input patterns by projecting out some components. The new expression is obtained by projecting the formula defining the expression over the remaining attributes [7]. Note that no projection is defined over the data source, since in this case the structure and the measures would have to be recomputed.

Let $c$ be a class over pattern type $pt$. Let $ls$ be a non empty list of attributes appearing in $pt.ss$, $lm$ a list of attributes appearing in $pt.ms$. Then, the projection operator is defined as follows:

$\pi_{(ls,lm)}(c) = \{(new(), \pi^s_{ls}(s), d, \pi^m_{lm}(m), \pi_{ls \cup lm}(e)) | \exists p \in c, p = (pid, s, d, m, e)\}$.
In the previous formula, $new()$ is a function returning new pattern identifiers, $\pi^m_{lm}(m)$ is the usual relational projection and $\pi^s_{ls}(s)$ is defined as follows: (i) if $s$ is of type TUPLE(t), then $\pi^s_{ls}(s)$ is the usual relational projection; (ii) if $s$ is of type SET(t) or BAG(t), then $\pi^s_{ls}(s)$ is obtained by removing from $s$ the unrequired components, maintaining the existing nesting.

*Example 4.* Let $A$ a class over the pattern type AssociationRule. The projection $\pi_{(<head>,<support>)}(A)$ returns a set of patterns whose structure contains only the *head* component and whose measure contains only the *support* one. □

**Selection**. The selection operator allows one to select the patterns belonging to a certain class satisfying a certain condition, involving any possible pattern component. Conditions are formulas constructed over the following atomic formulas, by using $\vee, \wedge$, and $\neg$:

- $t_1 \theta v$, $t_1 \theta t_2$, where $t_1, t_2$ are path expressions starting from $ss$ or $ms$ components, $t_1$ and $t_2$ denote elements having the same type $t$ (or two compatible types), $v$ is a value for $t$, and $\theta$ is a suitable operator for $t$. If $t$ is not a base type (thus, it is a pattern type), the allowed operators are: (i) identity equality ($=^i$); (ii) shallow and deep value equality ($=^{se}$ and $=^{de}$); (iii) similarity ($=^s$). The previous operators can also be directly applied to patterns belonging to a class. To this purpose, we assume that unary operators syntactically and semantically equivalent to the ones presented above exist.
- $t_1 \theta t_2$, where $t_1, t_2$ denote either data source or expression components and $\theta \in \{\equiv, \subseteq\}$. $\equiv$ stands for equivalence and $\subseteq$ for containment between intensional data source descriptions (i.e., between queries). Note that equality or containment are checked by considering the intensional definition of data sources, without accessing raw data (thus, we assume that selection is not a cross-over operation). A similar consideration holds for expression components.

Let $c$ be a class over pattern type $pt$ and $F$ a formula. The selection of $c$ with respect to $F$ is denoted as $\sigma_F(c)$ and is defined as:
$\sigma_F(c) = \{p | p \in c \; and \; p \; satisfies \; F\}$

*Example 5.* The query: *find all association rules belonging to class* AR1 *whose body contains "Boots" or whose confidence is greater than 0.75* can be expressed as: $\sigma_{'Boots' \; IN \; s.body \; OR \; m.confidence > 0.75}(AR1)$. □

**Drill-Down**. The drill-down operator allows one to navigate the refinement relationship between patterns, from a pattern to some of the patterns it refines. More formally, let $c$ be a class over pattern type $pt$ and let $a$ be an attribute appearing in the source component, $ds$, associated with the pattern type. Then the drill-down operation, denoted by $\delta_a(c)$, returns the following set:
$\delta_a(c) = \{p | \exists p' \in c, p' = (pid, s, d, m, e), d.a_1....a_n.a = p\}$[1].

[1] $d.a_1....a_n.a$ denotes a path expression starting from the patterns data source component and ending at attribute $a$.

*Example 6.* Consider the refinement relationship between ClusterOfRules and AssociationRule. Let $C$ be a class of type ClusterOfRules: $\delta_{rule}(C)$ returns a set of association rules. □

**Roll-Up**. The roll-up operator allows one to navigate the refinement relationship between patterns, from a pattern $p$ to some of the patterns obtained by refining $p$. More formally, let $c$ be a class over pattern type $pt_1$ and let $pt_2$ be a pattern type refining $pt_1$. The roll-up operator, when applied to $c$, returns the pattern instances of pattern type $pt_2$ refining at least one pattern belonging to $c$. The roll-up operation is formally defined as follows:

$\rho_{pt}(c) = \{p' | \exists p \in c, \exists$ a class $c'$ over $pt$ such that $\exists p' \in c', p'$ refines $p\}$.

*Example 7.* Consider a class $C$ of type AssociationRule, $\rho_{\mathsf{ClusterOfRules}}(C)$ returns a set of patterns of type ClusterOfRules refining at least one pattern in $C$. □

**Decomposition**. The decomposition operator allows one to navigate the composition relationship, returning for each pattern a component pattern associated with a certain attribute. More formally, let $c$ be a class over pattern type $pt$ and let $a$ be an attribute appearing in the structure schema, $ss$, associated with a pattern type. The decomposition operation is denoted by $\mathcal{C}_a(c)$ and it is formally defined as follows:

$\mathcal{C}_a(c) = \{p | \exists p' \in c, p' = (pid, s, d, m, e), s.a_1....a_n.a = p\}$

*Example 8.* Consider a class $c$ over pattern type ClusterOfRules, $\mathcal{C}_{representative}(c)$ returns a set of patterns of pattern type AssociationRule. □

**Join**. The join operation provides a way to combine patterns belonging to two different classes according to a join predicate specified by the user.

Let $c_1$ and $c_2$ be two classes over two pattern types $pt_1$ and $pt_2$. In order to formally define the join operator, we need to introduce the concept of *join predicate* and *composition function*. A join predicate $F$ is any selection condition defined over a component of patterns in $c_1$ and a component of patterns in $c_2$. A composition function $c()$ for pattern types $pt_1$ and $pt_2$ is a 4-tuple of functions $c = (c_{ss}, c_{ds}, c_{ms}, c_f)$, one for each pattern component. For example, function $c_{ss}$ is defined as $c_{ss} : Dom(pt_1.ss) \times Dom(pt_2.ss) \rightarrow \mathcal{SS}$, i.e., it takes as input two structure values of the right type and return a new structure value, for a possible new pattern type, generated by the join. Functions for the other pattern components are similarly defined. Given two patterns $p_1 = (pid1, s1, d1, m1, f1) \in c_1$ and $p_2 = (pid2, s2, d2, m2, f2) \in c_2$, $c(p_1, p_2)$ is defined as the pattern $p$ with the following components, $s : c_{ss}(s1, s2) d : c_{ds}(d1, d2) m : c_{ms}(m1, m2) f : c_f(f1, f2)$

The join of $c_1$ and $c_2$ with respect to the join predicate $F$ and the composition function $c$, denoted by $c_1 \bowtie_{F,c} c_2$, is now defined as follows:

$c_1 \bowtie_{F,c} c_2 = \{c(p_1, p_2) | p_1 \in c_1 \wedge p_2 \in c_2 \wedge F(p_1, p_2) = true\}$.

Similarly to the relational context, the *Natural Join* can be defined as a special type of join. We assume it can be applied only to classes and pattern types having the same data source schema. The natural join is defined as $c_1 \bowtie_{F,c} c_2$,

where: $F$ is the join predicate requiring the equality between data sources and for attributes with the same name and type in the structure components; $c()$ is the following composition function: $c = (c_{<,>}, \downarrow_d, \cup_{null}, \wedge)$, where: (i) $c_{<,>}$ returns a record with two components, one for each input structure value; (ii) $\downarrow_d$ is a function that takes two patterns with the same data source and returns it; (iii) $\cup_{null}$ returns a record with one component for each input measure and assign a "*null*" to all of them (thus, measures are not recomputed for the new dataset); (iv) $\wedge$ is the logical conjunction.

Since in a natural join operation $F$ and $c$ are fixed, the natural join notation can be simplified. Thus, it is simply denoted by $c_1 \bowtie c_2$.

*Example 9.* Consider two classes $c_1, c_2$ on pattern type AssociationRules. Suppose we want to generate new association rules by transitive closure, i.e., informally, given two rules $A \rightarrow B \in c_1$ and $B \rightarrow C \in c_2$, we want to generate $A \rightarrow C$. Such rules can be generated through a join operation $c_1 \bowtie_{F,c} c_2$ applied to $c_1$ and $c_2$ with join predicate $F \equiv c_1.ss.body = c_2.ss.head$ and using a specific composition function $c$. A reasonable composition function generates the new structures as described above. Data sources can be combined by considering the natural join between the input sources since the new rule will refer only to the tuples belonging to both datasets. Null values can be assigned to measures if we do not want to recompute them on the new dataset. Finally, expressions can be combined by considering the intersection of the input ones. The resulting patterns are of type AssociationRules.

Note that if patterns belonging to $c_1$ and $c_2$ are generated from the same data source, the natural join of $c_1$ and $c_2$ ($c_1 \bowtie c_2$) can be computed. In this case, the obtained result is similar to the one obtained by applying the join. The only difference concerns the structure of the generated patterns: for the natural join, it is a record with two components, that are association rules. □

### 5.2 Cross-over queries

Cross-over queries are a particular type of queries that correlate patterns with raw data, providing a way for navigating from the pattern layer to the raw data layer and vice versa. In the following, we present some cross-over operators that from our point of view are quite useful in real applications.

**Drill-Through.** The drill-through operation allows one to navigate from the pattern layer to the raw data layer, through pattern data sources and expressions. Thus, it takes as input a class and it returns a raw data set. More formally, let $c$ be a class over pattern type $pt$ and let $a$ be an attribute associated with a base data type in $ds$. Then, the drill-through operation is denoted by $\gamma_a(c)$ and it is formally defined as: $\gamma_a(c) = \{\pi_a(d') | \exists p \in c, p = (pid, s, d, m, e), d' = e(d)\}$. In the previous expression, $e(d)$ represents the result of the evaluation of expression $e$ against the (extensional) data source of the considered pattern. As a special case, we assume that $\gamma(c)$ projects the data source over all its attributes.

**Covering.** The covering operation allows one to determine whether a given pattern holds for a specified data source. Let $p$ be a pattern, possibly selected by using other query language operators, and $d$ a data source. The covering operation, denoted by $\vartheta(p, d)$, is defined as: $\vartheta(p, d) = true$ iff $p \in \mathcal{E}(pt, p, true, \mu)$ where $pt$ is the pattern type of $p$ and $\mu$ is the mining function used to generate $p$ (retrieved from the catalog).

**PML operations reinterpreted as PQL operations.** Extraction, recomputation, and synchronization can be interpreted as PQL operators. In this case, the generated patterns are not inserted in the pattern base. Rather, they are returned as query results.

*Example 10.* Let $AR$ be a class of type AssociationRule, and $CR$ be a class of type ClusterOfRules. Let $a_1$ and $a_2$ be two association rules in $AR$. The following are some examples of interesting queries over $AR$ and $CR$.
*Q1* - Find rules containing "Bread" in the head or having a support greater than 0.6, belonging to the dataset by which patterns in class $CR$ have been mined:

$\sigma_{'Bread'\ IN\ ss.head\ OR\ ms.support>0.6}(\delta_{rule}(CR))$

*Q2* - Retrieve all association rules belonging to class $AR$ having at least confidence 0.7 and whose body contains "Boots", and project the result over the body structure component and the support measure component:

$\pi_{(<head>,<support>)}(\sigma_{'Boots'\ IN\ ss.body\ AND\ ms.confidence>0.75}(AR))$

*Q3* - Retrieve all representatives of clusters of rules refining at least one association rule belonging to class $AR$ mined from the dataset doneted by $Q\_Rome$:

$\mathcal{C}_{representative}(\rho_{ClusterOfRules}(\sigma_{Q\_Rome \subseteq d}(AR)))$

*Q4* - After computing the join presented in Example 9, it is possible to recompute the measures by using a cross-over operation ($\mu_m$ is a measure function for association rules): $\mathcal{S}(c_1 \bowtie_{F,c} c_2, \mu_m)$. □

## 6 Related work

As we have already discussed, the PANDA approach [9] introduces a logical separation between raw data and patterns in order to efficiently manage both of them through dedicated management systems. Differently, in the inductive databases, data and patterns are stored together [5, 3] and the knowledge discovery process is modeled as an interactive process in which users can query data as well as patterns. In this context, no attempt is made toward a general model for heterogeneous patterns. Rather, inductive databases rely on specific types of patterns (i.e. association rules, clusters, functional dependencies). Also approaches which claim to be general enough to cover all possible pattern types, such as [2], are then instantiated over few of them.

Due to the considered models, most of the proposed languages deal with specific types of patterns, by usually using an SQL-like syntax [6, 5, 8]. In those cases, standard query languages are extended with fixed operators supporting the pattern mining process from a collection of relational data. No user-defined

mining function can be specified [4]. In [2] relational algebra has been extended with operators specifying mining tasks, selecting patterns according to specified conditions, and combining data and patterns in the same query (cross-over queries). However, the proposed operators do not seem to cover all possible queries that could be useful from an application point of view.

Another approach for querying patterns has been presented in [10], where a pattern algebra has been presented. However, no formal pattern model is provided and the proposed operations seem too simple to be useful in practice. Finally, we remark that none of the proposed languages takes into account synchronization aspects and hierarchical relationships between patterns.

## 7 Conclusions and future work

In this paper, we have proposed some basic operators to cope with pattern manipulation and retrieval. The proposed operators cover and extend the operations already proposed in the literature to deal with patterns of arbitrary type. Future work includes the definition of a pattern calculus, equivalent to the proposed algebra and the analysis of expressive power and complexity of the proposed languages. Furthermore, we plan to investigate issues concerning soundness and completeness of the proposed pattern query language.

## References

1. E. Bertino, B. Catania, M. Golfarelli, M. Halkidi, A. Maddalena, S. Skiadopoulos, S. Rizzi, M. Terrovitis, P. Vassiliadis, M. Vazirgiannis, and E. Vrachnos. The Logical Model for Patterns. Technical Report TR-2003-02, PANDA, 2003.
2. J. F. Boulicaut, M. Klemettinen, and H. Mannila. Modeling KDD Processes within the Inductive Database Framework. In *Proc. of the Data Warehousing and Knowledge Discovery*, pages 293–302, 1999.
3. L. De Raedt. A Perspective on Inductive Databases. *ACM SIGKDD Explorations Newsletter*, 4(2):69–77, 2002.
4. B. Goethals and J. Van den Bussche. A Priori versus a Posteriori Filtering of Association Rules. In *Proc. of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 1999.
5. T. Imielinski and H. Mannila. A Database Perspective on Knowledge Discovery. *Communications of the ACM*, 39(11):58–64, 1996.
6. T. Imielinski and A. Virmani. MSQL: A Query Language for Database Mining. *Data Mining and Knowledge Discovery*, 2(4):373–408, 1999.
7. P. Kanellakis, G. Kuper, and P. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, 51(1):25–52, 1995.
8. R. Meo, G. Psaila, and S. Ceri. An Extension to SQL for Mining Association Rules. *Data Mining and Knowledge Discovery*, 2(2):195–224, 1999.
9. S. Rizzi, E. Bertino, B. Catania, M. Golfarelli, M. Halkidi, M. Terrovitis, P. Vassiliadis, M. Vazirgiannis, and E. Vrachnos. Toward a Logical Model for Patterns. In *Proc. of the 22nd International Conference on Conceptual Modeling (ER 2003)*, Chicago, 2003.
10. A. Tuzhilin. A Pattern Discovery Algebra. In *Proc. of the 1997 Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 71–76, 1997.