Modeling and Language Support for the Management of Pattern-Bases[*]

Manolis Terrovitis[†] Panos Vassiliadis[‡] Spiros Skiadopoulos[†] Elisa Bertino[§]
Barbara Catania[¶] Anna Maddalena[¶]

Abstract

In our days knowledge extraction methods are able to produce artifacts (also called patterns) that concisely represent data. Patterns are usually quite heterogeneous and require ad-hoc processing techniques. So far, little emphasis has been posed on developing an overall integrated environment for uniformly representing and querying different types of patterns. Within the larger context of modelling, storing, and querying patterns, in this paper, we: (a) formally define the logical foundations for the global setting of pattern management through a model that covers data, patterns and their intermediate mappings; (b) present a pattern specification language for pattern management along with safety restrictions; and (c) introduce queries and query operators and identify interesting query classes.

1. Introduction

Nowadays, we are experiencing a phenomenon of information overload, which escalates beyond any of our traditional beliefs. As a recent survey states [13], the world produces between 1 and 2 exabytes of unique information per year, 90% of which is digital and with a 50% annual growth rate. To deal with the vast amounts of collected data, we reduce the available data to *knowledge artifacts* (e.g., clusters, association rules) through data processing methods (pattern recognition, data mining, knowledge extraction) that reduce their number and size (so that they are manageable by humans), while preserving as much as possible from their hidden/interesting/available information. Again, the volume, diversity and complexity of these knowledge artifacts make their management by a DBMS-like environment imperative. In the remainder of this document, we will refer to all these knowledge artifacts as *patterns*.

So far, patterns have not been adequately treated as persistent objects that can be stored, retrieved and queried. Thus, the challenge of integration between patterns and data seems to be achievable by designing fundamental approaches for providing database support to pattern management. In particular, since patterns represent relevant knowledge, often very large in size, it is important that such knowledge is handled as first-class citizen. This means that patterns should be modelled, stored, processed, and queried, in a fashion analogous to data in traditional DBMSs.

To this end, our research is focused mainly towards providing a generic and extensible model for patterns, along with the necessary languages for their definition and manipulation. In this context, there is already a first attempt towards a model for pattern management [15], which is able to support different forms of patterns (constraints, functions, etc.) as the new data types of the PBMS. In this paper, we provide formal foundations for the above issues by the following means: First, we formally define the logical foundations for the global setting of PBMS management through a model that covers data, patterns, and intermediate mappings. Second, we discuss language issues in the context of pattern definition and management. In particular, we present a pattern specification calculus that enables us to specify pattern semantics in a rich and concise way. Safety issues are also discussed in this context. Finally, we introduce queries and identify interesting query classes for the problem. We introduce query operators for patterns and discuss their

* This work was partially funded by the European Commission and the Hellenic Ministry of Education through EPEAEK II and the PANDA IST Thematic Network.

† School of Electrical and Computer Engineering, Nat'l Technical Univ. of Athens, Zographou 157 73 Athens, Hellas, {mter,spiros}@dblab.ntua.gr

‡ Dept. of Computer Science, Univ. of Ioannina, Ioannina, Hellas, pvassil@cs.uoi.gr

§ CS Department, Purdue University, bertino@cs.purdue.edu

¶ Dept. of Computer and Information Science, Univ. of Genoa, Italy, {catania,maddalena}@disi.unige.it

usage and semantics.

The rest of this paper is organized as follows. In Section 2 we introduce the notion of Pattern-Base Management system. Section 3 presents a generic model for the coexistence of data and patterns. In Section 4 we present the Pattern Specification Language, that enables us to specify pattern semantics. In Section 5 we explain how patterns can be queried and introduce query classes and operators. In Section 6 we present related work. Finally, Section 7 offers conclusions and topics of future work. A long version of this work appears in [18].

2. Patterns and Pattern-Base Management Systems (PBMS's)

Patterns can be regarded as artifacts, which describe (a subset of) raw data with similar properties and/or behavior, providing a compact and rich in semantics representation of data. There exists a *data space* (the space of raw data) and a *pattern space*. Patterns and data are related through many-to-many relationships.

Patterns can be managed by a *Pattern-Base Management System* (PBMS) exactly as database records are managed by a database management system. In our setting, a PBMS can be envisaged as a system where:

- patterns are (semi-)automatically extracted from raw data and loaded in the PBMS;
- patterns are updated as new (existing) data are loaded into (deleted from or updated in) the raw database. These updates can be done in an ad-hoc, on-demand or periodical (batch) manner;
- users are enabled to define the internal structure of the PBMS through an appropriate definition language;
- users are allowed to pose queries to and retrieve answers from the PBMS, with the results of these answers properly visualized and presented;
- an (approximate or exact) mapping between patterns and raw data is available whenever retrieval of raw data corresponding to patterns is needed.

The reference architecture for a PBMS is depicted in Fig. 1 and consists of three major layers of information organization. In the bottom of Fig. 1, we depict the *data space* consisting of data stores that contain raw data (forming thus, the Raw Data Layer). Raw data can be either managed by a DBMS or can be stored in files, streams or any other physical mean that is managed outside a DBMS. At the top of Fig. 1, we depict the PBMS repository that corresponds to the *pattern space* and contains patterns. The PBMS repos-

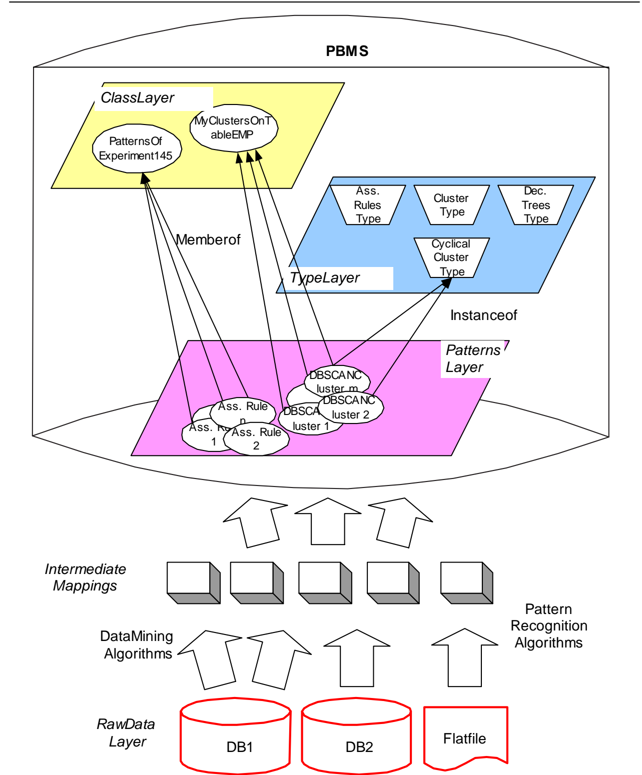

Figure 1. Reference architecture for the Pattern-Base Management System

itory is managed by the Pattern-Base Management System. Finally, in the middle of Fig. 1, we can observe the *intermediate mappings* that relate patterns to their corresponding data, forming the Intermediate Data Layer. Intermediate mappings facilitate the justification of any knowledge inferred at the PBMS with respect to the raw data. The overall architecture is called *Integrated Pattern-Base Management Architecture*, or simply *Pattern Warehouse*.

Next, we present a brief description of all the entities that appear in this abstract description of the PBMS:

Intermediate Mappings Layer. Ideally, we would like this layer to be part of the PBMS, involving specialized storage and indexing structures. In general, one can imagine that the intermediate mappings can be either precisely traced (e.g., through some form of join index between patterns and data) or imprecisely approximated. In the latter case, one can employ different variations of these approximations through data reduction techniques (e.g., wavelets), summaries, or even on-line data mining algorithms. For practical purposes, though, the PBMS should be constructed in such a way

that it functions even if intermediate results are out of its control (which we would expect as the most possible scenario in real-world scenarios), or even absent.

Pattern Layer. Patterns are compact and rich in semantics representations of raw data. In the general case, although not obligatorily, patterns are generated through the application of knowledge extraction algorithms. In Fig. 1, two such algorithms have been applied: an algorithm for the extraction of association rules and an algorithm for the extraction of cyclical clusters.

Type Layer. The PBMS Pattern Types describe the intentional definition, i.e., the syntax of the patterns. Patterns of same type share similar characteristics, therefore Pattern Types play the role of data types in traditional DBMS's or object types in OODBMS's. The type layer should be extensible to be able to incorporate user-defined pattern types.

Class Layer. The PBMS classes are collections of patterns which share some semantic similarity. Patterns that are members of the same class are obligatorily required to belong to the same type. Classes are used to create patterns with predefined semantics given by the designer; by doing so, the designer makes it easier for the users to work on them in a meaningful way. For example, a class may comprise patterns that resulted from the same experiments, like the association rules of Fig. 1.

3. Modeling Data and Patterns

In this section, we will give the formal foundations for the treatment of data and patterns within the unifying framework of a pattern warehouse. First, we introduce the notions of data types, attributes and relations (in the usual relational sense). We exploit the definitions already given in [5] for this purpose. Then, we formally define pattern types, pattern classes, patterns as well as the intermediate mappings. Finally, we define pattern bases and pattern warehouses.

3.1. The Data Space

In this section, we will deal with the formal definition of the entities of the *data space*, i.e., data types, relations and databases. Practically, we start with the data model proposed by Abiteboul and Beeri in [5], and make some changes in the type definitions. This data model is a many-sorted model which facilitates the definition of complex values. Our minor changes focus on inserting *names* in the types definitions, so we can easily access the inner components of complex types. In the data model, each constant and each variable is associated with a type and each function and predicate with a signature. We start by introducing simple data types.

Data types are structured types that use domain names, *set* and *tuple* constructors and attributes. For reasons of space and simplicity, we focus our examples in the domains of integers and reals, throughout the rest of the paper.

Definition 1 Data Types *(or simply,* types*) are defined as follows [5]:*

- *If $\widehat{D}$ is a domain name and A is an attribute name then $A : \widehat{D}$ is an* atomic *type.*
- *If $T_1, \ldots, T_n$ are types and $A, A_1, \ldots, A_n$ are distinct attribute names then $A : [A_1{:}T_1, \ldots, A_n{:}T_n]$ is also a type, called* tuple *type.*
- *If T is a type and A is an attribute name then $A : \{T\}$ is also a type. We call these types* set *types.*

For a k-ary predicate the signature is a k-tuple of types and for a k-ary function it is a $k+1$-tuple of types.

The *values* of a specific type are defined the natural way. For atomic types, we assume an infinite, countable set of values as their domain, which we call **dom**(T). The domain of set types is defined as the powerset of the domain of the composing type. The domain of tuple types is defined as the product of their constituent types.

Relations in our setting are considered to be sets of tuples defined over a certain composite data type. We model relations in the object-relational context and we make the following assumptions:

- For reasons of simplicity, we assume that relations are sets of tuples of the same type instead of just sets (in contrast with [5] which requires only that relations are sets).
- At least one of the tuple components, by default named *RID*, is atomic and all its values are unique. Intuitively, we want each relation tuple to have a row identifier, according to classical (object-) relational terminology. This enables us to use just sets instead of *bags*. We consider *RID* to be an implicit attribute and we do not explicitly refer to it when we define the data schema.

Definition 2 ([5]) *A* database schema *is a pair $\widehat{DB} = \langle [\widehat{D_1}, \ldots, \widehat{D_k}], [\widehat{R_1}{:}T_1, \ldots, \widehat{R_n}{:}T_n] \rangle$, where $T_1, \ldots, T_n$ are set types involving only the domain names corresponding to the data types $\widehat{D_1}, \ldots, \widehat{D_k}$ and $\widehat{R_1}, \ldots, \widehat{R_n}$ are relation names.*

Definition 3 ([5]) *An instance of* $\widehat{DB}$*, i.e., a* database*, is a structure* $DB = \langle [D_1, \ldots, D_k], [R_1, \ldots, R_n] \rangle$*, where* R_i*'s are relations and* D_i*'s are domains.*

We also refer to $\widehat{DB}$ as the *database type*, and to DB as the *database value*.

As we will see in the following, we need to be able to define patterns over joins, projections and selections over database relations. To do that, we extend Definition 2 with a set of *materialized views* $V_1, \ldots, V_m$ which are defined over relations $R_1, \ldots, R_n$ using relational algebra. We require that each tuple in a materialized view has a unique identifier called *RID*. Throughout the rest of the paper, we address views as relations, unless explicitly specified otherwise.

3.2. The Pattern Space

Now that we have defined the constructs found at the data space, we are ready to proceed with the definition of the entities belonging to the pattern space. Therefore, we will introduce pattern types, which are templates for the actual patterns and pattern classes which are groups of semantically similar patterns. The definition of a pattern base then comes straightforwardly.

Patterns as defined in [15] are compact, yet rich in semantics, representations of the raw data. This informal principle is formally translated as a quintuple. We will intuitively define these components here and give the definition right next.

- First, a pattern is uniquely identified by a *Pattern Id* (*PID*).
- Second, a pattern has a *structure*: for example, an association rule comprises a *head* and a *body*, and a cyclical cluster comprises a *center* and a *radius*.
- Third, a pattern corresponds to some underlying *data*. The subset of the underlying data space that is represented by the pattern must be specified, e.g., through the appropriate relation.
- Fourth, a pattern informs the user on its quality, i.e., how closely does it approximate reality as compared to the underlying data, through a set of statistical *measures*. For example, an association rule is characterized by a confidence and a support measure.
- Finally, a $formula$ provides the richness in semantics for the pattern. The $formula$ demonstrates a possibly simplified form, of the relation between the data that are represented by the pattern and the pattern structure. In Section 4 we present a *Pattern Specification Language (PSL)* in which the formula is expressed.

A *Pattern Type* represents the intentional description of a pattern, pretty much like abstract data types do in the case of object-relational data. In other words, a pattern type acts as a template for the generation of patterns. Each pattern is an *instance* of a specific pattern type. There are four major components that a Pattern Type specifies.

- First, the pattern type dictates the structure of its instances, through a *structure schema*. For example, it obliges association rules to comprise a head and a body.
- Moreover, a pattern type specifies a *data schema* which dictates the schema of the underlying data which have produced the pattern type; practically this is the schema of the relation which can be employed as the test-bed for pattern generation/definition.
- Third, it dictates a *measure schema*, i.e., which set of statistical measures that quantify the quality of the approximation is employed by the instances of the pattern type.
- Finally, a template for the formula of the instances dictates the structure of the formula. The formula is a predicate bounding the *DataSchema* and the $StructureSchema$, expressed in the PSL.

Definition 4 *A* Pattern Type *is a quintuple [*$Name$*,* $StructureSchema$*,* $DataSchema$*,* $MeasureSchema$*,* $Formula$*] such that (a)* $Name$ *is a unique identifier among pattern types, (b)* $StructureSchema$ *is a distinct complex type (can be set, set of sets etc), (c)* $DataSchema$ *is a relation type, (d)* $MeasureSchema$ *is a tuple of atomic types and (e)* $Formula$ *is a predicate expressed in the PSL language over the* $StructureSchema$ *and the* $DataSchema$*.*

Definition 5 *A* Pattern (Instance) p *over a Pattern Type PT is a quintuple [*PID*, Structure, Data,* $Measure$*,* $Formula$*] such that (a)* PID *is a unique identifier among all patterns of the same class, (b) Structure and* $Measure$ *are valid values of the respective schemata of* PT*, and (c) Data and* $formula$ *are expressions in the PSL language, properly instantiating the corresponding expressions of the pattern type* PT*.*

Example 1 *Let us now present an example of a pattern type* $Cluster$ *that defines a cyclical cluster and an example of one of its instance.*

Pattern Type $Cluster$

Name	*Cluster*
Structure Schema	$disk$:[$Center$: [X:$real$, Y:$real$], Rad:$real$]
Data Schema	rel:{[$A1$:$real$, $A2$:$real$]}
Measure Schema	$Precision$:$real$
Formula Schema	$(t.A1 - disk.Center.X)^2 + (t.A1 - disk.Center.Y)^2 \leq disk.Rad^2$ *where* $t \in rel$

Pattern Instance CustomerCluster

Pid	*337*
Structure	$disk$:[$Center$:[X:32, Y:90], Rad:12]
Data	$customer$:{[Age, $Income$]}
Measure	$Precision$: 0.91
Formula	$(t.Age-32)^2+(t.Income-90)^2 \leq 12^2$ *where* $t \in customer$

Intuitively we can see that the formula requires that all data that belong to the relation *customer* must satisfy the predicate $(t.Age - 32)^2 + (t.Income - 90)^2 \leq 12^2$. *Precision* in this case indicates that only 91% of them do.

In order to define Pattern Types correctly, we need to be able to define their *DataSchema* properly. Since a Pattern Type is a generic construct, not particularly bound to a specific data set, we employ a set of *auxiliary names*, which are employed in the definition of the *DataSchema* of Pattern Types for the specification of generic relations and attributes.

Having said that, the instantiation procedure that generates patterns on the basis of Pattern Types, is straightforward. Assume that a certain Pattern Type *PT* is instantiated in a new pattern p. Then:

- The domains involved in the *StructureSchema* and the *MeasureSchema* of *PT* are instantiated by valid values in p.
- The auxiliary relation and attribute names in the *DataSchema* of *PT* are replaced by regular relation and attribute names of an underlying database.
- Both the previous instantiations apply for the *FormulaSchema*, too: the attributes of the *StructureSchema* are instantiated to values and the auxiliary names of the *DataSchema* are replaced by regular names. All other variable names remain the same.

Having defined the data space and the pattern entities, we are ready to define the notions of *Pattern Class* and *Pattern Pase (PB)*. A Pattern Class over a pattern type is a collection of semantically related patterns, which are instances of this particular pattern type. Pattern classes play the role of pattern placeholders, just like relations do for tuples in the relational model.

Definition 6 *A* Pattern Class *is a triplet [Name, PT, Extension] such that (a) Name is a unique identifier among all classes, (b) PT is a pattern type and (c) Extension is a finite set of patterns with pattern type PT.*

Definition 7 *A* Pattern Base Schema *defined over a database schema* $\widehat{DD}$ *is defined as* $\widehat{PB} = \langle[\widehat{D_1}, \ldots, \widehat{D_n}], [\widehat{PC_1}:PT_1, \ldots, \widehat{PC_m}:PT_m]\rangle$, *where* PT_i*'s are pattern types involving the domains* $\widehat{D_1}, \ldots, \widehat{D_n}$ *and* $\widehat{PC_i}$*'s are pattern class names.*

Definition 8 *An instance of* $\widehat{PB}$*, i.e., a* pattern base*, over a database DB is a structure* $PB = \langle[PT_1, \ldots, PT_k], [PC_1, \ldots, PC_m]\rangle$*, where* PC_i*'s are pattern classes defined over pattern types* PT_i *with patterns whose data range over the data in DB.*

3.3. The Pattern Warehouse

Having defined the data and the pattern space, we are ready to introduce the global framework, in the context of which data- and pattern-bases coexist. To this end, we formally define the intermediate mappings between data and patterns and the overall context of patterns, data and their mappings.

Each pattern corresponds to a set of underlying data whom it represents. At the same time, each record in the source database corresponds to a (possibly empty) set of patterns that abstractly represent it. We assume a mapping Φ that relates patterns with their corresponding data. Through this mapping, we can capture both the relationship between a pattern and its corresponding data and at the same time, the relationship of a record with its corresponding patterns.

For reasons of simplicity, we avoid defining the relationship between data items and patterns at the level of individual relations and classes. Rather, we employ a generic representation, by introducing the union of all data items Δ and the union of all patterns Ω. Practically, the existence of *RID*'s and *PID*'s allows us to perform this union.

Definition 9 *The* active data space *of all data items of a database instance* $DB = \langle[D_1, \ldots, D_k], [R_1, \ldots, R_n]\rangle$*,* Δ_{DB}*, is the union of all relations, i.e.,* $\Delta_{DB} = R_1 \cup \cdots \cup R_n$*. The* active pattern space *of all patterns* Ω *of a pattern base instance PB is the union of all pattern classes, i.e.,* $\Omega_{PB} = PC_1 \cup \cdots \cup PC_n$*.*

Definition 10 *Given the active data- and pattern spaces* Δ_{DB} *and* Ω_{PB}*, an intermediate pattern-data*

mapping Φ *over* Δ_{DB} *and* Ω_{PB} *is a total function* $\Phi : \Delta_{DB} \times \Omega_{PB} \rightarrow \{true, false\}$*. We say that a data item* d is represented *by a pattern* p *and we write* $d \hookrightarrow p$ *or* $p \hookleftarrow d$ *iff* $\Phi(d, p) = true$*.*

It should be obvious now that *the formula of each pattern is an approximation of the mapping* Φ. In principle, it is an issue of implementation and mostly administration whether the intermediate mappings will be explicitly saved (with the storage and maintenance cost that this incurs) or simply approximated by the pattern formula (with the respective approximation error). In the sequel, we will demonstrate the usage of the formula as an approximation for the intermediate mappings.

Now, we are ready to define the notion of *Pattern Warehouse* which incorporates the underlying database (or source), the pattern bases and the intermediate mappings. Notice that although we separate patterns from data, we need the full environment in order to answer interesting queries, going all the way back to the data and to support interactive user sessions that navigate from the pattern to the data space and vice-versa.

Definition 11 *A* Pattern Warehouse Schema *is a pair* $\langle \widehat{DB}, \widehat{PB} \rangle$*, where* $\widehat{DB}$ *is a database schema and* $\widehat{PB}$ *is a pattern base schema.*

Definition 12 *A* Pattern Warehouse *is an instance of a pattern warehouse schema defined as a triplet:* $\langle DB, PB, \Phi \rangle$*, where* DB *is a database instance,* PB *is a pattern base instance, and* Φ *is an intermediate pattern-data mapping over* DB *and* PB*.*

4. Pattern Specification Language and Formula

The pattern formula describes the relation between the patterns, which are described in the structure field, and the raw data which are described in the source field. It is evident that we need a common language to describe all these fields. Therefore, in this section, we present the Pattern Specification Language (PSL), with particular focus on the following aspects: (a) language syntax, (b) the treatment of functions and predicates by PSL and (c) safety considerations.

Syntax. We chose as Pattern Specification Language the complex value calculus, presented in [5] by Abiteboul and Beeri. This calculus is a many-sorted calculus. The sorts are types as defined previously. Each constant and each variable is associated with a type and each function and predicate with a signature. The signature of a k-ary predicate is a k-tuple of types. The signature of a k-ary function is a $k+1$-tuple of types, involving the types of the parameters and the result of the function. The set of terms of the language is the smallest set that contains the atomic constants and variables, and it is closed under the application of functions. Simple formulae consist of predicates applied to terms and formulae are combinations of atomic formulae through the combination of the connectives $\wedge, \vee, \neg$, and the quantifiers $\forall, \exists$.

Functions and Predicates. Functions and predicates are quite important in the PBMS setting, since the approximation of the data to patterns mapping, usually needs complex functions to be expressed. Functions and predicates can possibly appear both in the formula field and in queries, associating relation names with the pattern structure. We believe that having *interpreted* functions is the best approach for the PBMS, since we would like the formula to be informative to the user and we would like to be able to reason on it.

Safety and Range Restriction. The formula is a predicate that we would ideally like to be true for all the data that are mapped to a pattern. Notice that the formula by itself does not contain a logical expression involving the pattern structure schema and the data schema, i.e., it is not a query on the relations of the raw data. The formula is merely a *predicate* to be used in queries. We would like for example to use it in queries that navigate between the data and the pattern space like the following:

$$\{x \mid fp(x) \wedge x \in R\}$$

where fp is a formula predicate and R is a relation appearing in the *Data* component. We require that fp is defined in such a way that we can construct queries like the previous, which are "safe". Safety is considered in terms of *domain independence*. Still, we cannot adopt the classical notion of domain independence (which restricts values to the active domain of the database), since even the simple functions can create new values (not belonging to the domain of the database). Therefore, we should consider a broader sense of domain independence similar to the one presented in [5, 17, 8], which allows the finite application of functions. For example, the n-depth domain independence as suggested in [5] considers domain independence with respect to the active domain closed under n application of functions. This includes the active domain and all the values that can be produced by applying the database functions n times, where n some finite integer.

The easiest way to ensure safety in these terms is to *range restrict* all variables appearing in a query. To this end, we introduce the *where* keyword in the *formula*, which facilitates the mapping of the formula predicate free variables to the relation schema that appears in the *DataSchema* or *Data* component. More specifi-

cally, we require that *there are no free variables in the fp that are not mapped to the relation of the Data component by the use of the where keyword.* This restriction, guarantees that all the variables appearing in fp are either range restricted or that the system knows how to range restrict them to a finite set of values when fp is used in a query.

Now we can formally define the well-formed *formula* for the pattern-type:

Definition 13 *A pattern type formula is of the form:*

$$fp(\overline{dv}, \overline{pv}), where\ \overline{dv} \in ds \quad (1)$$

where fp (formula predicate) is a PSL predicate, $\overline{dv}$ are variable names mapped by the where *keyword to the relation in DataSchema and $\overline{pv}$ are variable names that appear in the StructureSchema.*

At instantiation time $\overline{pv}$ is assigned values of the *Structure* component and $\overline{dv}$ is mapped to the relation appearing in *Data* component. The definition for the pattern well-formed *formula* is now straightforward:

Definition 14 *A pattern formula is of the form:*

$$fp(\overline{dv}), where\ \overline{dv} \in ds \quad (2)$$

where fp (formula predicate) is a PSL predicate, $\overline{dv}$ are variables mapped by the where keyword to the relation appearing in Data component.

From the previous definitions the semantics of the *where* keyword become evident: we impose that the variables of the formula will take values from specific relations when the formula predicate is employed in queries.

Example 2 *Let us consider the following formulas.*

1. $f(x)$ *where* $x \in R(x)$
2. $f(g(x), y)$ *where* $x \in R(x)$

In the first formula variable x is mapped to R using the where *keyword, thus the formula is well formed. Keep in mind that the formula predicate by itself is just the part $f(x)$, which is not range restricted. The second formula is not well-formed since y is not mapped via* where *to any relation, or otherwise range restricted.*

5. Querying the Pattern Warehouse

We define queries to be posed over the pattern warehouse and not individually over its data- or pattern-base components. Through this approach, we are able to sustain queries traversing from the pattern to the data space and vice-versa. At the same time, the consistency of the results is guaranteed by the pattern-data mapping Φ.

Definition 15 *Let $\mathcal{PW}$ the set of all possible Pattern Warehouses. A query is a function with signature $\mathcal{PW} \rightarrow \mathcal{PW}$. Given a query q and a pattern warehouse $pw = (DB, PB, \Phi)$, with $\widehat{pw} = (\widehat{DB}, \widehat{PB})$, $q(pw) = (DB', PB', \Phi')$, $\widehat{DB'} = \langle [\widehat{D}_1, ...\widehat{D}_k], [\widehat{R}_1{:}T_1] \rangle$, $\widehat{PB'} = \langle [\widehat{D}_1, ..., \widehat{D}_m], [\widehat{PC}_1{:}PT_1] \rangle$. We assume that $\forall t_r, t_p (t_r \in R_1 \wedge t_p \in PC_1) \Rightarrow \Phi'(t_r, t_p)$.*

Note that, similarly to the relational case, the result of a query is always a pattern warehouse containing just one relation and one pattern class. It is also important to point out that, in practice, even if a query always involve both the data and pattern space, operations over patterns are executed in isolation, locally at the PBMS. The reference to the underlying data is activated only on-demand (whenever the user specifically requests so) and efficiently enabled through the stored intermediate mappings or the formula approximation.

5.1. Query operators

In this section we introduce query operators that allow basic queries over the the PW. Assuming that $\mathcal{DB}$ denotes the set of all possible database instances and $\mathcal{PB}$ the set of all possible pattern bases, we consider the following groups of operators:

- *Database operators*: they can be applied locally to the DBMS. $op : \mathcal{DB} \rightarrow \mathcal{DB}$. We denote the set of database operators with $\mathcal{O}_{\mathcal{D}}$.
- *Pattern base operators*: they can be applied locally to the PBMS. $op : \mathcal{PB} \rightarrow \mathcal{PB}$. We denote the set of database operators with $\mathcal{O}_{\mathcal{P}}$.
- *Cross-over database operators*: they involve evaluation on both the DBMS and the PBMS, the result is a database. $op : \mathcal{DB} \times \mathcal{PB} \rightarrow \mathcal{DB}$. We denote the set of database operators with $\mathcal{O}_{\mathcal{CD}}$.
- *Cross-over pattern base operators*: they involve evaluation on both the DBMS and the PBMS, the result is a pattern base. $op : \mathcal{DB} \times \mathcal{PB} \rightarrow \mathcal{PB}$. We denote the set of database operators with $\mathcal{O}_{\mathcal{CP}}$.

In the following, we present examples of the last three classes of operators (database operators coincide with usual relational operators). Before presenting such operators, we introduce some examples of predicates defined over patterns.

5.1.1. Pattern predicates We identify two main classes of atomic predicates: predicates over patterns and predicates over pattern components. From those atomic predicates we can then construct complex predicates. In the following, we denote pattern components by using the dot notation. For example, the measure component of a pattern p is denoted by $p.Measure$.

Predicates over pattern components. They check properties of specific pattern components. Let p_1 and p_2 be two patterns, possibly selected by some queries. The general form of a predicate over pattern components is $t_1 \theta t_2$, where t_1 and t_2 are path expressions that must define components of patterns p_1 and p_2, of *compatible* type and θ must be an operator, defined for the type of t_1 and t_2. For example, if t_1 and t_2 are integer expressions, then θ can be a disequality operator (e.g. one of $<, >$). We consider the following special cases:

- If t_1 and t_2 are pattern data for patterns p_1 and p_2, then $\theta \in \{=, \subseteq\}$. $t_1 = t_2$ is true if and only if $\forall x\ x \hookrightarrow p_1 \Leftrightarrow x \hookrightarrow p_2$ and $t_1 \subseteq t_2$ is true if and only if $\forall x\ x \hookrightarrow p_1 \Rightarrow x \hookrightarrow p_2$.
- If t_1 and t_2 are pattern formulas for patterns p_1 and p_2, then $\theta \in \{=, \preceq\}$. $t_1 = t_2$ is true if and only if $t_1 \equiv t_2$ and $t_1 \preceq t_2$ is true if and only if t_1 logically implies t_2.

Predicates over patterns. We consider the following set of predicates:

- *Identity* ($=$). Two patterns p_1 and p_2 are identical if they have the same PID, i.e. $p_1.PID = p_2.PID$.
- *Shallow equality* ($=^s$). Two patterns p_1 and p_2 are shallow equal if their corresponding components (except for the PID component) are equal, i.e. $p_1.Structure = p_2.Structure$, $p_1.Source = p_2.Source$, $p_1.Measure = p_2.Measure$, and $p_1.formula = p_2.formula$. Note that, to check the equality for each component pair, the basic equality operator for the specific component type is used.
- *Deep equality* ($=^d$). Two patterns p_1 and p_2 are deep equal if their corresponding data are identical, i.e., $\forall x\ x \hookrightarrow p_1 \Leftrightarrow x \hookrightarrow p_2$.
- *Subsumption* ($\preceq$). A pattern p_1 subsumes a pattern p_2 ($p_1 \preceq p_2$) if they have the same structure but p_2 represents a smaller set of raw data, i.e. $p_1.Structure = p_2.Structure$, $p_1.Source \subseteq p_2.Source$ and $p_1.formula \preceq p_2.formula$.

Complex predicates. They are defined by applying usual logical connectives to atomic predicates. Thus, if F_1 and F_2 are predicates, then $F_1 \wedge F_2$,$F_1 \vee F_2$,$\neg F_1$ are predicates. We make a *closed world* assumption, thus the calculation of $\neg F$ is always finite.

5.1.2. Pattern base operators $\mathcal{O}_{\mathcal{P}}$ In this subsection, we introduce several operators defined over patterns. Some of them, like set-based operators, renaming and selection are quite close to their relational counterparts; nevertheless, some others, like join and projection have significant differences.

Set-based operators. Since classes are sets, usual operators such as union, difference and intersection are defined for pairs of classes of the same pattern type.

Renaming. Similarly to the relational context, we consider a renaming operator ρ that takes a class and a renaming function and changes the names of the pattern attributes according to the specified function.

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 [12]. 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 of pattern type pt. Let ls be a non empty list of attributes appearing in $pt.Structure$ and lm a list of attributes appearing in $pt.Measure$. Then, the projection operator is defined as follows:

$$\pi_{(ls,lm)}(c) = \{(id(), \pi^s_{ls}(s), d, \pi^m_{lm}(m), \pi_{ls \cup lm}(f)) \mid \exists p \in c, p = (pid, s, d, m, f)\}$$

In the previous definition, $id()$ is a function returning new pids for patterns, $\pi^m_{lm}(m)$ is the usual relational projection of the measure component and $\pi^s_{ls}(s)$ is defined as follows: (i) if s is a tuple type, then $\pi^s_{ls}(s)$ is the usual relational projection; (ii) if s is a set type, then $\pi^s_{ls}(s)$ is obtained by keeping the projected components and removing the rest from set elements. The last component $\pi_{ls \cup lm}(f)$ is the new formula. This can only be computed in certain cases, when the theory over which the formula is constructed admits projection. This happens for example for the polynomial constraint theory [12].

Selection. The selection operator allows one to select the patterns belonging to one class that satisfy a certain predicate, involving any possible pattern component, chosen among the ones presented in Section 5.1.1. Let c be a class of pattern type pt. Let pr be a predicate. Then, the selection operator is defined as follows:

$$\sigma_{pr}(c) = \{p | p \in c \wedge pr(p) = true\}$$

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

Let c_1 and c_2 be two classes over two pattern types pt_1 and pt_2. A join predicate F is any predicate 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 $c = (c_{StructureSchema}, c_{DataSchema}, c_{MeasureSchema}, c_{Formula})$, one for each pattern component. For example, function $c_{StructureSchema}$ takes as input two structure values of the right type and returns 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:
$Structure : c_{StructureSchema}(s1, s2)$
$Data : c_{DataSchema}(d1, d2)$
$Measure : c_{MeasureSchema}(m1, m2)$
$Formula : c_{formula}(f1, f2)$.

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

5.1.3. Cross-over database operators $\mathcal{O}_{\mathcal{CD}}$

Drill-Through. The drill-through operator allows one to navigate from the pattern layer to the raw data layer. Thus it takes as input a pattern class and it returns a raw data set. More formally, let c be a class of pattern type pt and let d be an instance of the data schema ds of pt. Then, the drill-through operator is denoted by $\gamma(c)$ and it is formally defined as follows:

$$\gamma(c) = \{d | \exists p, p \in c \ \wedge \ d \hookrightarrow p\}$$

Data-covering. Given a pattern p and a dataset D, sometimes it is important to determine whether the pattern represents it or not. In other words, we wish to determine the subset S of D represented by p (p can also be selected by some query). To determine S, we use the formula as a query on the dataset. Let p be a pattern, possibly selected by using query language operators, and D a dataset with schema $(a_1, ..., a_n)$, compatible with the source schema of p. The data-covering operator, denoted by $\theta_d(p, D)$, returns a new dataset corresponding to all tuples in D represented by p. More formally,

$$\theta_d(p, D) = \{t \mid t \in D, p.formula(t.a_1, ..., t.a_n) = true\}$$

In the previous expression, $t.a_i$ denotes a specific component of tuple t belonging to D and $p.formula(t.a_1, ..., t.a_n)$ is the formula predicate of p instantiated by replacing each variable corresponding to a pattern data component with values of the considered tuple t.

Note that, since the drill-though operator uses the intermediate mapping and the data covering operator uses the formula, the covering $\theta(p, D)$ of the data set $D = \gamma(p)$ returned by the drill through operator, might not be equal to D. This is due to the approximating nature of the pattern formula.

5.1.4. Cross-over pattern base operators $\mathcal{O}_{\mathcal{CP}}$

Pattern-covering. Sometimes it can be useful to have an operator that, given a class of patterns and a dataset, returns all patterns in the class representing that dataset (a sort of inverse data-covering operation). Let c be a pattern class and D a dataset with schema $(a_1, ..., a_n)$, compatible with the source schema of the c pattern type. The pattern-covering operator, denoted as $\theta_p(c, D)$, returns a set of patterns corresponding to all patterns in c representing D. More formally:

$$\theta_p(c, D) = \{p \mid p \in c, \forall t \in D \; p.formula(t.a_1, ..., t.a_n) = true\}$$

Note that: $\theta_p(c, D) = \{p | p \in c, \theta_d(p, D) = D\}$

6. Related Work

Although significant effort has been invested in extending database models to deal with patterns, no coherent approach has been proposed and convincingly implemented for a generic model.

There exist several standardization efforts for modeling patterns, like the Predictive Model Markup Language (PMML) [4], which is an XML-based modeling approach, the ISO SQL/MM standard [2], which is SQL-based, and the Common Warehouse Model (CWM) framework [1], which is a more generic modeling effort. Also, the Java Data Mining API (JDMAPI) [3] addresses the need for a language-based management of patterns. Although these approaches try to represent a wide range of data mining result, the theoretical background of these frameworks is not clear. Most importantly, though, they do not provide a generic model capable of handling arbitrary cases of pattern types; on the contrary, only a given list of predefined pattern types is supported.

To our knowledge, research has not dealt with the issue of pattern management per se, but, at best, with peripheral proximate problems. For example, the paper by Ganti et. al. [9] deals with the measurement of similarity (or deviation, in the authors' vocabulary) between decision trees, frequent itemsets and clusters. Although this is already a powerful approach, it is not generic enough for our purpose. The most relevant research effort in the literature, concerning pattern management is found in the field of inductive databases,

meant as databases that, in addition to data, also contain patterns [10], [7]. Our approach differs from the inductive database one mainly in two ways. Firstly, while only association rules and string patterns are usually considered there and no attempt is made towards a general pattern model, in our approach no predefined pattern types are considered and the main focus lies in devising a general and extensible model for patterns. Secondly, differently from [10], we claim that the peculiarities of patterns in terms of structure and behavior, together with the characteristic of the expected workload on them, call for a logical separation between the database and the pattern-base in order to ensure efficient handling of both raw data and patterns through dedicated management systems.

Finally, we remark that even if some languages have been proposed for pattern generation and retrieval [14, 11], they mainly deal with specific types of patterns (in general, association rules) and do not consider the more general problem of defining safe and sufficiently expressive language for querying heterogeneous patterns.

7. Conclusions and Future Work

In this paper we have dealt with the issue of modelling and managing patterns in a database-like setting. Our approach is enabled through a Pattern-Base Management System, enabling the storage, querying and management of interesting abstractions of data which we call patterns. In this paper, we have (a) formally defined the logical foundations for the global setting of PBMS management through a model that covers data, patterns and intermediate mappings and (b) discussed language issues for PBMS management. To this end we presented a pattern specification language for pattern management along with safety constraints for its usage and introduced queries and query operators and identified interesting query classes.

Several research issues remain open. First, it is an interesting topic to incorporate the notion of type and class hierarchies in the model [15]. Second, we have intentionally avoided a deep discussion of statistical measures in this paper: it is more than a trivial task to define a generic ontology of statistical measures for any kind of patterns out of the various methodologies that exist (general probabilities, Dempster-Schafer, Bayesian Networks, etc. [16]). Finally, pattern-base management is not a mature technology: as a recent survey shows [6], it is quite cumbersome to leverage their functionality through object-relational technology and therefore, their design and engineering is an interesting topic of research.

References

[1] Common Warehouse Metamodel (CWM). http://www.omg.org/cwm, 2001.

[2] ISO SQL/MM Part 6. http://www.sql-99.org/SC32/WG4/Progression_Documents/FCD/fcd-datamining-2001-05.pdf, 2001.

[3] Java Data Mining API. http://www.jcp.org/jsr/detail/73.prt, 2003.

[4] Predictive Model Markup Language (PMML). http://www.dmg.org/pmmlspecs_v2/pmml_v2_0.html, 2003.

[5] S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4(4):727–794, 1995.

[6] B. Catania, A. Maddalena, E. Bertino, I. Duci, and Y. Theodoridis. Towards a benchmark for pattern bases. http://dke.cti.gr/panda/index.htm, 2003.

[7] L. De Raedt. A perspective on inductive databases. *SIGKDD Explorations*, 4(2):69–77, 2002.

[8] M. Escobar-Molano, R. Hull, and D. Jacobs. Safety and translation of calculus queries with scalar functions. In *Proceedings of PODS*, pages 253–264. ACM Press, 1993.

[9] V. Ganti, R. Ramakrishnan, J. Gehrke, and W.-Y. Loh. A framework for measuring distances in data characteristics. *PODS*, 1999.

[10] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58–64, 1996.

[11] T. Imielinski and A. Virmani. MSQL: A Query Language for Database Mining. *Data Mining and Knowledge Discovery*, 2(4):373–408, 1999.

[12] P. Kanellakis, G. Kuper, and P. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, 51(1):25–52, 1995.

[13] P. Lyman and H. R. Varian. How much information. http://www.sims.berkeley.edu/how-much-info, 2000.

[14] R. Meo, G. Psaila, and S. Ceri. An Extension to SQL for Mining Association Rules. *Data Mining and Knowledge DiscoveryM*, 2(2):195–224, 1999.

[15] S. Rizzi, E. Bertino, B. Catania, M. Golfarelli, M. Halkidi, M. Terrovitis, P. Vassiliadis, M. Vazirgiannis, and E. Vrachnos. Towards a logical model for patterns. In *Proceedings of ER 2003*, 2003.

[16] A. Siblerschatz and A. Tuzhillin. What makes patterns interesting in knowledge discovery systems. *IEEE TKDE*, 8(6):970–974, 1996.

[17] D. Suciu. Domain-independent queries on databases with external functions. In *Proceedings ICDT*, volume 893, pages 177–190, 1995.

[18] M. Terrovitis, P. Vassiliadis, S. Skadopoulos, E. Bertino, B. Catania, and A. Maddalena. Modeling and language support for the management of pattern-bases. Technical Report TR-2004-2, National Technical University of Athens, 2004. Available at `http://www.dblab.ece.ntua.gr/pubs`.