Modeling and Language Support for the Management of Pattern-Bases

Manolis Terrovitis[1] Panos Vassiliadis[2] Spiros Skiadopoulos[1] Elisa Bertino[3] Barbara Catania[4] Anna Maddalena[4]

***Abstract*— Information overloading is today a serious concern that may hinder the potential of modern web-based information systems. A promising approach to deal with such problem is represented by knowledge extraction methods 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.**

I. Introduction

Nowadays, we are experiencing a phenomenon of information overload, which escalates beyond any of our traditional beliefs. As a recent survey states [1], 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. Clearly, this sheer volume of collected data in digital form calls for novel information extraction, management and querying techniques, thus posing the need for novel Database Management Systems (DBMSs). Still, even novel DBMS architectures are insufficient to cover the gap between the exponential growth of data and the slow growth of our understanding [2], due to our methodological bottlenecks and simple human limitations. To compensate for these shortcomings, 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 citizens. 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 [3], [4], 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 usage and semantics.

The rest of this paper is organized as follows. In Section II we introduce the notion of Pattern-Base Management system. Section III presents a generic model for the coexistence of data and patterns. In Section IV we present the Pattern Specification Language, that enables us to specify pattern semantics. In Section V we explain how patterns can be queried and introduce query classes and operators. Finally, in Section VI we present related work. Section VII offers conclusions and topics of future work.

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

To effectively describe what patterns are in our context and why they are so useful, we present an exemplary scenario, summarized in Fig. 1. On the left hand side of Fig. 1, one can observe the transactions made in a particular store. An algorithm for the extraction of association rules of the form $body \Rightarrow head$ has been applied over these data and the results are depicted in the right-hand side of Fig. 1. We choose a 2D representation of the association rules, with each axis

This work was supported by the PANDA IST Thematic Network.

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

[2] Dept. of Computer Science, Univ. of Ioannina, Ioannina, Hellas, pvassil@cs.uoi.gr

[3] DICO, Univ. of Milan, Italy, bertino@dico.unimi.it

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

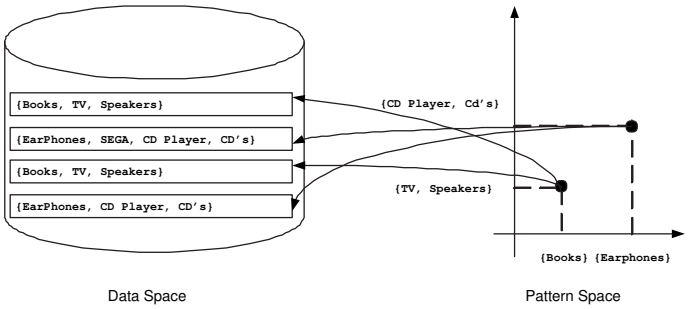

Fig. 1. Patterns (association rules) and their mapping to raw data

representing the head or the body of the rule and the rules being points in the 2-dimensional space (e.g., the association rule $\{TV, Speakers\} \Rightarrow Books$ is defined by the respective points in the $Body$ and $Head$ axes).

Thus, 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. Based on this observation, we can base the discussion on the following assumptions:

- There exists a *data space* (the space of raw data) and a *pattern space*.
- There always exist *relationships* among the members of the data space and the members of the pattern space. In general, these relationships can be of cardinality *many-to-many*, i.e., a pattern can correspond to more than one data item and vice versa.

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 [5];
- 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. 2 and consists of three major layers of information organization. In the bottom of Fig. 2, 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. 2, we depict the PBMS repository that corresponds to the *pattern space* and contains patterns. The PBMS repository is managed by the Pattern-Base Management System. Finally, in the middle of

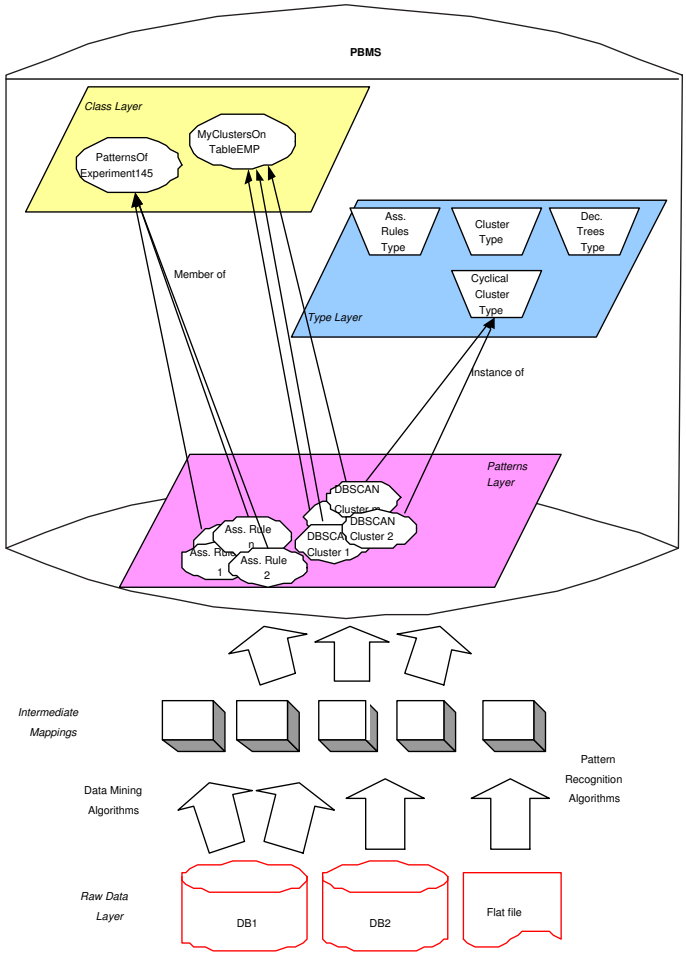

Fig. 2. Reference architecture for the Pattern-Base Management System

Fig. 2, 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; for example, they could be used to retrieve the rows that produced the association rule of Fig. 1. 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. *To provide a solution towards this problem, we approximate the relationship between the pattern*

and the data space through a specific language, the Pattern Specification Language (PSL), which will be introduced in the sequel. PSL is applied both as a language for describing pattern types and instances and as a means to reason over the properties of the described patterns.

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. 2, two such algorithms have been applied: an algorithm for the extraction of association rules and the DBSCAN algorithm [6] for the extraction of 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. Normally, we anticipate the PBMS to come with a set of built-in, popular pattern types (e.g., association rules, clusters, see also Fig. 2). Still, the type layer must be extensible, simply because the set of pattern types that it incorporates must be extensible.

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. 2.

III. 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 will quickly introduce the notions of data types, attributes and relations (in the usual relational sense). We will exploit the definitions already given in [7] for this purpose. Then, we will proceed to define formally pattern types, pattern classes, patterns as well as the intermediate mappings. Then, we will define pattern bases and pattern warehouses.

A. The Data Space

In this section, we will deal with the formal definition of the entities of the *data space* (Fig. 1), i.e., data types, relations and databases.

Practically, we start with the data model proposed by Abiteboul and Beeri in [7], 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 [7]:

- If 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 predicates 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.

Example 1: Let us consider the following types.

$$T_1 = [X{:}real,\ Y{:}real,\ Z{:}integer]$$
$$T_2 = \{\ [Id{:}integer,\ Disk{:}\{\ [Center{:}[X{:}real, Y{:}real],\ Rad{:}real]\ \}\ ]\ \}$$

The expressions

$$[X{:}4.1,\ Y{:}5.2,\ Z{:}3]$$
$$\{\ [Id{:}7,\ Disk{:}\{\ [Center{:}[X{:}2, Y{:}3], Rad{:}4]\ \}]\ \}$$

are values of types T_1 and T_2 respectively.

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 [7] which require 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 ([7]): 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 ([7]): 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*.

Example 2: The following expressions are valid relation schemata:

$$\widehat{R_1}{:}\{[A{:}\{\}, B{:}[], C{:}\{[]\}]\} \text{ and } \widehat{R_2}{:}\{[A, B, C]\}$$

whereas the following expressions are not valid:

$$\widehat{R_3}:\{A:\{[]\}\} \text{ and } \widehat{R_4}:[A, B, C].$$

Expression $\widehat{R_3}$ is invalid as a set of *sets* instead of a set of *tuples*. $\widehat{R_4}$ on the other hand, is invalid as a tuple (instead of a *set* of tuples). Note also that in all our examples we have omitted domains for brevity. Using the full definition is easy; for example, $\widehat{R_2}$ could be: $\widehat{R_2}:\{[A:string, B:string, C:real]\}$

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.

B. 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 [3] are compact, yet rich in semantics, representation 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 IV 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 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 3: Let us now present an example of a pattern type $Cluster$ that defines a circular 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)*. Our final goal is to introduce the global framework, called *Pattern Warehouse*, as a unified environment in the context of which data- and pattern-bases coexist.

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.

Let us now define pattern bases, which are practically finite collections of pattern classes, defined over a set of pattern types and containing pattern instances.

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.

C. 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 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. Naturally, based on Φ we can compute the mappings (a) Φ_{pd}, mapping patterns to their corresponding data and (b) Φ_{dp}, giving all the patterns for each record in the database. Notice that since the relationship is *many-to-many*, in the general case, Φ is a mapping and not a function.

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.

IV. 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 requirements, (b) language syntax, (c) the treatment of functions and predicates by PSL and (d) safety considerations.

Requirements. Given the complexity of the relation that may exist between the data and the patterns, as well as the various complicated structures of data we may face, several requirements arise for the pattern specification language:

- The easy and safe handling of (new) *functions* and *predicates*, in order to describe complex relations between patterns and data.
- Queries must be able to *reason* for patterns and data based on the formula. This means that the formula must be expressed in a generic language that facilitates reasoning methods.
- The formula must be *informative* to the user, i.e., the user must be able to intuitively understand the abstraction provided by the pattern through simple inspection.

Syntax. Considering the previous requirements, we chose as Pattern Specification Language the complex value calculus, presented in [7] 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 the signature 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 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. Fixed semantics thus, become necessary if everyone should be able to understand what a function name stands for.

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 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 [7], [8], [9], which allows the finite application of functions. For example, the n-depth domain independence as suggested in [7] considers domain independence with respect to the active domain closed under n application of functions. This means that 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 specifically, 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:

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

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

At instantiation time pv is assigned values of the $Structure$ component and 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:

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

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 4: 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.

V. Querying the Pattern Warehouse

As already mentioned before, in our approach, data and patterns are distinct entities, stored in different ways (data-

vs. pattern-bases). This is justified mainly for performance, maintenance and administration issues (e.g., we anticipate that the PBMS will employ different data structures and query processing for the management of patterns; the back-stage synchronization of patterns and evolving data is also more efficient in a multi-tier, load balanced architecture). Nevertheless, all these do not imply that the querying process is restricted simply to patterns or data. More precisely, 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.

An important point where the PW queries differ from the relational queries is the existence of the formula component and the reasoning options that it offers. Notice that we do not consider the extraction of new patterns from the data itself, i.e., the creation of a formula from the system, given only the data. This work cannot be done automatically, without some semantic knowledge from the user, and it cannot be done with simple queries.

A. Classification of Queries

In the sequel, we present a fundamental classification of the queries. The classification is based on the type of the results generated by the query. Indeed, even if queries are defined over pattern warehouses, they may deal either with data, patterns or both. In the following, $\emptyset_{PB}$ represents an empty pattern base and $\emptyset_{DB}$ an empty database.

1) Q1: Queries whose results are data: Queries of this type always return a pattern warehouse with an empty pattern base, i.e., $q(\langle DB, PB, \Phi\rangle) = \langle DB', \emptyset_{PB}, -\rangle$, where $-$ represents no function (since the domain is empty). These queries may ask for data that comply to some expression, or to a combination of pattern formulas and other restrictions given in the query. The queries of this type are thus very close to the queries we ask in a DBMS although here we use the pattern formulas, i.e, both the database and the pattern base of the input warehouse must be accessed to answer the query.

For example, suppose you want to determine data belonging to two different classes (thus, data classified in two different ways) obtained as result of a classification process and represented by two different patterns with formula f_1 and f_2 respectively. In this case, the query condition corresponds to the conjunction between f_1 and f_2.

2) Q2: Queries that have as results patterns: Queries of this type always return a pattern warehouse containing patterns of the input warehouse that satisfy some conditions or new patterns created from existing ones. In all cases, the database of the output pattern warehouse is empty, i.e., $q(\langle DB, PB, \Phi\rangle) = \langle \emptyset_{DB}, PB', -\rangle$.

Queries of the first type are very close to queries belonging to class *Q1*, i.e., they select patterns already contained in the input pattern base. For example, the query selecting patterns satisfying condition $x \in PC \wedge \phi(x)$ is an example of query belonging to $Q2$, returning all patterns belonging to class PC and satisfying ϕ. Queries of this type do not require the usage of the input database for their computation.

The second type of queries are more interesting and support the generation of new patterns (of possibly new pattern types) from the ones contained in the input warehouse. To these patterns, new PIDs are assigned automatically and a (re)-computation of the measures is required. Note that, differently from the first type of queries, in this case, due to the measure (re)-computation, both the input database and pattern base are used to answer this type of queries.

One of the main important questions that arises in this case is how the formula of the new patterns is created. In general, new formulas can be created by applying well defined operators (for example conjunction) to the formulas of some input patterns. Of course, in this case, the data source of the output pattern has to be changed accordingly from the data source of the input ones.

3) Q3: Queries that have as a result both patterns and data: These queries are combinations of the queries of the previous categories. They return a pattern warehouse where both the database and pattern base may not be empty. For the computation of this kind of queries, both the input database and pattern base are required. For example, a query of this type, might involve asking the system to display the pattern that had the most data mapped to it, as well as the data themselves, ranging over the patterns of the intersection of two pattern classes (in order to find the most important patterns – practically the ones generated by two different data mining algorithms).

B. Query operators

The classification of the queries implies the existence of some basic operators that provide data/pattern base manipulations. In particular, queries belonging to each class rely on different types of operators, depending on the result we want to obtain. More precisely, we consider the following groups of operators:[1]

- *Database operators*: they can be applied locally to the DBMS. $op : \mathcal{DB} \rightarrow \mathcal{DB}$. We denote the set of database operators with $\mathcal{O}_{\mathcal{D}}$.

[1]In the following we denote with $\mathcal{DB}$ the set of all possible database instances and with $\mathcal{PB}$ the set of all possible pattern bases.

- *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 $\mathcal{O}_{\mathcal{CP}}$.

Cross-over operators correlate patterns with raw data, providing a way for navigating from the pattern layer to the raw data layer and vice versa. Their execution requires querying the raw data repository, thus they can be considered as *expensive* operations.

Queries presented in Section V-A rely on the previous operators. In particular, for queries of type $Q1$, the new database can be generated by using some database or cross-over database operators. On the other hand, for queries of type $Q2$, the new pattern base can be generated by using pattern base or cross-over pattern base operators. Queries of type $Q3$ can use all the considered operators to generate the new database and/or the new pattern base.

Note that, similarly to the relational case, operators manipulate relations and/or classes. Relation schemas, pattern types, functions and predicate sets can be implicitly computed from them. Thus, in the following, according to the relational case, operators are defined over relations and classes, and the corresponding data/pattern bases are implicitly computed.

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.

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.

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 [10]. 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:

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

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 $\pi_{ls \cup lm}(e)$ 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 [10].

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 V-B.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:

$c_1 \bowtie_{F,c} c_2 =$

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

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 pairs of patterns having the same data source, which is also assigned to the output pattern. The natural join is defined as $c_1 \bowtie_{F,c} c_2$, where F is the predicate requiring the equality for attributes with the same name and type in the structure components and $c()$ is the following composition function: $c = (c_{<,>}, \downarrow_d, \cup_{null}, \wedge)$, where:

- $c_{<,>}$ returns a record with two components, one for each input structure value;
- $\downarrow_d$ is a function that takes two patterns with the same data source and returns it;
- $\cup_{null}$ 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);
- $\wedge$ is the logical conjunction.

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,

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

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 true. This is due to the approximating nature of the pattern formula.

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:

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

Note that:

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

VI. 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) [11], which is an XML-based modeling approach, the ISO SQL/MM standard [12], which is SQL-based, and the Common Warehouse Model (CWM) framework [13], which is a more generic modeling effort. Also, the Java Data Mining API (JDMAPI) [14] 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. [15] 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 [16], [17]. 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 [16], 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 [18], [19], 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.

VII. 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 [3]. 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. [20]). Finally, pattern-base management is not a mature technology: as a recent survey shows [21], 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] P. Lyman and H. R. Varian, "How much information," http://www.sims.berkeley.edu/how-much-info, 2000.

[2] J. Gray, "The information avalanche: Reducing information overload," http://research.microsoft.com/ Gray/Talks/, 2002.

[3] 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.

[4] S. Rizzi, E. Bertino, B. Catania, and M. Golfarelli, "A logical framework for pattern representation," in *Proceedings of the PANDA Workshop on Pattern-Base Management Systems*, 2003, pp. 31–38.

[5] J. Widom, "Research problems in data warehousing," in *Proceedings of CIKM '95, Baltimore, Maryland, USA*. ACM, 1995, pp. 25–30.

[6] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 1996, pp. 226–231.

[7] S. Abiteboul and C. Beeri, "The power of languages for the manipulation of complex values," *VLDB Journal: Very Large Data Bases*, vol. 4, no. 4, pp. 727–794, 1995. [Online]. Available: citeseer.nj.nec.com/abiteboul95power.html

[8] D. Suciu, "Domain-independent queries on databases with external functions," in *Proceedings of Database Theory - ICDT'95,Prague, Czech Republic, January 11-13, 1995*, ser. Lecture Notes in Computer Science, G. Gottlob and M. Y. Vardi, Eds., vol. 893. Springer, 1995, pp. 177–190.

[9] M. Escobar-Molano, R. Hull, and D. Jacobs, "Safety and translation of calculus queries with scalar functions," in *Proceedings of PODS'93, May 25-28, 1993,Washington,DC*. ACM Press, 1993, pp. 253–264.

[10] P. Kanellakis, G. Kuper, and P. Revesz, "Constraint Query Languages," *Journal of Computer and System Sciences*, vol. 51, no. 1, pp. 25–52, 1995.

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

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

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

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

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

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

[17] L. De Raedt, "A perspective on inductive databases," *SIGKDD Explorations*, vol. 4(2), pp. 69–77, 2002.

[18] R. Meo, G. Psaila, and S. Ceri, "An Extension to SQL for Mining Association Rules," *Data Mining and Knowledge DiscoveryM*, vol. 2, no. 2, pp. 195–224, 1999.

[19] T. Imielinski and A. Virmani, "MSQL: A Query Language for Database Mining," *Data Mining and Knowledge DiscoveryM*, vol. 2, no. 4, pp. 373–408, 1999.

[20] A. Siblerschatz and A. Tuzhillin, "What makes patterns interesting in knowledge discovery systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 6, pp. 970–974, 1996.

[21] 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.