Dipartimento di Informatica e
Scienze dell'Informazione

A Unified Framework for Heterogeneous Pattern Management

by

Anna Maddalena

Theses Series **DISI-TH-2006-04**

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e
Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

A Unified Framework for Heterogeneous Pattern Management

by

Anna Maddalena

April, 2006

Dottorato di Ricerca in Scienze e Tecnologie dell'Informazione e della Comunicazione
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science (S.S.D. INF/01)

Submitted by Anna Maddalena
DISI, Univ. di Genova
maddalena@disi.unige.it

Date of submission: March 2006

Title: Pattern Based Management System: Data Models and Architectural Aspects

Advisor: Barbara Catania
DISI, Univ. degli Studi di Genova
catania@disi.unige.it

Ext. Reviewers: Alberto Belussi
Dipartimento di Informatica - Univ. degli Studi di Verona
alberto.belussi@univr.it

Stephane Bressan
National University of Singapore
steph@nus.edu.sg

Abstract

In the last few years, a huge amount of electronic data has become available in many different applicational contexts, such as, for example, market-basket analysis, e-commerce, financial and stock exchange evaluation, semantic Web, image recognition. This vast volume of collected digital data does not constitute knowledge by itself. Knowledge extraction processes (such as, for example, pattern recognition and data mining techniques) and data management techniques are often required to extract from them concise and relevant information that can be interpreted, evaluated and manipulated by users in order to drive and specialize business decision processes. Hidden knowledge can be represented by several kinds of knowledge artifacts, usually called *patterns*. Market-basket association rules, clusters, clickstreams, and image features are only some examples of significant types of patterns. Despite their heterogeneity, each pattern represents a semantic characteristic of a certain, possible huge, raw dataset in a concise way.

The ability to manipulate different types of patterns under a unique environment is becoming a fundamental issue for any 'intelligent' and data-intensive application. Unfortunately, the specific characteristics of patterns make traditional DBMSs unsuitable for pattern representation and management. Indeed, patterns can be generated from different application contexts resulting in very heterogeneous structures, possibly sharing hierarchical relationships. Moreover, patterns can be generated by using some data mining tools but can also be user-defined. Moreover, they can be extracted (a-posteriori patterns) but also known in advance by the users and used for example to check how well some data source is represented by them (a-priori patterns). Finally, patterns should be manipulated (e.g. extracted, synchronized, and deleted) and queried through dedicated languages. In particular, synchronization issues are very important, since source data change with high frequency and the ability to maintain a semantic alignment between patterns and raw data from which they have been extracted constitute a key feature in supporting decision processes.

Several approaches have been proposed so far by both the scientific and industrial communities for pattern management. However, they usually deal with some predefined types of patterns and mainly concern pattern extraction and exchange issues. Most of them rely on a-posteriori data mining patterns and do not provide a direct support for user-defined and a-priori patterns. Additionally, pattern hierarchies are in general not taken into account. Usually, advanced capabilities for pattern extraction and management are provided

for each pattern type but integration of heterogeneous patterns is in general not directly supported.

What is missing is therefore a unified framework dealing with heterogeneous patterns in a homogeneous way. This PhD dissertation addresses this problem by proposing a general framework for pattern representation and management, supporting most of the pattern management capabilities introduced above.

After a detailed analysis of the existing proposals and the identification of pattern management requirements, as a first contribution, a formal model for pattern representation is presented. The proposed model supports advanced pattern modeling features, such as: user-defined patterns, validity aspects related to patterns, and pattern hierarchies.

Based on the proposed pattern model, ad-hoc pattern manipulation and query languages are proposed. Concerning the query language, both an algebraic and a calculus-based language are provided and proved to be equivalent. Differently from other existing proposals, they support the combination of possibly heterogeneous patterns together and with raw data. The manipulation language provides the management of both a-posteriori and a-priori patterns and various synchronization operations.

The presentation of the pattern model and languages is then completed by the investigation of several theoretical issues concerning the expressive power and complexity of the proposed framework, leading to the identification of conditions under which it becomes tractable while remaining sufficiently expressive.

Finally, in order to show how the proposed pattern management framework can be effectively used in advanced data-intensive applications, the dissertation is completed with the presentation of PSYCHO, a prototype system we have developed, implementing the proposed framework. PSYCHO allows the user to use standard pattern types or define new ones according to the proposed logical model and supports pattern generation, manipulation, and analysis under an integrated environment. The technologies used in developing the prototype rely on object-relational database technologies integrated with logical constraint solving features, through the usage of object-oriented programming.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The focus of this PhD Thesis is the development of a framework for the management of heterogeneous patterns. In the following, the problem of pattern management is informally introduced pointing out the notion of *pattern* considered in this thesis (Section 1.1). As we will see, the pattern management problem spreads in many different disciplines and branches of research and the unified management of heterogeneous patterns constitutes an open research problem, which is the object of this PhD work as detailed in Section 1.2. The organization of this PhD thesis is described in Section 1.3.

1.1 The Pattern Management Problem

The increasing opportunity of quickly collecting and cheaply storing large volumes of data, and the need of extracting concise information to be efficiently manipulated and intuitively analyzed, are posing new requirements for data management systems in both industrial and scientific applications.

Raw data are recorded from various sources in the real world, often by collecting measurements from various instruments or devices (e.g., cellular phones, environment measurements, monitoring of computer systems, etc.). The determining property of raw data is the vastness of their volume; moreover, a significant degree of heterogeneity may be present. Clearly, data in such huge volumes do not constitute knowledge *per se*, i.e. little useful information can be deduced simply by their observation, so they hardly can be directly exploited by human beings. Thus, more elaborate techniques are required in order to extract the hidden knowledge and make these data valuable for end-users.

An usual approach to deal with such huge quantity of data is to reduce the available data to knowledge artifacts (i.e., clusters, association rules, etc.) through data processing methods

(pattern recognition, data mining, knowledge extraction) that reduce their number and size to make them manageable from humans while preserving as much as possible their hidden, interesting information. This knowledge artifacts are also called *patterns*. Therefore, the term *pattern* is a good candidate to generally denote these novel information types, characterized by a high degree of diversity and complexity.

Definition 1 A pattern is a compact and rich in semantics representation of raw data.

Patterns can be regarded as knowledge units that effectively describe entire subsets of data (in this sense, they are *compact*). The quality of the representation achieved by a pattern can be quantified by using some statistical measures. Depending on their measures, patterns can describe relevant data properties (in this sense, they are *rich in semantics*).

Pattern management is an important issue in many different contexts and domains. The most important context in which pattern management is required is business intelligence and data mining. Business intelligence concerns a broad category of applications and technologies for gathering, storing, analyzing, and providing access to data to help enterprises in business decision process. Data mining is one of the fundamental activities involved in business intelligence applications besides querying and reporting, OLAP processing, statistical analysis, and forecasting. The knowledge units resulting from the data mining tasks can be quite different. As an example, in the context of the market-basket analysis, association rules, involving sold items derived from a set of recorded transactions, are often generated. In addition, in order to perform a market segmentation, the user may also be interested in identifying clusters of customers, based on their buying preferences, or clusters of products, based on customer buying habits. In financial brokerage, users cope with stocks trends derived from trading records. In epidemiology, users are interested in symptom-diagnosis correlations mined from clinical observations.

Pattern management is a key issue also in many other domains not involving directly a data mining process. For instance, in information retrieval, users are interested in extracting keyword frequencies and frequent sets of words appearing in the analyzed documents in order to specialize searching strategies and to perform similarity analysis. Content-based music retrieval is another domain in which patterns have to be managed in order to represent and query rhythm, melody, harmony [Con02]. In image processing, recurrent figures in shapes can be interpreted as specific types of patterns [NPP]. In machine learning [Mit99], predictions and forecasting activities are based on classifiers, which can be interpreted as specific types of patterns.

Recently, pattern management is becoming much more important not only in centralized architectures but also in distributed ones. Indeed, the diffusion of the Web and the improvement of networking technologies speed up the requirement for distributed knowledge discovery and management systems. For instance, in the Web context, sequences of

Application	*Raw data*	*Type of pattern*
market-basket analysis	sales transactions	item association rules
signal processing	complex signals	recurrent waveforms
mobile objects monitoring	measured trajectories	equations
information retrieval	documents	keyword frequencies
image recognition	image database	image features
market segmentation	user profiles	user clusters
music retrieval	music scores, audio files	rhythm, melody, harmony
system monitoring	system output stream	failure patterns
financial brokerage	trading records	stock trends
click-stream analysis	web-server logs	sequences of clicks
epidemiology	clinical records	symptom-diagnosis correlations
risk evaluation	customer records	decision trees

Table 1.1: Examples of patterns

clicks collected by Web Servers are important patterns for clickstream analysis. Moreover, knowledge representation and management, in terms of patterns, is a fundamental issue in the context of the Semantic Web and in agent-based intelligent systems where metadata have to be shared among different parties. Some examples of patterns arising in different application domains are reported in Table 1.1.

While in most cases a pattern is interesting for end-users because it describes a recurrent behavior (e.g., in market segmentation, stock exchange analysis, etc.) observable in raw data, sometimes it is relevant just because it is related to some singular, unexpected event (e.g., in failure monitoring).

Depending on the specific domain, different processes can be used for pattern extraction, e.g., knowledge discovery processes for data mining patterns, feature extraction processes in multimedia applications, or manual processes when patterns are not extracted but directly provided by the user or the application (for example, a classifier not automatically generated from a training set). Moreover, we may be interested not only in patterns generated from raw data by using some extraction or data mining tools (a-posteriori patterns) but also in patterns known by the users and used for example to check how well some data source is represented by them (a-priori patterns). Examples of a-posteriori patterns are, for instance, the previously cited association rules concerning sale transactions, which can be mined from a raw data set by applying the A-priori algorithm [AS94], or the clustering obtained as result of the application of the Single-Link algorithm [JMF99, HK01]. On the other side, a failure event should be considered an a-priori pattern the user wants to check over collected raw data. Note that, also an association rule can be considered an a-priori pattern if the user knows the rule (without applying any mining rule strategy) and she

wishes to verify whether it holds over a certain data source.

Patterns share some characteristics that make traditional DBMSs unable to represent and manage them. As discussed above, patterns can be generated from different application contexts resulting in very heterogeneous structures. Moreover, often, heterogeneous patterns have to be managed together. For instance, in a Web context, in order to well understand e-commerce buying habits of the users of a certain Web site, different patterns can be combined, for example: (i) navigational patterns (identified by clickstream analysis) describing their surfing and browsing behavior; (ii) demographic and geographical clusters, obtained with market segmentation analysis based on personal data and geographical features; (iii) frequencies of the searching keywords specified by the user when using a search engine (typical information treated in information retrieval); and (iv) metadata used by an intelligent agent-based crawling system (typical of the artificial intelligence domain) the user may adopt.

Since source data change with high frequency, another important issue consists in determining whether existing patterns, after a certain time, still represent the data source from which they have been generated, possibly being able to change pattern information when the quality of the representation changes. Finally, all kinds of patterns should be manipulated (e.g. extracted, synchronized, deleted) and queried through dedicated languages.

All the previous reasons motivate the need for the design of ad hoc Pattern Management Systems (PBMSs), i.e., according to Rizzi et al. [RBC+03], systems for handling (storing/processing/retrieving) patterns defined over raw data.

Definition 2 (*PBMS*) A *Pattern-Base Management System* (PBMS) is a system for handling (storing/processing/retrieving) patterns defined over raw data in order to efficiently support pattern matching and to exploit pattern-related operations generating intensional information. The set of patterns managed by a PBMS is called *pattern-base*. □

The Pattern Base Management System (PBMS) is therefore not a simple repository for the extracted knowledge (patterns); rather it is an engine supporting pattern storage (according to a chosen logical model) and processing (involving also complex activities requiring computational efforts). The design of a PBMS relies on solutions developed in several disciplines, such as: data mining and knowledge discovery, for a-posteriori pattern extraction; database management systems, for pattern storage and retrieval; data warehousing, for providing raw datasets; artificial intelligence and machine learning, for pattern extraction and reasoning; metadata management. Pattern management can therefore be seen as a relatively new disciplines lying at the intersection of several well-known application contexts.

Several approaches have been proposed in the literature to cope with knowledge and pattern management problems. They usually deal with some predefined types of knowledge

artifact (e.g., association rules, clusters, or patterns of strings) and they are primary devoted to cope with data mining patterns. Most of them rely on a-posteriori data mining patterns and do not provide a direct support for user-defined and a-priori patterns. In many cases, they mainly concern pattern extraction and exchange issues. Additionally, pattern hierarchies are in general not taken into account and, usually, advanced capabilities for pattern extraction and management are provided for each pattern type but integration of heterogeneous patterns is in general not directly supported.

However, the ability to manipulate different types of patterns under an integrated environment is a fundamental issue for any data-intensive and distributed application, where systems must be able to handle and analyze various types of knowledge, stored in multisite and multiowner data repositories, providing techniques for data analysis and knowledge inference. Nevertheless, a little emphasis has been posed in defining an overall framework to represent and efficiently manage different types of patterns in a uniform manner.

1.2 Research Problems and Thesis Objectives

Starting from the observation that what is still missing is a unified framework dealing with heterogeneous patterns in an homogeneous way, this PhD dissertation addresses this problem by proposing a general framework for pattern representation and management.

The overall aim of this PhD thesis is, therefore, to study the pattern management problem, to investigate the potentiality of a pattern-based management system in details, and to propose a homogeneous, effective, and efficient framework for it. In the remainder of this section the general objectives of this PhD work will be discussed pointing out investigated research problems and thesis goals.

Since the subject of this thesis is related to pattern management and, in particular, it is aimed at investigating the potentiality of a pattern-based management system, the first step of this PhD work consists in identifying the requirements of a pattern management system, starting from the analysis of the existing proposals addressing the pattern management problem.

After that, the first formal contribution of this thesis consist in the introduction of a model for pattern representation supporting all identified requirements is proposed. In particular, the proposed model supports, among the others, several advanced pattern modeling features, such as: user-defined patterns, temporal information representation, validity aspects related to patterns, and pattern hierarchies.

Then, based on the proposed logical model for patterns, ad-hoc pattern manipulation and query languages are proposed. Concerning the query language, both an algebraic and a calculus-based language are provided and proved to be equivalent. Differently from

other existing proposals, they support the combination of possibly heterogeneous patterns together and with raw data. The manipulation language provides the management of both a-posteriori and a-priori patterns and supports various synchronization operations.

The presentation of the pattern model and languages is then completed by the investigation of several theoretical issues concerning the expressive power and complexity of the proposed framework, leading to the identification of conditions under which it becomes tractable while remaining sufficiently expressive.

Finally, in order to show how the proposed pattern management framework can be effectively used in advanced data-intensive applications, this dissertation is completed with the presentation of a prototype system we have developed, implementing the proposed logical model and languages for pattern manipulation and querying.

The technologies used in developing the prototype rely on object-relational database technologies integrated with logical constraint solving features, through the usage of object-oriented programming. In particular, logical constraint solving features, provided by the Sicstus Prolog engine [sic], have been integrated through the usage of object-oriented programming (Java platform [jav] and the Jasper interface [jas] versus Prolog) within the object-relational environment provided by Oracle technologies [Ora]. The developed prototype is called PSYCHO (Pattern based management SYstem arChitecture prOtotype) and it allows the user to use standard pattern types (for instance, a direct support for standard data mining patterns such as association rules and clusters is provided) or define new (possibly, non data mining) pattern types, according to the proposed logical model (through the usage of a dedicated pattern definition language). Moreover, PSYCHO supports pattern generation, manipulation, and analysis under an integrated environment.

1.3 Overview of the Thesis

The remainder of this PhD thesis is organized as follows.

Chapter 2 surveys existing proposals related to the thesis work. The aim of this chapter is twofold: to survey different application contexts in which the need of dealing with heterogeneous types of patterns and to present an analysis of existing theoretical, industrial, and commercial proposals for pattern management.

Chapter 3, starting from the analysis presented in Chapter 2, identifies many different requirements to be taken into account when designing a Pattern Based Management System (PBMS) architecture.

Chapter 4 formalizes the logical models used: the one for data (which is a slight extension of the complex value model proposed in [AB95]) and the one proposed for pattern

representation and management. The proposed pattern model supports all the identified modeling requirements.

Chapter 5 deals with pattern querying and manipulation issues. It introduces ad-hoc pattern query and manipulation languages exploiting all the potentialities of the proposed pattern model. Concerning the query language, both an algebra and a calculus are provided which support the combination of possibly heterogeneous patterns together and with raw data. The manipulation language provides the management of both a-posteriori and a-priori patterns and supports various synchronization operations.

Chapter 6 focuses on several theoretical aspects concerning the proposed framework for pattern management. In particular, the equivalence between the proposed query algebra and calculus is proved and several other theoretical issues concerning the complexity of the proposed framework are investigated. This leads to the identification of conditions under which it becomes tractable while remaining sufficiently expressive.

Chapter 7 presents the prototype framework providing a pattern management solution based on the proposed logical model and languages. The prototype is called PSYCHO which stands for *Pattern based management SYstem arChitecture prOtotype*. We outline that it is developed using object-relational technologies provided by the Oracle platform [Ora].

Finally, Chapter 9 concludes the thesis work by summarizing the PhD research work contributions and by discussing several open issues and topics for future research work.

For easy of reference, two appendixes are provided. Appendix A presents basic notions concerning First-order theories with constraints, whereas in Appendix B the proofs of the main results presented in Chapter 6 are presented.

Chapter 2

Related Work on Pattern Management

As we saw in Chapter 1, knowledge intensive applications rely on the usage of patterns, to represent in a compact and semantically rich way huge quantity of heterogeneous raw data. Several theoretical and industrial approaches (relying on standard proposals, metadata management, and business intelligence solutions) have already been proposed for pattern management. The aim of this chapter is to: (i) present some examples of different patterns arising from several application domains in which a pattern management process could be usefully applied (Section 2.1); (ii) identify key features for pattern management by focusing on a typical data mining scenario (Section 2.2); (iii) survey related work concerning pattern management (Section 2.3). Existing proposals dealing with patterns are then compared with respect to the identified features.

2.1 Application Domains for Pattern Management

Various application domains involving data management (storage, process, retrieve, and data analysis) produce different forms of patterns representing the data insights. In the sequel, we present some representative pattern application domains and the corresponding types of patterns they produce. We start by dealing with data mining (Section 2.1.1), since it is without doubt the most relevant application domain for pattern management. After that, we discuss signal processing (Section 2.1.2), in which pattern management solutions can be very useful in addressing, for example, content-based music retrieval issues. Then, in Section 2.1.3, we review patterns arising in the information retrieval context. Finally, in Section 2.1.4, we briefly discuss many different kinds of patterns arising in mathematic domains.

2.1.1 Data Mining

Data mining is the process of discovering interesting and previously unknown knowledge, i.e. patterns, from large amounts of data stored either in databases, data warehouses or other information repositories. Thus, it constitutes a step in the knowledge discovery process.

There are various types of data stores and database systems on which data mining tasks can be performed, thus different kinds of patterns can be mined. Moreover, there are many different data mining algorithms that accomplish a limited set of tasks producing different types of patterns [BL04], such as: i) the *clustering* techiques which identifies *clusters* that provide a classification scheme, ii) the *classification* of database values with respect to some defined categories, based on a *classifier*, iii) the *rule extraction* producing *association rules* or *frequent itemsets*, iv) the discovery and analysis of *sequences*. In the sequel, we discuss those data mining tasks and the types of patterns they produce.

2.1.1.1 Clustering

Clustering is one of the most useful tasks in data mining processes for discovering groups and identifying interesting distributions in the underlying data. The clustering problem concerns the partitioning of a given data set into groups (i.e., clusters) such that the data in a cluster are more similar to each other than data in different clusters [HK01]. For example, consider a database containing items purchased by customers. A clustering procedure could group the customers in such a way that customers with similar buying habit belong to the same cluster. Thus, the main idea under a clustering process is to reveal *sensible* groups of data, which allow us to discover similarities and differences among data. This idea is applicable in many fields, such as life sciences, medical sciences, and engineering. Moreover, in different contexts, the clustering tasks may have different names. For instance, in pattern recognition the term used is 'unsupervised learning', in biology or ecology, 'numerical taxonomy' is used, in social science the term 'topology' is used, finally, in graph theory clustering is called 'partition'.

It is important to outline that, in the clustering process, there are no predefined classes and no examples that would show what kind of desirable relations should be valid among the data. Indeed, the clustering process may result in different partitionings of a data set, depending on the specific criterion used for clustering. This explain why it is often referred as an unsupervised mining process [BL04]. On the other hand, classification (see Section 2.1.1.2) is a procedure of assigning a data item to a predefined set of categories [FPSS96], which can be viewed as a supervised process.

Many different clustering algorithms have been proposed in literature. We may cite, for example, *Single-link*, *Complete-link*, and *K-means* family algorithms [HK01]. Cluster

analysis is a major tool in a number of applications in many fields of business and science. More specifically, some typical applications of the clustering are in the following fields [HK01, TK03]:

Business: where clustering may help marketers in discovering significant groups in their customers database and characterize them based on purchasing patterns.

Biology: where it can be used to define taxonomies, categorize genes with similar functionality and gain insights into structures inherent in populations.

Spatial data analysis: where the main problem consists in being able to manage and examine the huge amounts of spatial data collected from satellite images, medical equipment, Geographical Information Systems (GIS), image database exploration etc. Clustering may help to automate the process of analysing and understanding spatial data and to reduce the dimension of data to be examine.

Web mining: where clustering is used to discover significant groups of Web documents, which in general are huge collections of semi-structured documents. In this context, the obtained Web document clusters are used to to specialize the information discovery process. In general, clustering may serve as a pre-processing step for other algorithms, such as classification, which would then operate on the detected clusters.

There are many different measures that may be used to evaluate the quality of a clustering result. In general, they can be classified in: *internal quality measures*, which evaluates the goodness of a clustering partition without any reference to external knowledge, and *external quality measures*, which evaluates the quality of the clusters resulting by a clustering process with respect to a well-known taxonomy. For instance, the *Yahoo!* search engine [yah] allows the automatic generation of Web document taxonomies concerning specific application domains. Such document taxonomies can be used to determine whether a clustering technique (or more precisely, the similarity distance notion over which it is based) produces good results or not. For more details, concerning this topic, we refer the reader to [Mla98].

Among the external quality measure, we simply recall the *entropy*, based on probabilistic criteria, and the *F-measure*, based on the notions of 'precision' and 'recall' borrowed from the Information Retrieval context (see below). We refer interested readers to [JMF99, HK01] for more details. However, we are primarly interesting in measures assessing the quality of single clusters, viewed as patterns; thus, in the following, we briefly recall the most important internal quality measure: the cohesiveness.

Cohesiveness The cohesiveness measure (often called *average intra-cluster distance*) is used to evaluates how much similar are the items belonging to a certain cluster. It

can be evaluated by using, for instance, the Euclidean distance between each pair of points [JMF99] and by taking the average value. It takes values inside $[0, 1]$. The higher is the average intra-cluster distance, higher is the cluster cohesiveness, i.e. more similar are the items within the cluster. Many other different distance metrics can be used to evaluated the average intra-cluster distance; usually, the similarity distance used by the clustering algorithm is applied.

2.1.1.2 Classification

Many efforts of statistics, pattern recognition, and machine learning communities have been devoted to the classification problem [RS98]. Classification is the process of finding set of models (or functions) that describe and distinguish data classes or concepts, for the purpose of being able to use the model to predict the class of items whose class label is unknown. The derived model is based on the analysis if a set of *training data* (i.e., items whose label is known) [HK01]. A number of classification techniques have been developed. Among these, the most popular are: *Bayesian classification* and *Decision Trees*.

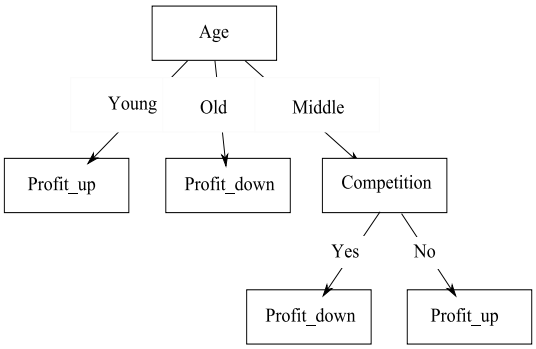
Bayesian classification is a supervised learning approach, based on Bayesian statistical theory and its aim is to classify a sample x to one of the given classes $c_1, c_2, ..., c_N$ using a probability model defined according to Bayes theory [CS96]. Indeed, each category is characterized by a prior probability of observing the category c_i and a given sample x is assumed to belong to a category c_i with the conditional probability density function $p(x|c_i) \in [0, 1]$. Then, for each x, using the above definitions for p and based on Bayes formula, a posterior probability $q(c_i|x)$ can be evaluated. Any new input pattern y is classified into a category c such that c is the class for which $q(c|y)$ is the highest posterior probability.

Decision trees are one of the widely used techniques for classification and prediction [Mit99]. A decision tree is constructed based on a training set of pre-classified data. Each internal node of the decision tree specifies a test of an attribute of the instance and each branch descending of that node corresponds to one of the possible values for this attribute. Finally, each leaf corresponds to one of the defined classes. The classification based on decision trees is another example of supervised learning. Indeed, whenever a new input data has to be classified using a decision tree, the procedure is the following: (i) start at the root of the tree and test the input with respect to the attribute specified by this node; (ii) move to the next internal node following the branch to traverse according to the test result; (iii) iterate the process until a leaf is reached. The final class of the leaf node reached is the class to which the input data has to be assigned.

The most commonly used technique between the ones described above are decision trees, since they are more comprehensible for human users and they can be constructed by exploiting only the information contained in the training data set in contrast to Bayesian

Example Data	Age	Competition	Type	Profit
1	old	yes	software	down
2	old	no	software	down
3	old	no	hardware	down
4	midlife	yes	software	down
5	midlife	yes	hardware	down
6	midlife	no	hardware	up
7	midlife	no	software	up
8	young	yes	software	up
9	young	no	hardware	up
10	young	no	software	up

(a) Training data set



(b)

```
IF Age='Young' THEN Profit ='Up'

IF Age='Old' THEN Profit ='Down'

IF Age='Middle' AND Competition = 'Yes'
THEN Profit ='Down'

IF Age='Middle' AND Competition='No'
THEN Profit ='Up'
```

(c) Rules describing the decision tree in Fig. 2.1(b)

Figure 2.1: Decision tree classification: 2.1(a) the training data set, 2.1(b) the decision tree, and 2.1(c) decision tree induced rules

approach which requires additional information (e.g. prior probabilities). Another important aspect concerning the knowledge produced by the decision trees is that it can be easily represented in the form of rules, which are easier for human users to understand, particularly when the tree is very large. Indeed, a rule can be created for each root-to-leaf path of the decision tree. Thus, one rule can be associated with each leaf. The antecedent of this rule is the conjunction of all conditions associated to each edge leading to the leaf and the consequent is the class prediction. Rules describing knowledge codified by the tree in Fig. 2.1(b), built over the training data set in Fig. 2.1(a), are shown in Fig. 2.1(c).

As we have already discussed, classification is a form of data analysis that can be used to define data models describing data classes or predict data trends. It is widely used for supporting decisions in business and science.

There exist many different measures that may be used to evaluate the quality of a classifier (e.g., a decision tree). In the following, we briefly recall, the most common (for more details

see [HK01]).

Sensitivity The sensitivity of a classifier is used to evaluate how well it recognizes the positive samples. It is the ratio between the number of true positive (i.e., items that are correctly classified as positive samples) and the total number of positive samples. More specifically, let $|true\_pos|$ be the number of true positive samples and $|pos|$ be the total number of positive samples, it is defined as follows:

$$Sensitivity = |true\_pos|/|pos|$$

Specificity The specificity of a classifier is used to evaluate how well it recognizes the negative samples. It is the ratio between the number of true negative (i.e., items that are correctly classified as negative samples) and the total number of negative samples. More specifically, let $|true\_neg|$ be the number of true negative samples and $|neg|$ be the total number of negative samples, it is defined as follows:

$$Specificity = |true\_neg|/|neg|$$

Precision The precision measures the percentage of items classified as positive samples that actually are positive samples. More specifically, let $|true\_pos|$ be the number of true positive samples and $|false\_pos|$ be the number of false positive samples (i.e., items that are incorrectly classified as positive samples, but they are not), it is defined as follows:

$$Precision = |true\_pos|/(|true\_pos| + |false\_pos|)$$

2.1.1.3 Association Rules

Association rules reveal underlying interactions between attributes in the data set. Given a domain D of values and a set of transactions, each corresponding to a subset of D, an association rule takes the form $B \Rightarrow H$, where $B \subseteq D$, $H \subseteq D$, $H \cap B = \emptyset$. H is often called the head of the rule, while B is its body. The informal meaning of the rule is that, given a transaction T, it often happens that when T contains B then it also contains H. This qualitative information can be quantified by using measures giving an indication of the association rules importance and reliability. Such measures are often called quality measures or interestingness measures [AS94, HK01].

A well-known method to mine association rules from transactional dataset is the *Apriori* algorithm proposed by Agrawal and Srikant in 1994 [AS94]. According to this approach, the process of mining association rules is based on two steps. First, find all frequent itemsets, i.e., sets of items (itemsets), which occur at least as frequently as a (pre-fixed)

minimum support. Second, generate association rules from those frequent itemsets. These rules must satisfy minimum support and minimum confidence criteria.

Another approach for association rules mining is the *FP-growth* algorithm [HPY00], which produce association rules by avoiding a candidate generation phase (for a survey and a comparison of existing mining algorithms for association rules we refer the reader to [HGN00]).

There are many different measures that may be used to evaluate the quality of an association rules of the form $B \rightarrow H$:

Confidence The confidence (or strength) of an association rule is the ratio between the number of transactions satisfying both rule body (B) and head (H) and the number of transactions satisfying just the body. More precisely:

$$confidence = |H \cap B|/|B|$$

where $|X|$ denotes the number of transactions covered by the element $|X|$. It takes values inside $[0, 1]$. The higher is the confidence value more important is the association rule.

Support The support of an association rule is the ratio between the number of transactions satisfying the body of the rule (i.e., having items appearing in B) and the total number of transactions. More precisely:

$$support = |B|/N$$

where N is the total number of considered transactions. It takes values inside $[0, 1]$. An association rule with support value near to 1 can be considered as an important association rule.

Coverage The coverage of an association rule is the ratio between the number transactions satisfying the rule (i.e., having all items appearing in both B and H) and the total number of transactions. More specifically, coverage is defined as:

$$coverage = |H \cap B|/N$$

where N is the total number of transactions under consideration and $|H \cap B|$ denotes the number of transactions covered by H and B. Coverage can be considered as an indication of how often a rule occurs in a data set and, therefore, it can be used to evaluate how a rule is significant with respect to the data set.

Leverage The leverage of an association rule is a more complex statistical measure evaluating the proportion of additional cases covered by both B and H with respect to those expected if B and H were independent of each other. This is a measure of the

importance of the association that includes both the strength and the support of the rule. More specifically, it is defined as follows:

$$leverage = support(H|B)(support(LHS) * support(RHS))$$

It takes values inside $[-1, 1]$. Values of leverage equal or under 0, indicate a strong independence between B and H. On the other hand, values of leverage near to 1 are indication of an important association rule.

Lift The lift of an association rule is the confidence divided by the proportion of all cases that are covered by the H. This is a measure of the importance of the association and it is defined as follows:

$$lift = confidence(B \rightarrow H)/support(H).$$

It takes values inside $\mathbb{R}^+$ (the space of the real positive numbers).

Among the above discussed quality measures for association rules, support and confidence are the most widely used and they are also known as Agrawal and Srikants itemset measures [AS94]. From their definitions, we could say that confidence corresponds to the strength of a rule while support corresponds to its statistical significance.

Mining rules is one of the main tasks in a data mining process, which has been widely studied since rules provide a concise statement of potentially useful information that is easily understood by the end-users. The typical application of association rule mining is market-basket analysis, aimed at identifying customer-buying habits by finding associations between the different items that customers place in their 'shopping baskets'. The discovery of such associations can help retailers develop marketing strategies by finding which items are frequently purchased together by customers. However, in general terms, the discovery of interesting association relationships among huge amounts of business or scientific databases records can help in decision making processes.

2.1.1.4 Sequential patterns

Sequential pattern mining is the mining of frequently occurring patterns related to time or other sequences. Often, the term 'time series analysis' is used. Most studies of sequential patterns mining concentrate on symbolic patterns.

The problem of mining sequential patterns can be stated as follows. Given a potentially large sequence (e.g., a string) S, we are interested in sequences of literals (i.e., sequential patterns) of the form $a \rightarrow b$, where a, b are substrings of S, such that the frequency of ab is not less than some minimum support and the probability that a is immediately followed by b is not less than a minimum confidence.

Moreover, constraints on the kinds of sequential patterns the user is interested in can be specified through string 'templates', such as: serial episodes, parallel episodes, or regular expressions [HK01]. A serial episode is a set of events that occur in a certain order whereas a parallel episode is a set of events whose occurrence ordering is not important. For instance, the sequence $A \rightarrow B$ is a serial episode implying that B follows A while $A\&B$ is a parallel episode indicating that both A and B occur in our data but their ordering is not important. The user can also specify constraints in the form of regular expressions. For example, the template $(A|B)C * (D|E)$ indicates that the user would like to find patterns where first A and B occur but without taking into account their relative ordering, then the event C occur one or more times, and finally D and E occur in some undetermined order.

In daily and scientific life sequential data are available and used everywhere. Some representative examples are text, weather forecast data, satellite data streams, business transactions, telecommunications records, experimental runs, DNA sequences, histories of medical records. Discovering sequential patterns is a key issue in all those application domains, since it may help the user or scientist in interpreting recurring phenomena or extracting similarities, and in predicting coming events. Indeed, for example, in genetics the identification and study of the human genome lead to the discovery of DNA sequences which are the keys of living organism functionalities.

2.1.2 Signal Processing: Content-based Music Retrieval

Large-scale storage of sound and music issues have been emerged in recent years when the wide-area distribution of multimedia over the Internet has become possible. This rises several new requirements for data management systems, which have to provide flexible and powerful support for music and audio data. Some basic issues when dealing with music data and patterns are primary connected to compression of audio and protection of digital data aimed at safe delivery of music through the Web or private networks. On the other hand, such data are massive and huge catalogs of music data are available for users. Therefore, issues concerning content-based search have to be taken into account [PRC00].

We point out that music data are quite complex in their structure and they can be described by using several different patterns revealing one of its characteristics, such as: harmony, melody, rhythm, etc. Therefore, content-based music retrieval is a difficult task rising at least the following problems: instrument recognition, melody spotting, musical key extraction, musical pattern recognition, composer recognition, music structure extraction and music segmentation.

In order to support music data and pattern management, a variety of signal processing solutions have been developed to address the challenges imposed by music retrieval, in particular focusing on music recognition and understanding aspects.

Several researchers have addressed the problem of feature extraction from stored pieces of music. Feature selection is a topic of ongoing research and various solutions have been proposed in the context of machine learning [HYG02]. Stochastic modeling and dynamic time warping techniques have also been proposed for the comparison of music signals. For example, Hidden Markov Models have been used to build statistical representations of musical pieces and based on them music similarity have been investigated [PTK02]. Moreover, some approaches for discovering musical structure in audio recordings have proposed [Con02, DH02] in order to be used for investigating the music data similarity based on their structure.

The above discussion clearly suggests that, in order to support music retrieval, the potentiality of a pattern management system has to be exploited. In particular, functionalities aimed at representing different kinds of music patterns, automatically extracted from huge quantity of music data, and comparing them are a basic requirement for such a management system.

2.1.3 Information retrieval

Another research field where patterns are involved is that of Information Retrieval [BYRN99]. In this context, the goal is to retrieve from a *corpus* - i.e., a collection of text documents - information satisfying user requests. The main problem concerns the fact that user queries are often vaguely defined, in contrast to queries posed over traditional Database systems, due to a lack of a query language or algebra. In general, a user query consists of few keywords specifying several arguments the user is interested in and she wants to retrieve in the corpus those documents concerning these arguments. In many cases, users expectations are not satisfied.

Assume to have a corpus of n documents where each document is indexed by m index terms. The entire corpus can be represented as an $m \times n$ matrix A. Each column of A represents a document in the corpus. Thus, the entry $A(i, j)$ codifies the frequency of the occurrence of the i-th index term in the j-th document in the corpus. Using the traditional *Vector-Space model*, each index term is considered as a base vector in an m-dimensional space U. Thus, each document D is a vector which is a linear combination of several base vectors corresponding to index terms appearing in D. Similarly, queries are linear combinations of index terms base vectors in U. The goal of the retrieval system is to return among documents in the corpus those documents most similar to the query, according to the cosine distance between the vector modeling the query $\vec{q}$ and the vector modeling a document $\vec{d}$ in the corpus. More formally, given a query vector $\vec{q}$ and a corpus of several documents each represented by a vector document $\vec{d_i}$, the set of interesting documents has the form: $D = \{d | cos(\vec{q}, \vec{d}) \geq t\}$, where t is a user-defined threshold. Under a knowledge management perspective, each query q (i.e., each query vector $\vec{q}$) can

be seen as a pattern representing, at an high level of abstraction, a set of documents (i.e., all interesting document vectors $\vec{d}$ such that $cos(\vec{q}, \vec{d})$ is greather than a certain threshold value).

The key assumption under this approach is the orthogonality of index terms, i.e., no two index terms are correlated in their appearance within a text. Of course, this is not the case, when the corpus contains human-generated discourses. Indeed, humans often uses synonymies, for instance 'car' and 'automobile', and polysemonies, i.e., words which semantics is context-dependent, such as 'jaguar', which can be used to refer to the wild animal or to a special kind of car.

A possible solution to solve those problems relies on identifying 'patterns' in terms usage. In other words, one identifies text clusters, where synonymous terms, although distinct, have similar usage characteristics, i.e. they are used in similar contexts. Under this perspective, a polysemous word is expected to have distinct usage patterns for each distinct meaning, so a text referring to a jaguar in a zoo, shall appear in a different cluster than a text about Jaguar X-type. This methodology is known as *Semantic Indexing*. In particular, the Latent Semantic Indexing (LSI) [DDL$^+$90] applies the well known algebraic decomposition, the Singular Value Decomposition (SVD), to the matrix A term-document matrix A. In essence, LSI projects the original vector space spanned by the columns of A, to a low-dimensional 'semantic' space, where only the important correlations of indexing terms appear, as those appear in the spectral structure of the matrix. Also queries are projected to the induced space, and then documents are retrieved with respect to the cosine distance, leading to more efficient results. There are several applications of LSI aimed at discovering important patterns in word usage. For instance, LSI method is efficiently used when searching bilingual corpora (English and French) or in searching textual documents containing spelling errors, which is a common situation when dealing with texts produced by an OCR software [DLLL97]. Moreover, this is fundamental, for example in the data warehousing context, in order to allow metadata exchange between different environments.

There are two basic measures for assessing the quality of a text retrieval solution: *precision* and *recall*.

Precision Precision is the percentage of retrieved documents that are in fact relevant to the query (i.e., correct query responses). More precisely, let *Relevant* be the set of documents relevant for a query and *Retrieved* be the set of all retrieved documents, it is defined as follows:

$$precision = |Relevant \cap Retrieved|/|Retrieved|$$

Recall This is the percentage of documents relevant for the query that are effectively retrieved. More precisely, it is defined as follows::

$$Recall = |Relevant \cap Retrieved|/|Relevant|$$

2.1.4 Mathematics

In mathematical science, many different kinds of patterns arises. Some of them are very simple and familiar to us, such as, for instance: (i) prime numbers (e.g. $2, 3, 5, 7, 11, 13, ...$), (ii) composite numbers (e.g. prime factorization, such as: $6 = 2 * 3$), (iii) square numbers (e.g. $25 = 5^2$, $36 = 6^2$, ...), (iv) Fibonacci numbers (i.e., $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$).

In those case, the pattern expresses a common characteristic all instances have to satisfy. In other words, it acts as a constraint. In some applications, such as telecommunication data fault analisys [BCH00] or in equation solvings [SB99], predefined behaviour of a number sequence can be very useful in order to understand the trends of collected data.

Besides numbers, many different other types of patterns arise in mathematics. Among those, we recall, as a typical example, equations, which may produce graph when solved according to one of its variables. We may identify families of equations, for example, linear equations (e.g., $y = x + 7$ producing graph of a line in a 2D coordinate system) or non linear ones (e.g., $y = x^2 + 3$ and $x^2 + y^2 = 4$ producing, respectively, a parabola and a circle in a 2D system). Moreover, some equations (or equation systems) describe patterns (e.g., triangles, squares, n-vertexes polygon) that may be recurrent in several shapes. Such patterns are widely used in geometric modeling applications in order to reconstruct and reproduce complex polygonal images [CN94] or in image processing to recognize familiar shapes in images [NPP].

On the other side, equations are fundamental in the context of moving objects in order to describe (portion of) trajectories of the monitored objects. Indeed, positions of a moving object are taken at certain instant of time. The entire trajectory of the moving object is approximated by the interpolating equation (possibly not linear) covering all 2D points corresponding to the observed positions. In this case, the quality of the moving object trajectory description can be measured by using the variance of the distances of moving object positions with respect to the corresponding point over the equation.

Finally, we recall that mathematical constraints are used to represent and manipulate in a finite way infinite set of data. Thus they can be considered patterns, independently from the chosen mathematical theory for expressing them. They are exploited in the context of constraint databases [Rev95a, KKR90]. For example, in a constraint database, the conjunction of dis-equality constraints over reals: $X < 2 \wedge Y > 3$, where X and Y are real variables, represents the infinite set of tuple having the X attribute less than 2 and the Y attribute greater than 3. Thus, constraints are a powerful mechanism that can finitely model infinite information, such as spatial data. Indeed, spatial objects can be seen as infinite sets of points, corresponding to the solutions of particular arithmetic constraints. For example, the constraint $X^2 + Y^2 \leq 9$ represents a circle with center in point $(0, 0)$ and with radius equal to 3.

2.2 Key Features in PBMS Evaluation

In the following we first present a typical data mining scenario (Section 2.2.1) and then, based on it and on the peculiarities of patterns arising in different application domains presented in the previous section, we present useful parameters in comparing existing pattern management solutions. In particular, we consider three different aspects: (i) architecture for a pattern management system (Section 2.2.2); (ii) pattern models (Section 2.2.3); (iii) pattern languages (Section 2.2.4).

2.2.1 A Data Mining Scenario

According to what previously stated in Section 2.1, the market-basket analysis is a typical data mining application concerning the task of finding and handling association rules and clusters concerning customer's transactions. Suppose a commercial vendor traces shop transactions concerning milk, coffee, bread, butter, and rice, and applies data mining techniques to determine how she can further increase her sales. The vendor deals with different kinds of patterns: association rules, representing correlations between sold items; clusters of association rules, grouping rules with respect to their similarity; clusters of products, grouping products with respect to their type and price. Now, we suppose the vendor wants to execute the following operations:

Step 1: modeling heterogeneous patterns. Since the vendor deals with (at least) three different types of patterns she would like to generate and manage those patterns all together, in the same system, in order to be able to manipulate all these knowledge in an integrated way.

Step 2: periodic pattern generation. At the end of every month, the vendor mines from his transaction data association rules over sold products, by filtering interesting results with respect to certain thresholds. She considers reliable the rules extracted from the instant in which they have been generated till the last day of the month. The vendor then groups the rules into clusters.

Step 3: pattern querying. The vendor may be interested in analyzing patterns stored in the system, i.e., to retrieve patterns satisfying certain conditions, to combine them in order to construct new patterns, to establish whether a pattern is similar to another one, and to correlate patterns and raw data. For instance, the vendor may be interested in retrieving all association rules mined during March 2005 involving *bread* or similar items, or in identifying all association rules extracted from a certain set of transactions, with a reasonable level of detail (i.e., with quality measures higher than specified thresholds). In order to solve the last query, both the data management and the pattern management system have to be used.

Step 4: promotion of a new product. From April 2005, the vendor will start to sell a certain product P. To promote P in advance, she may promote some other products she already sells, for which there exists a correlation with P, in order to stimulate the demand for P. In this way, it is possible that customers will start to buy P without the need for a dedicated advertising campaign. In order to know, for example, whether *bread* may stimulate the sale of P, she may insert in the system an association rule such as $bread \rightarrow P$ (not automatically generated) and verify whether it holds or not with respect to the recorded transactions.

Step 5: pattern update, synchronization, and deletion. Patterns may have to be updated. For instance, the user may know that the quality of the representation that a pattern achieves with respect to its data source has been changed because source data have been changed, thus the pattern quality measures (evaluated at insertion time) have to be updated, since the pattern may no longer be semantically valid, i.e. it may not correctly represent the updated source data. As an example, when on April 1 2005, the vendor starts to sell a new product P, new raw data concerning sales are collected, new patterns are generated and, at the same time, patterns previously extracted may not correctly represent source data. Thus, a synchronization is required between data and patterns to reflect to patterns the changes occurred in raw data. In this case, the measures may change. Finally, there is the need for pattern deletion operations. For example, the vendor may be interested in deleting all patterns that are no more semantically valid or in removing from the system all rules having 'rice' as value in its head or its body.

In the following sections, based on the previous scenario and on the characteristics of different kinds of patterns introduced in Section 2.1, we present several useful parameters in comparing existing pattern management solutions. In particular, Section 2.2.2 deals with possible solutions for architecture for a pattern management system, Section 2.2.3 is focused on pattern models, and Section 2.2.4 relies on pattern languages.

2.2.2 Architecture for a Pattern Base Management System

The architecture of a PBMS can be integrated or separated. In an integrated architecture, raw data and patterns are stored together, by using the same data model, and managed in the same way. On the other side, in a separated architecture, raw data are stored and managed in a traditional way by a DBMS whereas patterns are stored and managed by a dedicated PBMS.

Since in the integrated architecture a unique data model is used for both data and patterns, design of the pattern base is simplified. For example, an association rule can be represented in the relational model by using a set of relational tuples, each containing the head of the

rule and one element in the body. However, traditional data models may not adequately represent all patterns characteristics, thus making manipulation operations more complex. Further, by storing patterns with data, we rely on traditional DBMS capabilities for what concerns query expressive power and query optimization. In particular, under an integrated architecture, the mining process is usually seen as a particular type of query. However, patterns may require sophisticated processing that, in traditional systems can only be implemented through user-defined procedures.

Separated architectures manage data and patterns by using two distinct systems. Thus, two models and languages have to be used to deal with pattern based applications. The usage of a specific pattern data model guarantees a higher and more tailored expressive power in pattern representation. Moreover, operations over data are activated by the PBMS only by demand, through the so-called *cross-over queries*. The PBMS can therefore support specific techniques for pattern management and retrieval. Mining operations are not part of the query language; rather, they are specific manipulation operators. Finally, specific query languages can be designed providing advanced capabilities, based on the provided pattern representation.

2.2.3 Pattern Models

We can define a pattern model as a formalism by which patterns are described and manipulated inside the PBMS. In defining a pattern model, we believe that the following aspects should be taken into account.

Pattern heterogeneity support. The ability to model heterogeneous patterns is very important to make the PBMS flexible and usable in different contexts. Most of the systems allow the user to manipulate different types of patterns (see Step 1 of the scenario, for an example in the data mining context) that usually correspond to different knowledge extraction process results. Moreover, the ability to support user-defined pattern types is a fundamental requirement for a PBMS. However, in many cases, patterns of different types cannot be used 'together' in a unified framework. Moreover, often, it is not possible for the user to define new pattern types, which are therefore predefined.

Relation between raw data and patterns. By the definition of pattern it follows that each pattern represents a possibly huge set of source data. Thus, there exists a relationship among each pattern an the data source it represents. Thus, in order to make the pattern richer in semantics and provide additional significant information for pattern retrieval, it may be useful to store such a relation. In the case of patterns generated by applying a mining technique, the data source associated with a pattern is the dataset from which the patterns has been mined. Most of the systems recognize

the importance of this aspect and provide a mechanism to trace the source data set from which a pattern has been generated. In the proposed scenario, this corresponds to maintain information concerning the dataset from which association rules have been extracted. Such information can then be used, for example, to solve some of the queries pointed out in Step 3 of the scenario. On the other side, in a mathematical context the set of points interpolated by a strict line have to be associated with the line itself. Besides the source dataset, it may be useful to exactly know the subset of the source dataset represented by the pattern. For examples, to generate rule $bread \rightarrow milk$ only transactions containing 'bread' and 'milk' are considered from the overall set of transactions in the source dataset. This subset can be represented in a precise way, by listing its components, or approximate way, by providing a formula satisfied by the elements of the source dataset from which the pattern has been probably generated. Most of the systems do not support the representation of this relationship or support it only in an approximated way.

Quality Measures. It is important to be able to quantify how well a pattern represents a raw data set, by associating each pattern with some quantitative measures. Quality measures vary with respect to the structure of the pattern and with respect to the application domain one considers. For example, in the identified scenario, each association rule mined from data is associated with *confidence* and *support* values. On the other side, if we consider an information retrieval scenario *precision* and *recall* values, of a set of text documents with respect to a certain input query, would be stored. Most of the systems allow the user to express this quality information, which is in general computed during pattern generation and never modified.

Temporal features. Since source data change with high frequency, it is important to determine whether existing patterns, after a certain time, still represent the data source from which they have been generated. This happens when, given a pattern p extracted at time t, the same pattern p can be extracted at time $t' > t$ from the same raw dataset, with the same or better measure values. In this case, we say the pattern is semantically valid at time t'. When this happens and measures change, the system should be able to change pattern measures. In the last decade, a lot of work of the database community was concentrated in the definition and formalization of temporal databases [TCG+93, Sno95, RS02, BFGM03, BJW00] in order to offer the possibility of dealing with time-varing data which are typical of many different application domains. Therefore, according to the temporal database theory, two different time information for patterns can be considered: (i) a *transaction time*, corresponding to the time the pattern "starts to live" in the system and (ii) a *validity time*, corresponding to the instant(s) (eventually, periods of tyme) in which the pattern is assumed to be reliable with respect to its data source.

Hierarchies over types. Another important feature a pattern management system should provide is the capability to define some kind of hierarchy over the existing pattern types, in order to introduce relationships such as specialization or composition that increase expressivity, reusability and modularity. For instance, in the proposed scenario, the vendor deals with association rules and with more complex patterns that are clusters of association rules (see Step 1). Thus, a composition relationship is exploited.

2.2.4 Pattern Languages

As in any DBMS, in a PBMS at least two different types of languages must be provided: the *Pattern Manipulation Language (PML)*, providing the basic operations by which patterns can be manipulated (e.g. extracted, synchronized, deleted), and the *Pattern Query Language (PQL)*, supporting pattern retrieval.

PQL queries take as input pattern sets and data sets and returns patterns or data. On the other hand, PML operations take as input a pattern dataset and return a new pattern set, which replaces in the pattern base the input one. Aspects concerning both manipulation and query languages for patterns can be evaluated by means of several parameters introduced in the following.

2.2.4.1 Pattern Manipulation Language Parameters

Automatic extraction. This is the capability of a system to generate patterns starting from raw data using a mining function. It corresponds to the extraction phase of a knowledge data discovery process and generates a-posteriori patterns. In the proposed scenario, association rules generated in Step 2 of the scenario represent a-posteriori patterns.

Direct insertion of patterns. There are patterns that the user knows a-priori and wish to verify over a certain data source. They are not extracted from raw data, but inserted directly from scratch in the system. Ad hoc primitives are therefore needed to perform this operation. In the proposed scenario, patterns described in Step 4 are examples of a-priori patterns.

Modifications and deletions. Patterns can be modified or deleted. For example, users may be interested in updating information associated with patterns (such as their validity in time or the quality of raw data representation they achieve, represented in terms of measures) or in removing from the system patterns satisfying (or do not satisfying) certain characteristics. For instance, in the proposed scenario (Step 5), the user is interested in removing an association rule when it does not correctly

represent anymore the source data set. Not all the systems guarantee sufficiently expressive deletion and update operations over patterns; in many cases, only pattern generation and querying are provided.

Synchronization over source data. Since modifications in raw data are very frequent, it may happen that a pattern, extracted at a certain instant of time from a certain data source, does not correctly represent the data source after several modifications occur (Step 5). Thus, the need for a synchronization operation arises in order to align patterns with data source they represent. This operation is a particular type of update operations for patterns. For instance, in the proposed scenario (Step 5), the user is interested in updating the measure values associated with a certain rule (such as: $bread \rightarrow P$) when the source data change. Synchronization can also be executed against a different dataset in order to check whether a pattern extracted from a certain data source holds also for another data source. In this case, we call it 'recomputation'. For example, suppose the vendor receives a data set DS concerning sales in the month of January 2005 in another supermarket. She may be interested in checking whether the association rules mined from her data set represent reasonable patterns for the new data set DS. Unfortunately, synchronization between raw data and patterns (Step 5) is rarely supported.

Mining function. Patterns are obtained from raw data by applying some kind of mining function, e.g. the APriori [AS94] algorithm can be used to generate association rules (Step 2), the regression method can be used to determine the line which approximate a point distribution, and the vector-space technique can be used to determine the set of relevant documents for a query. The presence of a library of mining functions and the possibility to define new functions if required makes pattern manipulation much more flexible.

2.2.4.2 Pattern Query Language Parameters

Queries against Patterns. The PBMS has to provide a query language to retrieve patterns according to some specified conditions. For example, all association rules having 'bread' in their body may need to be retrieved (Step 3). In general, pattern collections have to be supported by the system in order to be used as input for queries. Similarly to the relational context where a relation contains tuples with the same schema, patterns in a collection should have the same type. Moreover, it is highly desirable the language to be closed, i.e. each query over patterns must return a set of patterns of the same type over which other queries can be executed.

Pattern combination. Operations for combining patterns together should be provided as an advanced form of reasoning. Combination can be seen as a sort of 'join' between

patterns. For example, transitivity between association rules can be seen as a kind of pattern join.

Similarity. An important characteristic of a pattern language is the ability to check pattern similarity, based on pattern structure and measures. Only few general approaches for pattern similarity have been provided that can be homogeneously applied to different types of patterns. Moreover, only few existing pattern query languages support such kind of operation.

Queries involving source data. According to the chosen architecture and the logical model, a system managing patterns has to provide operations not only for querying patterns but also data. Such queries are usually called cross-over queries. When the system adopts a separated architecture, cross-over operations require the combination of two different query processors in order to be executed. In our scenario, the second query of Step 3 is a cross-over query.

2.3 Related Work on Pattern Management

In the following, we revise related work concerning pattern management by considering scientific proposals, industrial proposals, and commercial systems.

In general, scientific community efforts mainly deal with the definition of a pattern management framework providing a full support for heterogeneous pattern generation and management, thus providing back-end technologies for pattern management applications. Examples of these approaches are the 3W Model [JLN00] and the inductive databases approach - investigated in particular in the CINQ project [CIN01, MLK04].

On the other hand, industrial proposals mainly deal with standard representation purposes for patterns resulting from data mining and data warehousing processes, in order to support their exchange between different architectures. Thus, they mainly provide the right front-end for pattern management applications. Examples of such approaches are the Predictive Model Markup Language [PMM], the Common Warehouse Model [CWM05], the SQL/MM data mining part [SQLa], and the Java Data Mining API [JDM04].

Moreover, since patterns can be seen as a special type of metadata, pattern management has also some aspects in common with metadata management and, since they are strategical to enhance the market competiveness, pattern management is addressed also in the context of commercial DBMSs.

Before briefly introducing existing proposals for pattern management, we notice that all the proposed approaches are mainly focused over data mining patterns. In presenting solutions

for pattern management, we use the previously identified key features (see Section 2.2) to compare them.

The remainder of this section is organized as follows. Section 2.3.1 surveys theoretical proposals for pattern management mainly addressing modeling, extraction, manipulation and pattern querying issues. Section 2.3.2 is focused on standards for pattern representation proposed by the industrial world. Section 2.3.3 presents metadata management solutions that can be applied to patterns. Finally, Section 2.3.4 discusses how patterns are supported by the most popular commercial Data Based Management Systems.

2.3.1 Theoretical Proposals

In the scientific community, the need for a unified framework supporting pattern management is widespread and it covers many different contexts and domains. Thus, many efforts have been put in the formalization of the overall principles under which a Pattern Based Management System can be developed, providing the background for the development of back-end technologies to be used by pattern-based applications. In the sequel, we briefly present the following two theoretical proposals for pattern management:

- Inductive Databases Approach [IM96, Rae02, CIN01]: an inductive framework where both data and patterns are stored at the same layer and treated in the same manner;
- 3-Worlds Model [JLN00]: a unified framework for pattern management based on the definition of three distinct worlds: an intensional world (containing intensional descriptions of patterns), an extensional world (containing an explicit representation of patterns), and a world representing raw data.

2.3.1.1 Inductive Databases Approach

Under the inductive database approach [IM96, Rae02], patterns are represented according to the underlying model for raw data. More precisely, the knowledge repository is assumed to contain both datasets and pattern sets. Within the framework of inductive databases, knowledge discovery is considered as an extended querying process [MLK04, RJLM02]. Thus, a language for an inductive database is an extension of a database language that allows one to: (i) select, manipulate and query data as in standard queries; (ii) select, manipulate and query patterns; and (iii) execute cross-over queries over patterns. Queries can, then, be stored in the repository as views, in this way datasets and pattern sets are intensionally described.

Inductive databases have been mainly investigated in the context of the CINQ project of the European Community [CIN01], which tries to face both theoretical and practical

issues of inductive querying for the discovery of knowledge from transactional data. CINQ covers several different areas, spreading from data mining task to machine learning. The considered data mining patterns are itemsets, association rules, episodes, clusters, etc. In the machine learning context, interesting patterns considered by the project are equations describing quantitative laws, statistical trends, and variations over data.

From a theoretical point of view, a formal theory is provided for each type of patterns, providing: (i) a language for pattern description; (ii) evaluation functions, for computing measures and other significant data related to patterns; (iii) primitive constraints for expressive basic pattern properties (e.g. minimal/maximal frequency, minimal accuracy). By using primitives constraints, extraction and further queries (seen as a postprocessing step in the overall architecture) can be interpreted as constraints and executed by using techniques from constraint programming, using concepts from constraint-based mining. Other manipulation operations, such as the insertion of patterns from scratch in the system, are delegated to the underlying DBMS. Note that since a theory is provided for each type of patterns, integration is not an issue of the project.

From a more practical point of view, extension of existing querying standards, such as SQL, have been provided in order to query specific types of patterns, mainly association rules. The combination of a data mining algorithm (usually some variant of the Apriori algorithm [AS94]) with a language such as SQL (or OQL) offers some interesting querying capabilities. Among the existing proposals, we recall the following:

DMQL (Data Mining Query Language) [HFW+96] is an SQL-based data mining language for generating patterns from relational data. An object-oriented extension of DMQL, based on Object Query Language (OQL [BEJ+00], has been presented in [ESF01]. Discovered association rules can be stored in the system, but no postprocessing (i.e., queries over the generated patterns) is provided. Indeed, they are simply presented to the user and a further iterative refining of mining results is possible only through graphical tools. The obtained rules can be specialized (generalized) by using concept hierarchies over source data. Besides association rules, other data mining patterns can be generated, such as: data generalizations (a sort of aggregates), characteristic rules (assertions describing a property shared by most data in certain data set, for example the symptoms of a certain disease), discriminant rules (assertions describing characteristics that discriminate a dataset from another one), data classification rules (patterns for data classification). For each type of pattern, a set of measures is provided (e.g. confidence and support for association rules) and conditions over them can be used in order to generate only patterns with a certain quality level.

MINE RULE [MPC98] extends SQL with a new operator, named *Mine Rule*, for discovering association rules from data stored in relations. By using the Mine Rule

operator, a new relation with schema (Body, Head, Support, Confidence) is created, containing a tuple for each generated association rule. The body and the head itemsets of the generated rules are stored in dedicated tables and referred within the rule-base table by using foreign keys. The cardinality of the rule body as well as minimum support and confidence values can be specified in the Mine Rule statement. Mine Rule is very flexible in specifying the subset of raw data from which patterns have to be extracted as well as conditions that extracted patterns must satisfy. However, no specific support for post-processing (queries) is provided, even if standard SQL can be used since rules are stored in tables. Similarly to DMQL, hierarchies over raw data can be used to generalize the extracted association rules, or said better, to extract only association rules at a certain level of generalization. A similar operator, called XMine, for extracting association rules from XML documents has been presented in [BCC+02].

Mine-SQL (MSQL) [IV99] is another SQL-like language for generating and querying association rules. Similarly to MINE RULE approach, only association rules are considered. Also in this case, input transactions and resulting rules are stored in relations. With respect to MINE RULE, it supports different types of statements, one for rule extraction (GetRules), one for rule post-processing (SelectRules), and some predicates for cross-over queries, i.e., queries involving both data and patterns, (Satisfy, Violate). Concerning extraction, MSQL is less flexible in specifying the source data set; indeed, it must be an existing table or view. However, similarly to MINE RULE, constraints over the rules to be generated can be specified. Extracted patterns can be further queried using the SelectRules operator. Various conditions can be specified, depending on the body and the head of the rules. By using the SelectRules statement, it is also possible to recompute measures of already extracted rules over different datasets. In order to explicitly support cross-over queries, MSQL proposes the operators satisfy and violate. They determine whether a tuple satisfies or violates at least one or all the association rules in a given set, specified by either using GetRules or SelectRules commands.

Table 2.1 presents an example of usage of the just presented languages for extracting association rules from sale transactions stored in a relation Transactions, storing them, when possible, in relation MarketAssRules.

We finally recall that results achieved in the context of the CINQ project have been experimented in a machine learning context with the implementation of a molecular fragment discovery demo system [Mol04]. On the other side, concerning a pure data mining context, they have been experimented in the demo version of Minerule Mining System [Min04] dealing with association rules over transactional data.

```
Find association rules
from Transactions
with support
threshold=0.3
with confidence
threshold=0.9
```

(a)

```
Mine Rule MarketAssRules
AS
select distinct
1..n item as Body,
1..n item as Head,
Support,
Confidence
from Transactions
group by tr♯
extracting rules with
Support:0.3,
Confidence:0.9
```

(b)

```
GetRules(Transactions)
into MarketAssRules
where confidence > 0.9
and support > 0.3

SelectRules(MarketAssRules)
where body has (bread=yes)

select *
from Transactions
where VIOLATES ALL (
GetRules(Transactions)
where body has (bread=yes)
and confidence > 0.75 )
```

(c)

Table 2.1: Languages for patterns proposed in the context of the Inductive Databases: (a) DMQL pattern extraction statement; (b) Mine Rule statement; (c) Mine-SQL statements

2.3.1.2 3-Worlds Model

The 3-Worlds (3W) Model [JLN00] is a unified framework for pattern management. Under this approach, the pattern model allows one to represent three different worlds: the intensional world (I-World), containing the intensional description of patterns; the extensional world (E-World), containing an extensional representation of patterns; and the data world (D-World), containing raw data.

In the I-World, patterns correspond to (possibly overlapping) regions in a data space, described by means of linear constraints over the attributes of the analyzed data set. For example, a cluster of products based on their price in dollars can be described by the following constraint '10 <= price <= 20' (call it region *cheap_product*). More complex regions can be defined by using disjunctions of constraints. A collection of intensional patterns (i.e., regions) constitutes a '*Dimension*'.

In the E-World, each region is represented in its extensional form, i.e., by an explicit enumeration of the members of the source space satisfying the constraint characterizing the region, i.e. the intensional pattern. Thus, for instance, the extension corresponding to region cheap_product (contained in the I-World) contains all source data items with price between 10 and 20.

Finally, the D-World corresponds to the source data set, in the form of relations, from

which regions and dimensions can be created as result of a mining process.

Note that regions in the I-World are not predefined, thus user-defined patterns are allowed within this framework. Indeed, to define a new pattern, it is sufficient to describe a spatial region by using linear constraints. Furthermore, each region can be associated with a number of attributes, including measures, which do not have a special treatment. A generic $3W$ pattern p has, therefore, the following form:

$$p = (\bigvee_i (\bigwedge_j C), a_1, ..., a_n)$$

where C is a linear disequality constraint with real coefficients over the dataset attributes (i.e., attributes in the relations stored in the D-World) and $a_1, ..., a_n$ are attributes (or properties) associated with each pattern. Essentially, a pattern in this framework models a spatial region described by $\bigvee_i (\bigwedge_j C)$ and each a_i represents a semantic property associated with this region.

From the manipulation side, the framework does not support a-priori patterns. Indeed, operations to directly insert patterns in the system are not supported as well as deletion operation. Query languages for all the worlds have been proposed. In particular, for the D-World and the E-World, traditional relational languages can be used (with some minor extensions for the E-World). On the other hand, a dimension algebra has been defined over regions in the I-World, obtained by extending relational languages. The main operations of this language are described in the following:

- The selection operation allows pattern retrieval by invoking various spatial predicates such as overlap ($||$), containment ($\subseteq$), etc. between regions.
- The projection operation corresponds to the elimination of some property attribute i.e., it amounts to setting this value to 'true' in every region.
- A purge operator, returning inconsistent regions, i.e., regions whose constraint cannot be satisfied by any data point (thus, with an empty extensional representation). For instance, a region with constraint 'price $>$ 20 AND price $<$ 10' is clearly inconsistent, since the constraint is intrinsically unsatisfiable.
- Traditional relational operators (cartesian product, union, minus, and renaming) have then been extended to cope with sets of regions.

The following cross-over operators are also provided that allow the user to navigate among the three worlds: (a) automatic extraction of patterns (mine); (b) the assignment of an extension to a region (populate), given a certain data source; (c) the detection of the regions corresponding to a certain extension (lookup); (d) a sort of synchronization, providing the

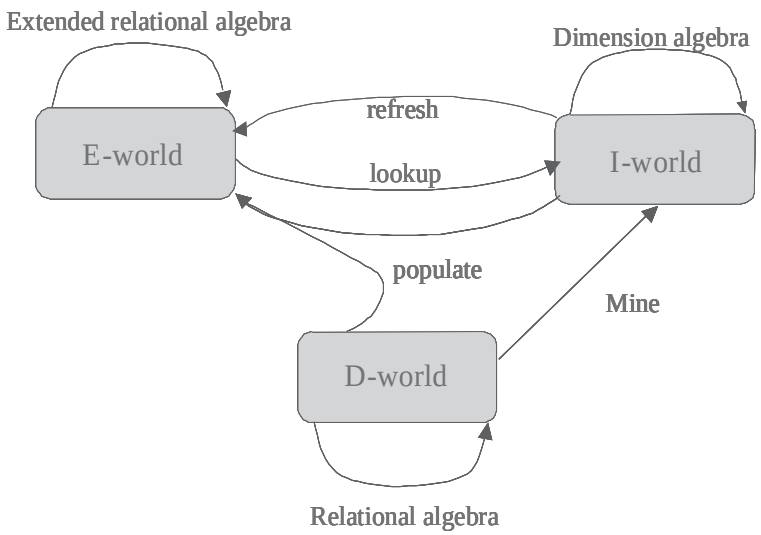


Figure 2.2: 3W algebraic operators

computation of new extensions starting from combinations of regions and a given dataset (refresh). Fig. 2.2 shows these operators.

We remark that, even if no manipulation language is explicitly provided in this approach, some of the proposed operators can be interpreted as manipulation operations over patterns or data when assuming to change the three worlds according to the query result. For example, the mine operator can be seen as a pattern manipulation operator when the result of the mining is made persistent in the I-World.

2.3.1.3 Concluding Discussion

The features of the frameworks presented above, according to the parameters introduced in Section 2.2, are summarized in Table 2.2.

As we can see from Table 2.2, concerning the architecture, the inductive database approach - studied in depth in the context of the CINQ Projects [CIN01] - does not propose a dedicated model for patterns. Rather, it exploits the underlying data model, thus it adopts an integrated approach. On the other hand, 3W relies on separated worlds for data management and pattern management.

For what concerns the model, the more general approach seems to be 3W, where there is no limitation on the pattern types that can be represented and hierarchies of patterns can be modeled. Moreover, it provides a precise mapping between patterns and source data (i.e., the extensional representation of patterns). Finally, it provides a support (even if limited) for pattern temporal validity information, whereas in the inductive database

approach temporal aspects are not directly addressed.

Concerning the manipulation language, 3W and CINQ do not support direct insertion of patterns as well as deletion and update operations. On the other hand, all the proposals take into account synchronization (recomputation) issues. Concerning the query language, the two approaches propose either one (or more) calculus or algebra languages, providing relational operators. Specific characteristics of languages provided in the context of inductive databases are summarized in Table 2.3. Concerning extracted patterns, MINE RULE and MSQL deals only with association rules, whereas DMQL and ODMQL deal with many different types of patterns. When patterns are stored, SQL can be used for manipulation and querying (including cross-over queries). Among the proposed languages, however, only MSQL proposes ad-hoc operators for pattern retrieval and post-processing.

As a final consideration, we observe that when dealing with applications managing different types of patterns (for example, this is the case of advanced knowledge discovery applications), among theoretical frameworks the 3W Model is the best solutions since it provides support for heterogeneous patterns in a unified way. On the other side, the inductive database approach provides better solutions for specific data mining context, such as association rules management, with a low impact on existing SQL-based applications.

2.3.2 Standards for Pattern Management

The industrial community has devoted significant efforts towards solving the pattern management problem and it has proposed some standards to support pattern representation and management in the context of existing programming languages and database (or data warehousing) environments in order to achieve interoperability and (data) knowledge sharing. Thus, they provide the right front-end for pattern management applications. In general, they do not support generic patterns and, similarly to the inductive database approach, specific representations are provided only for specific types of patterns. Moreover, they do not provide support for inter-pattern manipulation.

Some proposals, such as Predictive Model Markup Language [PMM] and Common Warehouse Data Model [CWM05], mainly deal with data mining and data warehousing pattern representation, respectively, in order to support their exchange between different architectures. Some others, such as Java Data Mining [JDM04] and SQL-MM Data Mining[SQLa], provide standard representation and manipulation primitives in the context of Java and SQL, respectively. In the following, all these proposals will be briefly presented and compared with respect to the parameters previously introduced.

	Parameter	Inductive Databases	3W Model
Model & Architecture	*Type of Architecture*	Integrated	Separated (3 layers: source data, mined data, and intermediate data)
	Predefined types	Itemsets Association Rules Sequences Clusters Equations	Users can define their own types, that must be represented as sets of constraints
	Link to source data	Yes, datasource is part of the architecture	Yes, datasource is one of the layers of the architecture. Pattern-Data relationship is precise
	Quality measures	Yes	Yes, as a semantical property
	Mining function	Yes, the mining process is a querying process	Yes
	Temporal features	No	Limited support, as semantical properties
	Hierarchical types	No	Yes
Manipulation	*Manipulation Language*	Manipulation through constraint-based querying and SQL	Yes
	Automatic extraction	Yes, constraint-based queries	Yes
	Direct insertion	No	No
	Modifications and Deletions	Yes (SQL)	No
	Synchronization over source data	Yes, recomputation	Yes, recomputation
	Mining function	No	No
Query	*Queries against patterns*	Constraint-based calculus	Algebra
	Pattern combination	No	Yes, Cartesian Product
	Similarity	No	No
	Queries involving source data	Yes	Yes

Table 2.2: Feature Comparison: Theoretical Proposals

	Parameter	DMQL & ODMQL	MINE RULE	MSQL
Model & Architecture	*Predefined types*	Association Rules Data generalization Characteristics rules Discriminant rules Data classification rules	Association Rules	Association Rules
Manipulation	*Manipulation Language*	Only extraction, but no storage	Only extraction	Yes
	Automatic extraction	Yes	Yes	Yes
	Direct insertion	No	Using standard SQL	Using standard SQL
	Modifications and Deletions	No	Using standard SQL	Using standard SQL
	Synchronization over source data	No	No	Yes, recomputation
Query	*Queries against patterns*	Only visualization and browsing	Using standard SQL	SQL-like
	Pattern combination	No	No	No
	Similarity	No	No	No
	Queries involving source data	No	Using standard SQL	Yes

Table 2.3: Feature Comparison: Theoretical Proposals (Query Languages)

2.3.2.1 Predictive Model Markup Language

PMML [PMM] is a standardization effort of DMG (Data Mining Group) an it consists in an XML-based language to describe data mining models (i.e., the mining algorithm, the mining parameters, and mined data) and to share them between PMML compliant applications and visualization tools. Figure 2.3 shows an extract of a PMML association rule mining model. Since PMML is primary aimed to the exchange of data between different architectures, no assumptions about the underlying architecture are done.

PMML traces information concerning the data set from which a pattern has been extracted by allowing the user to specify the data dictionary, i.e. the collection of raw data used as input for the mining algorithm. Concerning the mining function, it is possible to express the fact that a certain pattern has been mined from a certain raw data set by using a specified mining algorithm. However, no assumption is made on the existence of a mining library. For instance, in the example shown in Figure 2.3, the represented association rule is the result of the application of the A-Priori algorithm (algorithmName = 'Apriori').

Moreover, PMML does not allow the user to define its own types. Indeed, one can only define models of one of the predefined types, that, however, cover a very large area of the data mining context (see Table 1.1). It is also important to note that PMML allows the user to represent also information concerning quality measures associated with patterns. Due to its nature, PMML does not provide temporal features. Even if no general support for pattern hierarchies is provided, in PMML 3.0 refinement is supported for decision trees and simple regression models. More general variants may be defined in future versions of PMML.

All major commercial products supporting data knowledge management and data mining attempt to be compliant with PMML standard. Among them we recall Oracle Data Mining tool in Oracle10g [Ora], DB2 Intelligent Miner tools [DB2], and MS SQL Server 2005 Analysis Services [SQLb].

2.3.2.2 Common Warehouse Metamodel

CWM [CWM05] is a standardization effort of the OMG (Object Management Group) and it enables easy interchange of warehouse and business intelligence metadata between warehouse tools, platforms and metadata repositories in distributed heterogeneous environments. CWM is based on three standards: (i) UML (Unified Modeling Language) [UML], an object oriented modeling language, used for representing object models; (ii) MOF (Meta Object Facility) [MOF], which defines an extensible framework for defining models for metadata and provides tools with programmatic interfaces to store and access metadata in a repository; (iii) XMI (XML Metadata Interchange) [XMI], which allows metadata compliant with the MOF meta-model to be interchanged as streams or files with a standard

```
<PMML>
...
<!--items in input data for the mining of association rule #1 -->
  <Item id="1" value="milk"/>
  <Item id="2" value="coffe"/>
  <Item id="3" value="bread"/>
   ...
<!-- definition of the mining model used -->
<AssociationModel modelName="mba"
<!—mining algorithm used -->
      algorithmName="Apriori"
<!-- tuples in Transactions -->
      numberOfTransactions="10"
<!-- thresholds for support and confidence -->
      minimumSupport="0.3"
      minimumConfidence="0.9"
...
<!-- item sets involved in association rule #1 -->
  <!-- item set containing the item corresponding to ‘coffee’ -->
  <Itemset id="1" numberOfItems="1"> <ItemRef itemRef="2"/> </Itemset>
  <!-- item set containing the item corrsponding to ‘bread’ -->
  <Itemset id="2" numberOfItems="1"> <ItemRef itemRef="3"/> </Itemset>
...
<!-- association rules -->
<AssociationRule support="0.3" confidence="1.0" antecedent="1‟ consequent="2"/>
...
</AssociationModel>
</PMML>
```

Figure 2.3: PMML example

XML-based format. CWM has been defined as a specific metamodel for generic warehouse architectures. Thus, it is compliant with the MOF metamodel and relies on UML for the object representation and notation. Since MOF is a metamodel for metadata, also UML metamodels can be represented in MOF. This means that both CWM metadata and UML models can be translated into XML documents by using XMI through the mapping with MOF.

CWM consists of various meta-models, including a meta-model for data mining (CWM-DM), by which mining models and parameters for pattern extraction can be specified.

Unfortunately, CWM has been designed to analyze large amounts of data, where the data mining process is just a small part. Only few pattern types can be represented: clustering, association rules, supervised classification, approximation, and attribute importance. The user has not the capability to define its own pattern types and no temporal and hierarchical information associated with patterns can be modeled.

Finally, no dedicated languages for query and manipulation are proposed, since it is assumed manipulation is provided by the environment importing CWM-DM metadata.

Due to the complexity of the model, CWM is supported to some extent by most commercial systems providing solutions for data warehousing, such as Oracle, IBM (within DB2), Genesis and Iona Technologies (providing e-datawarehouse solutions), Unisys (providing backbone solutions for UML, XMI and MOF, core tools for CWM development). However, CWM-DM is rarely integrated in specific solutions for data mining where often only import/export in PMML, providing a much more simple pattern representation, is provided.

2.3.2.3 SQL-MM - DM

The International Standard ISO/IEC 13249 (ISO SQL-MM part 6) - 'Information technology - Database languages - SQL Multimedia and Application Packages (ISO SQL-MM)' [SQLa] is a specification for supporting data management of common data types (text, spatial information, images, data mining results) relevant in multimedia and other knowledge intensive applications through SQL. It consists of several different parts. Part number 6 is devoted to data mining aspects. In particular, it attempts to provide a standardized interface to data mining algorithms that can be layered at the top of any object-relational database system and even deployed as middleware when required, by providing several SQL user-defined types (including methods on those types) to support pattern extraction and storage.

Differently from PMML and CWM, it does not only address the issue of representation but also of manipulation. Thus, it can be used to develop specific data mining applications on top of an object-relational DBMS (ORDBMS). Four types of patterns are supported (thus,

the set of pattern types is not extensible and no support for user-defined pattern types is provided): association rules, clusters, regression (predicting the ranking of new data based on an analysis of existing data), and classification (predicting which grouping or class new data will best fit based on its relationship to existing data). For each pattern type, a set of measures is provided. For each of those models, various activities are supported:

- *Training*: the mining task (also called the model) is specified by choosing a pattern type, setting some parameters concerning the chosen mining function, and then applying the just configured mining function over a given dataset.
- *Testing*: when classification or regression are used, a resulting pattern can be tested by applying it to known data and comparing the pattern predictions with that known data classification or ranking value.
- *Application*: when clustering, classification or regression are used, the model can then be applied to all the existing data for new classifications or cluster assignment.

All the previous activities are supported through a set of SQL user-defined types. For each pattern type, a type DM_*Model (where the '∗' is replaced by a string identifying the chosen pattern type) is used to define the model to be used for data mining. The models are parameterized by using instances of the DM_∗Settings type, which allows various parameters of a data mining model, such as the minimum support for an association rule, to be set. Models can be trained using instances of the DM_ClassificationData type and tested by building instances of the DM_MiningData type that holds test data, and instances of the DM_MiningMapping type that specify the different columns in a relational table that are to be used as a data source. The result of testing a model is one or more instances of the DM_*TestResult type (only for classification and regression). When the model is run against real data, the obtained results are instances of the DM_*Result type. In most cases, instances of DM_*Task types are also used to control the actual testing and running of your models.

Since SQL-MM is primary aimed at enhancing SQL with functionalities supporting data mining, no specific support is provided for a-priori patterns. Advanced modeling features, such as the definition of pattern hierarchies and temporal information management, are not taken into account. However, queries over both data and patterns can be expressed, through SQL. In the same way, the specified mining model and patterns can be modified or deleted.

2.3.2.4 Java Data Mining API

The Java Data Mining (JDM) API [JDM04] specification addresses the need for a pure Java API to facilitate the development of data mining applications. While SQL-MM deals with

representation and manipulation purposes inside an ORDBMS, Java Data Mining is a pure Java API addressing the same issues. As any Java API, it provides a standardized access to data mining patterns that can be represented according to various formats, including PMML and CWM-DM. Thus, it provides interoperability between various data mining vendors by applying the most appropriate algorithm implementation to a given problem without having to invest resources in learning each vendor's API.

JDM supports common data mining operations, as well as the creation, storage, access, and maintenance of metadata supporting mining activities, under an integrated architecture, relying on three logical components: (i) Application Programming Interface (API), which allows end-users to access to services provided by the data mining engine (DME); (ii) Data Mining Engine (DME), supporting all the services required to the mining process, including data analysis services; and (iii) Mining Object Repository (MOR), where data mining objects are made persistent together with source data. Various technologies can be used to implement the MOR, such as a file-based environment, or a relational/object database, possibly based on SQL-MM specifications. The MOR component constitutes the repository against which queries and manipulation operations are executed.

Through the supported services, a-posteriori patterns of predefined types (see Table 2.4) can be generated by using several different mining functions. Similarly to SQL-MM, pattern extraction is executed through tasks, obtained by specifying information concerning the type of patterns to be extracted, the source dataset, the mining function and additional parameters. Each generated pattern is associated with some measures, representing the accuracy with respect to raw data. Patterns are then stored in the MOR and then used for mining activities. JDM supports various import and export formats, including PMML.

2.3.2.5 Concluding Discussion

A coincise feature comparison of existing standards is reported in Table 2.4. According to those information, we point out that, all the industrial proposals described above rely on an integrated architecture, where patterns are represented by using the underlying data model. Among them, PMML and CWM-DM simply address the problem of pattern representation. On the other hand, SQL-MM and JDM cope with both pattern representation and management. All standards provide a support for the representation of common data mining patterns. However, no user-defined patterns can be modeled, i.e., the set of pattern types is not extensible and no advanced modeling features concerning patterns, such as temporal information management associated with patterns and definition of hierarchies involving patterns, are supported. Concerning pattern management, no dedicated languages for pattern manipulation are supported.

Finally, we outline that, since PMML and CWM simply address the issue of pattern representation, they can be used in any Pattern Based Management System architecture.

As we will see later, most commercial systems support PMML, which guarantees a clear XML representation that can be easily integrated with other XML data; on the other hand, due to its complexity, CWM-DM is rarely supported. Differently, SQL-MM and JDM can be used to develop specific data mining applications on top of existing technologies. In particular, SQL-MM can be put on top of an ORDBMS environment, whereas JDM works in a JAVA-based environment, providing an implementation for the proposed API.

2.3.3 Metadata Management

Pattern can be interpreted as a kind of metadata. Indeed, metadata in general represent data over data and, since patterns represent knowledge over data, there is a strong relationship between metadata management and pattern management. However, as we have already stressed, pattern management is a more complex problem since patterns have some peculiar characteristics that general metadata do not have. Indeed, metadata are usually provided for maintaining process information, as in data warehousing, or for representing knowledge in order to guarantee interoperability, as in the Semantic Web and intelligent agent systems. Specific pattern characteristics, such as quantification of importance through quality measures, are not taken into account. Usually, metadata are not used to improve and drive decision processes. Since however metadata management has somehow influenced pattern management, in the following we briefly describe some approaches defined in this context.

In the artificial intelligence area many research efforts have been invested in the Knowledge Sharing Effort [KSE], a consortium working on solutions for sharing and reuse of knowledge bases and knowledge bases systems. Standards proposals of such a consortium are computer-oriented, i.e. they are not dedicated to human users even if in some cases they can take advantages in using the proposed standard languages. The most important contributions developed by the consortium are Knowledge Interchange Format (KIF) and Knowledge Query and Manipulation Language (KQML) specification. The first one is a declarative language to express knowledge about knowledge and it is used to exchange knowledge units among computers. It does not provide support for internal knowledge representation, thus each computer receiving KIF data translate them into its internal logical model in order to be able to apply some computation process. On the other side, the second contribution proposes a language and a protocol supporting interoperability and cooperation among collections of intelligent agents involved in distribute applications. KQML can be used as a language by an application to interact with an intelligent agent system or by two or more intelligent systems to interact cooperatively in problem solving.

Concerning the emerging Semantic Web research area, Web metadata management problem has been taken into account by the W3C and a framework for representing information in the Web, Resource Description Framework [RDF], has been proposed. One of the es-

	Parameter	PMML	JDM API	SQL-MM	CWM
Model & Architecture	*Type of architecture*	No architecture, only representation of patterns	Integrated	Integrated	No architecture, only representation of patterns
	Predefined types	Association Rules Decision tree Center/distribution based clustering (General) regression Neural networks Naive Bayes Sequences	Clustering Association Rules Classification Approximation Attribute importance	Clustering Association Rules Classification Regression	Clustering Association Rules Supervised classification Approximation Attribute importance
	Link to source data	Yes	Yes	Yes	Yes
	Quality measures	Yes	Yes	Yes	Yes
	Temporal features	No	No	No	No
	Hierarchical types	Partial	No	No	No
Manipulation	*Manipulation Language*	No	Java API	SQL	No
	Automatic extraction	No	Yes	Yes	No
	Direct insertion	No	Yes	Yes	No
	Modifications and Deletions	No	Possible using direct access to object via Java	Possible using direct access to object via SQL	No
	Synchronization over source data	No	No	No	No
	Mining function	Yes	Yes	Yes	Yes
Query	*Queries language*	No	Java API	SQL	No

Table 2.4: Feature Comparison: Industrial Proposals

sential goals of RDF (and RDF-Schema) is to allow - in a simple way - the description of Web metadata, i.e. information about Web resources and how such resources can be used by third-party, in order to make them available not only for human users but also for machines and automatic processes. RDF uses an XML-based syntax and it exploits the URI identification mechanism. Recently, an emerging research field coping with the integration of ontology management and Web data management has been started up. In this context, W3C proposes a recommendation for a dedicated language: the Web Ontology Language [OWL]. OWL is primary dedicated to applications that need to process the content of information instead of just presenting information to human users. OWL supports better machine interpretability of Web content than XML, RDF, or RDF Schema (RDF-S) solutions since it provides an extended vocabulary along with a more precise semantics.

Issues concerning metadata management have also been extensively considered in the context of the Dublin Core Metadata Initiative [DCM], "an organization dedicated to promoting the adoption of interoperable metadata standards and developing specialized metadata vocabularies for describing resources that enable more intelligent information discovery systems". The main aim of DCMI is to support Internet resources identification through the proposal of metadata standards for discovery across domains and frameworks (tools, services, and infrastructure) and for the metadata sharing.

2.3.4 Pattern Support in Commercial DBMSs

Since the ability to support business intelligence solutions enhances the market competiveness of a DBMS product, all the most important DBMS producers supply their products with solutions for business intelligence supporting data mining and knowledge management processes. Pattern management in commercial DBMSs is provided in the context of such environments. In the remainder of this section we will briefly discuss data mining solutions proposed by three leading companies in database technology: Oracle, Microsoft, and IBM.

2.3.4.1 Oracle Data Mining Tool

Starting from release 9i, Oracle technology supports data mining processing. In Oracle Data Mining 10g [Ora], basic data mining features have been specialized and enhanced. Oracle Data Mining (ODM) is a tool tightly integrated with Oracle DBMS supporting basic data mining tasks, such as classification, prediction, association, clustering and ranking attribute importance.

By using ODM, the user can extract knowledge (in the form of different kinds of patterns) from corporate data lying in the underlying Oracle databases or data warehouses. Supported patterns include categorical classifiers (computed by applying Naive Bayes Network

or Support Vector Machines), continuous/numerical classifier relying on linear or non-linear regression models (obtained by Support Vector Machines), association rules, clusters (produced by the K-Means algorithm or a proprietary clustering algorithm, named O-Cluster). Mining algorithms and machine learning methods are built-in in ODM, but the user may define a binning procedure to prepare input data and change some settings and/or define new parameters for the mining model through the ODM Java API. Statistical measures can be associated with classifiers and association rules.

In the latest release (i.e., ODM 10g-Release2), ODM's functionalities can be accessed in two ways: through a Java-based API or through the PL/SQL interface. Up to now, Java API and PL/SQL API are not interoperable, i.e. a model created in Java cannot be used in PL/SQL and vice-versa. To overcome this limitation, the next ODM releases will adhere to the JDM standard specification and a JDM API will be implemented as a layer on top of ODM PL/SQL API and the current Java API will be abandoned.

Concerning other data mining standards, ODM supports PMML import and export, but only for Naive Bayes and association rule models. Exchanges through PMML documents are fully supported between Oracle database instances, but the compatibility with PMML models produced by some other vendors can be achieved only if they use core PMML standard.

2.3.4.2 Microsoft SQL Server 2005

The business solution proposed by Microsoft SQL Server exploit OLAP, data mining and data warehousing tools [SQLb]. The pioneer data mining functionalities appeared in SQL Server 2000 (only two types of patterns were supported: decision trees and clusters), but they have been consolidated and extended in the recent SQL Server 2005 release. Within the SQL Server environment, there are tools supporting data transformation and loading, pattern extraction and analysis based on OLAP services.

SQL server 2005 allows the user to build different types of mining models dealing with traditional mining patterns (such as decision tree, clusters, Naive Bayes classifier, time series, association rules, and neural networks) and to test, compare, and manage them in order to drive the business decision processes. Seven mining algorithms are provided by SQL Server 2005, but the user may create new mining algorithms.

The entire knowledge management process is performed through a Mining Model Editor to define, view, compare, and apply models. Besides this editor, additional tools are provided to exploit other mining phases (for example, data preparation). Within the SQL Server 2005, through OLE DB for Data Mining [OLE], it is possible to mine knowledge from relational data sources or multi-relational repository. OLE DB for Data Mining extends SQL to integrate data-mining capabilities in other database applications. Thus, it provides

storage and manipulation features for mined patterns in an SQL style. Using OLE DB for Data Mining, extracted patterns are stored in a relational database. To create a mining model, a CREATE statement quite similar to the SQL CREATE TABLE statement can be used; to insert new patterns in your mining model, the INSERT INTO statement can be used; finally, patterns can be retrieved and predictions made by using the usual SQL SELECT statement. For the sake of interoperability and compatibility with standards, OLE DB for Data Mining specification incorporates PMML.

2.3.4.3 IBM DB2 Intelligent Miner

DB2 databases management environment provides support for knowledge management by mean of a suite of software tools, called *DB2 Intelligent Miner* [DB2], dedicated to the basic activities involved in the whole data mining process. In particular, DB2 Intelligent Miner supports fraud detection, market segmentation and customer profiling, and market basket analysis. Users may exploit all data mining functionalities as any other traditional relational function provided by the DBMS. The interaction between DB2 Intelligent Miner's tools takes place through SQL, Web Services, or Java.

In particular, an ad-hoc DB2 Extender for data mining allows the automatic construction of mining models within DB2/SQL applications and their update with respect to changes occurring in the underlying raw data. The generated mining models are PMML (vers. 2.0) models and are stored as Binary Large Objects (BLOBs). The other DB2 tools supporting training, scoring (or prediction), and visualization of a model works on PMML models, thus they can manage without additional overhead third-party PMML models. It is quite important to note that the scoring tool has the ability to score a mining model over data recorded not only on DB2 databases but also on Oracle ones. This capability has a great impact in applications development since it may reduce design and building costs.

Since DB2 Intelligent Miner's tools are tightly integrated with the database environment and the mining results are stored as BLOB, the user may interact with the system through an SQL API. In particular, by using SQL it is possible to perform association rules discovery, clustering and classifications techniques provided by the DB2 environment. Then, through ODBC/JDBC or OLE DB data mining results can be integrated within business applications developed using an external powerful programming language such as Java, for instance. The analysis of the mining results is supported by a Java-based results browser (named DB2 Intelligent Miner Visualizator).

Parameter	Oracle Data Mining (10g)	Microsoft SQL Server 2005	IBM DB2
Predefined types	Association rule Discrete and continuos classifier Cluster Attribute importance	Association rule and itemset Cluster Decision tree Naive Bayes classifier Time series Neural Networks	Association rule and itemset Cluster Tree classifier
Quality measures	Yes	Yes	Yes
Mining function	Built-in (user-defined settings)	Built-in (user-defined settings)	Built-in (user-defined settings)
Temporal features	No	No	Yes (scoring phase)
Hierarchical types	No	No	No
Supported Standards	PMML JDM (ODM10g-Rel.2)	PMML	PMML

Table 2.5: Feature Comparison: Commercial DBMSs

2.3.4.4 Concluding Discussion

As we have seen, commercial DBMSs do not provide a comprehensive framework for pattern management; rather they support business intelligence by providing an applicational layer offering data mining features, in order to extract knowledge from data, and by integrating mining results with OLAP instruments in order to support advanced pattern analysis. For this reason, in general, they do not provide a dedicated logical model for pattern representation and querying, since these aspects are demanded to the applications using the mined results. An exception is represented by SQL Server 2005, where pattern storage, manipulation, and querying is made through OLE DB for Data Mining, which can be considered an SQL-based language for pattern management.

All systems do not allow the user to define its own pattern types. Moreover, mining functions are built-in in the system; however, the user can modify some settings specializing the algorithm to the case he is interested in. Finally, none of the DBMSs takes into account advanced modelling aspects involving patterns, such as temporal information management and the existence of hierarchical relationships between patterns. Actually, only DB2 considers patterns-data synchronization issues, through a scoring mechanism that can be started up by some triggers monitoring raw data changes.

Table 2.5 summarizes the features of the described commercial DBMSs, by considering a subset of the previously introduced parameters.

Chapter 3

Towards a System for Pattern Management

According to what stated in Chapter 1, patterns refer to knowledge artifacts used to represent in a concise and rich in semantics way huge quantity of heterogeneous raw data. Patterns are relevant in any knowledge intensive application, such as, for instance, data mining, information retrieval, or image processing.

From the analysis proposed in Chapter 2, it follows that there is a gap between theoretical proposals for pattern management and standard/commercial ones, that spans from a lack of modeling capabilities (e.g., no support for user-defined patterns, pattern hierarchies, or temporal features management in standard/commercial proposals) to a lack of manipulation and processing operations and tools (no manipulation of heterogeneous patterns, no support for similarity, pattern combination and synchronization in standard/commercial proposals). More generally, the analysis has shown that, even if several proposals exist, an overall framework for representing and manipulating patterns is still missing.

The aim of this chapter is to make one step towards the definition of such a PBMS system by first identifying the basic requirements under which such a system can be developed (Section 3.1) and, then, informally describing a possible reference PBMS architecture (Section 3.2) as well as the features of suitable pattern model (Section 3.3) and languages (Section 3.4). The formal model and formal languages will then be introduced in Chapter 4 and 5, respectively.

3.1 Basic Requirements

As discussed in Chapter 2, there are many different requirements to be taken into account in designing a *Pattern Based Management System (PBMS)*. Some of them impact on the definition of the logical model, which has to be able to support all interesting pattern features pointed out in Section 2.2; some others are related to pattern usage, thus they impact on pattern manipulation, analysis, and querying. In the development of the framework we propose, we plan to take into account the following general requirements in designing the PBMS we propose:

1 - Pattern heterogeneity support. In Section 2.1 we have pointed out that there exist different types of patterns, generated from many different application contexts. In many cases, the need arises of managing such patterns together, in an homogeneaous manner, under a unified environment. For example, association rules and clusters of products are different kinds of patterns. However, in the market-basket analysis, it would be useful to consider both of them in order to rely on a deeper market segmentation knowledge and to be able to enhance the commercial strategy. Therefore, the ability to support many different kinds of patterns is a key requirement we plan to address.

2 - Pattern measures. According to what stated in Section 2.2.3, the support for pattern quality measures is an essential feature for any PBMS. Therefore, it constitutes a key requirement in designing our PBMS solution.

3 - Pattern hierarchies. As we have already discussed in Section 2.2.3, patterns can be quite complex in their structure and can involves hierarchical relationships with other patterns. As emerged from the discussion presented in Section 2.3, pattern hierarchy support is addressed by only few existing proposals in a comprehensive and efficacious way. We plan to address this requirements in designing our system.

4 - Patterns-data correlation. The ability to trace pattern data source is a fundamental issue we take into account in designing our PBMS proposal. Since pattern data source is a collection of raw data, we plan to address two ways of modeling such information: an extensional representation and an intensional representation. Besides the information correlating a pattern with its data source, it could be also useful to be able to identify within its data source which data items are effectively represented by the pattern and which not. However, in many cases such an information cannot be specified in a precise way, but it can be only approximated, by specifying the properties source data, represented by the pattern, satisfy. In designing our PBMS we will take into account also this requirement.

5 - Temporal information associated with patterns. From the pattern management point of view, we have to consider that, typically, in knowledge extensive applications, source data change with high frequency. Thus, an important issue in pattern mangement is related to the pattern-data synchronization, which consists in determining whether existing patterns, after a certain time, still represent the data source from which they have been generated and, possibly, being able to change pattern information when the quality of the representation changes.

Moreover, note that the notion of pattern validity can be interpreted under different points of view. It could be strictly related to the quality of source data representation achieved by the pattern (therefore, it evaluates whether a pattern represents source data in a sufficient good way, i.e. with quality measures better than certain user-specified thresholds) or it may express the validity of the source data representation achieved by the pattern at certain instants of time. The framework for pattern management we propose addresses temporal requirements concerning both pattern validity and pattern-data synchronization aspects.

6 - Pattern query language. A *Pattern Query Language (PQL)* supporting operations against patterns, operations combining patterns together and with source data (the so called cross-over queries), and operations moving up and down within possible pattern hierarchies are required. Another important issue concerns pattern similarity. We however point out that the specific topic concerning pattern similarity definition is a complex research issue and it is out of the scope of the activities of this thesis.

7 - Pattern manipulation language. Several manipulation operations over patterns (for example, pattern generation, insertion, synchronization, and deletion) have to be supported within the PBMS by means of a *Pattern Manipulation Language (PML)*. The first operation a PML must support concern pattern insertion. According to what presented in [RBC+03, BCG+03], pattern can be classified, with respect to how they are inserted in the system as follows:

- *A-posteriori patterns* are extracted from raw data, usually with data mining algorithms. Depending on the kinds of patterns the user is interested in, an ad-hoc mining process can be applied. For instance, if the user is interested in association rules, a variant of the Apriori algorithm [AS94] can be used, whereas, if she wants to analyzed clusters, a clustering algorithm has to be applied [HK01]. Besides association rules and clusters, common examples of a-posteriori patterns are: decision trees, trajectory equations, interpolating lines, etc.
- *A-priori patterns* express a property of the underlying data set that the administrator either knows in advance or plans to impose. For example, an association rule the user knows in advance concerning a certain sale domain can be considered an a-priori pattern. In general, a-priori patterns can be used to test

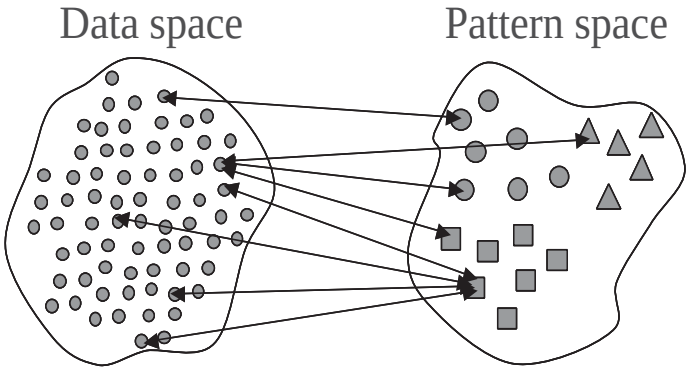


Figure 3.1: Data space and pattern space relationship

whether a certain characteristic is satisfied by underlying raw data. For example, an association rule can be inserted from scratch in the system and then it can be checked over a stored raw dataset in order to verify whether it holds.

In more details, some minimum requirements for the PML, we device the following: (i) the ability to insert in the system both patterns resulting from the application of some mining algorithm and user-defined patterns; (ii) the support for pattern updates, taking into account possible changes occurring in source data and pattern hierarchies, and the possibility to re-align patterns with source data they represent; (iii) the support for pattern deletion even in presence of hierarchical relationship between patterns.

3.2 The Reference Architecture

In the following, we present a reference architecture for designing a PBMS, addressing all basic requirements presented in Section 3.1. As a first consideration, we notice that a PBMS should cope with two distinct spaces: a source data space and a pattern space. There always exist a many-to-many relationship between elements in the data space (i.e., raw data) and elements in the pattern space (i.e., patterns) as shown in Fig. 3.1. Indeed, each pattern may represent more than one data item. On the other hand, items in the source data set may be represented by more than one pattern.

Starting from the fact that the pattern space must provide the representation of heterogeneous patterns according to what discussed in Section 3.1, a fundamental issue to address in designing a PBMS is whether the pattern space is distinct or coincide with the data space. As we saw in Section 2.2.2, in the first case, this means that we need two different logical models for representing the two spaces. On the other hand, if the two spaces coin-

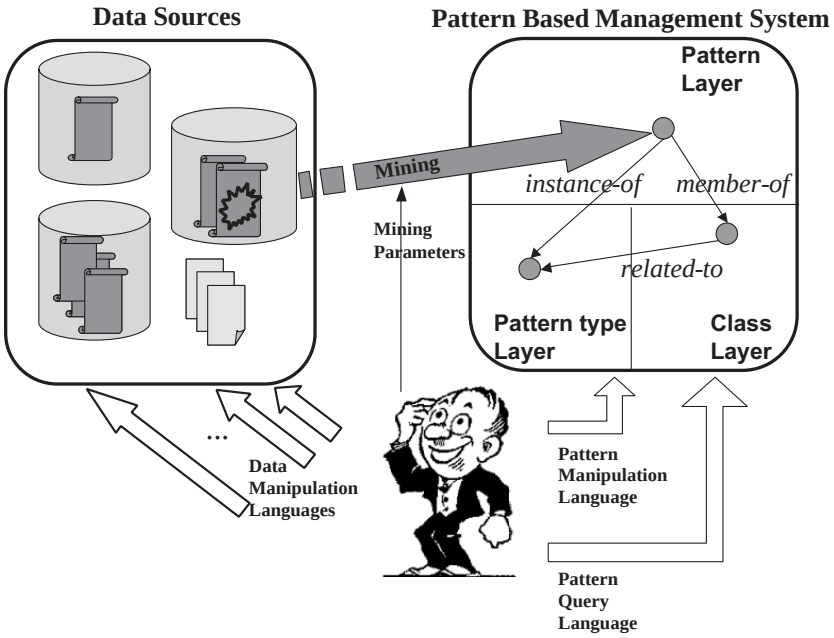


Figure 3.2: The Pattern Based Manegement System Architecture

cide, patterns are interpreted as a special kind of data. Then, since patterns are extracted from data, we may also characterize the pattern space as a view over the data space.

According to what stated in [RBC+03, CMM+04, JLN00], we claim that a clear distinction between the two pattern space is the best solution in order to represent and manage the specifity of patterns. Thus, the reference PBMS architecture we adopt is depicted in Figure 3.2. For the sake of simplicity, in this thesis, we assume source data are stored and managed by a DBMS. Therefore, source data have their own data model. Knowledge discovery algorithms are applied over these data to generate patterns to be fed into the PBMS. Therefore, within the PBMS ad-hoc logical modeling techniques for representing and storing patterns are required.

In our approach, patterns generation algorithms are supported by the manipulation language dedicated to patterns. Indeed, the pattern extraction is supported by the primitives of the pattern manipulation language. Such primitives permit a general approach for a-priori and a-posteriori patterns generation and, in the case of a-posteriori patterns, supports the usage of different pattern extraction algorithms. Moreover, manipulation language features allow the user to specify several mining parameters. On the other side, within the PBMS, the pattern retrieval and querying process is supported by ad-hoc pattern query language primitives. To these purposes, a *Pattern Query Language* and a *Pattern Manipulation Language* will be provided in order to efficiently retrieve patterns, insert, update, delete, or recalculate them against raw data.

3.3 The Reference Pattern Model

The logical model we propose for pattern representation is based on three main concepts: the *pattern type*, the *pattern*, and the *class*. In the following, these concepts are presented in an informal way. The formal definition of the model will be presented in Chapter 4.

3.3.1 Pattern Types

A *pattern type* represents the intensional form of patterns, giving a formal description of their structure, the quality measures they are associated with, their source data, the effective relationship correlating elements in the source data space with elements in the pattern space, and their validity in time. In the case of a-posteriori patterns, source data represents the data set from which they have been mined. On the other hand, in the case of a-priori patterns, source data simply is a data set represented by a pattern. Thus, pattern types play the same role of abstract data types in the object-oriented model. Informally, a pattern type is identified by its *name* and is characterized by the following elements:

- The *structure* schema, which defines the pattern space by describing the structure of the patterns instances of the pattern type. We remark that, the ability to define different structures for patterns is fundamental in order to support pattern heterogeneity (see Section 3.1, requirement 1).
- The *data source* schema, which defines the structure of the datasets from which patterns, instances of the pattern type being defined, are associated with (see Section 3.1, requirement 4). Characterizing the source schema is fundamental for every operation which involves both the pattern space and the source space.
- The *measure* schema, which describes the measures which quantify the quality of the source data representation achieved by the pattern (see Section 3.1, requirement 2). The role of this component is to enable the user to evaluate (through a ready-at-hand information) how accurate and significant for a given application each pattern is.
- The *formula*, which describes the relationship between the source space and the pattern space, thus carrying the intrinsic semantics of the pattern (see Section 3.1, requirement 4). In general, it is a constraint-based formula. Note that, though in some particular domains the formula may exactly express the inter-space relationship, in most cases it describes it only approximatively. Clearly, by exploting this pattern type component the mapping between patterns in the pattern spaces and one or more data items in the data space can be expressed, even if in an approximate way. There are many different formalisms that can be adopted for expressing pattern formulas;

for instance, the linear or the polynomial theories could be used. We will discuss and formalize this point when presenting in a more formal way the logical model for patterns we propose in Chapter 4 and in Chapter 6.

- The *validity period theory* (denote in this chapter by $TIME$), used to express pattern validity information (see Section 3.1, requirement 5). Depending on the application domain and the pattern type, different suitable temporal theories can be used. Clearly, the expressive power of the chosen temporal theory impacts the expressive power of the entire pattern model. In the proposed examples, we consider three possible temporal theories:
  - the *Time Interval Theory* (T^I) which supports the definition of intervals of validity, e.g. $[01-01-2006, 31-05-2006)$;
  - the *Union Interval Theory* (T^U) which supports the definition of valid time periods which are the union of disjoint intervals of time, e.g. $[01-01-2006, 31-01-2006] \cup [01-03-2006, 31-05-2006] \cup [12-06-2006, 21-07-2006)$;
  - the *Periodic Time Expression Theory* (T^P) which supports the definition of periodic valid time information, e.g. 'all monday morning in year 2006'.

  Aspects concerning this topic will be discussed in more detail in Chapter 6, where several choices of temporal theories will be presented (see Section 6.2).

In the following, we present some examples of pattern types. The notation used in presenting such examples will be formalized in Chapter 4. The first two examples present two pattern types for modeling common data mining patterns: market-basket association rules and clusters of products.

Example 1 *Suppose you want to model a pattern type for association rules over a set of sale transactions. The structure schema of the pattern type for association rule can be a tuple with two components: an element called 'head' and an element called 'body'. Each of them is a set of sold items, thus, they can be represented as set of strings. Each rule is mined from a collection of sale transactions and it has associated with two real quality measures, i.e. the confidence and the support (see Section 2.1). Finally, the relationship between an association rule and the raw data, i.e. sales transactions, it represents can be expressed with a quantified logical formula saying that all transactions covered by the association rule (i.e., containing all items appearing in the body or in the head of the rule) are the ones effectively represented by the rule. A periodic validity information can also be associated to each pattern of type association rule, in order to express, for instance, that a a certain rule holds in some specific periods, e.g. during sale, or in the morning, or in summer time, etc. Concluding, we may define the pattern type* `AssociationRule` *as follows.*

name:	AssociationRule
structure:	TUPLE(head:SET(STRING), body:SET(STRING))
data source:	SET(transaction:SET(STRING))
measure:	TUPLE(confidence:REAL, support:REAL)
formula:	$\forall x(x \in \mathsf{s.head} \vee\ x \in \mathsf{s.body} \Rightarrow x \in \mathsf{transaction})$
validity:	T^P

We point out that, in this case, the formula associated with the pattern type represents in an exact way the pattern-data correlation. Indeed, by checking this formula over the data source, for each association rule with a certain head and a certain body, among all transaction data recorded exactly the ones covered by the rule are identified. □

Example 2 *A circular shaped cluster of 2D points can be represented as a 2D circle with a* center *and a* radius. *The center represents the cluster representative*[1] *and it is modeled as a 2D point, i.e. it has two real coordinates CX and CY. The radius is the maximum distance between the representative and other cluster points, i.e., it is a real number.*
The data source schema of the pattern is a set of 2D points, i.e. a set of real coordinates (X, Y). The measure schema models the quality measures associated with each cluster, for example their average intra-cluster distance [JMF99].

name:	Cluster
structure:	TUPLE(center: TUPLE(CX: REAL,CY: REAL), rad: REAL)
data source:	SET(point : TUPLE(X: REAL, Y: REAL))
measure:	AvgIntraClusterDistance: REAL
formula:	$(p.X - \mathsf{s.center.CX})^2 + (\mathsf{p.Y} - \mathsf{s.center.CY})^2 \leq \mathsf{s.rad}^2$
validity:	T^I

Note that, in this case, the formula approximates the mapping between the subset of points in the data source represented by the pattern (whose generic item is represented by the variable p in the formula) and the circle representing the cluster structure (constituting the domain over which variable s of the formula ranges), as shown in Fig. 3.3(a). We outline that in this case the pattern-data mapping expressed by the formula is approximated, since given the 2D circle characterizing the pattern structure, some data source points may exist belonging to the cluster, but laying outside it and, at the same time, some points belonging to the circle may not corresponds to the cluster. In this example, we assume that the pattern validity for circular shaped clusters is represented according to the theory T^I. □

As we have already discussed, the logical model we propose addresses the heterogeneity requirements for patterns. In the following example, we focus on a typical mathematical pattern widely used in statistical applications, i.e., the interpolating line.

[1] A representative is an element, not necessary belonging to the data set from which the cluster has been generated, that characterizes all other elements in the cluster.

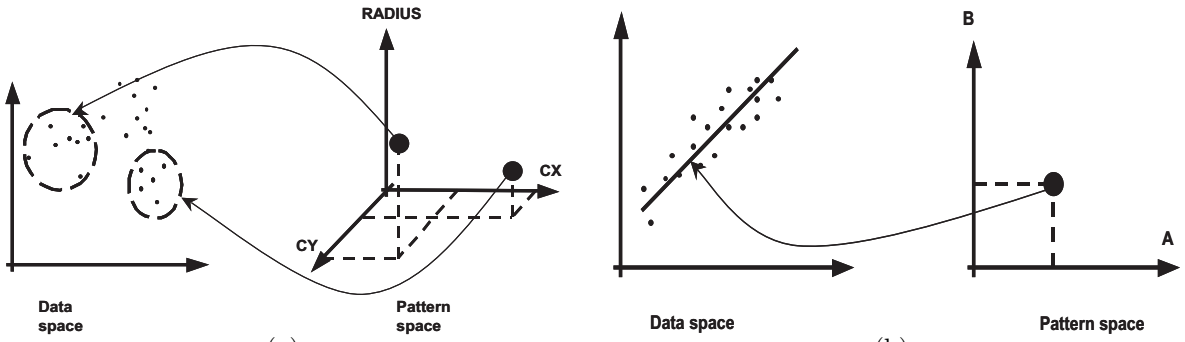


Figure 3.3: Data space and pattern space relationship represented in an approximated way by the formulas of the pattern types *Cluster* (a) and *InterpolatingLine* (b)

Example 3 *A straight line, described by an equation of the form $y = A * x + B$, which interpolates a 2D point distribution plotting the results of some experiments, is an example of mathematical pattern. From an application point of view, it may represents the approximation of a moving object trajectory. According to our proposal, we may define the pattern type, named 'InterpolatingLine', modeling such kind of patterns. Its structure schema includes the two real coefficients required to determine a line in a 2D plan, i.e. the angular coefficient and the offset. The data source schema is the collection of samples, which are the 2D points. The measure schema may consider, for instance, a fitting real value in order to evaluate how well the line approximates the distribution. The formula expressing the correspondence between the pattern and the source data is, obviously, the equation of the line in the 2D plan, having as coefficients the ones appearing in the pattern structure schema. Finally, a validity information to be associated with the interpolation line could be the instant of time at which the line has been generated. Therefore, a suitable temporal theory we may use for the pattern type validity component could be T^I. According to the previous discussion, the pattern type 'InterpolatingLine' can be defined as follows:*

name:	InterpolatingLine
structure:	TUPLE (A: REAL, B: REAL)
data source:	SET(sample: TUPLE(X: INTEGER, Y: REAL))
measure:	fitting: REAL
formula:	$d.Y = s.A * d.X + s.B$
validity:	T^I

We outline that, in this case, the formula - describing the correlation between a generic element d ranging over the data source of the pattern and a generic element s ranging within the pattern structure - is just an approximation of the pattern-data mapping (see Fig. 3.3(b)). Indeed, not all the sample points laying over the line and, at the same time, not all points belonging to the line are sample points. □

The following example, shows a time series pattern, represented according to the approach we propose.

Example 4 *In this example, a list of real values corresponding to samples are taken from a signal periodically (e.g. every second, starting from time 0). Each pattern represents a recurrent wave shape together with the position where it appears within the dataset and its amplitude shift and gain:*

name:	TimeSeries
structure:	TUPLE(curve : ARRAY[1..5](REAL), position : INTEGER, shift : REAL, gain : REAL)
data source:	samples : ARRAY[1..5](REAL)
measure:	similarity : REAL
formula:	$\mathsf{d.samples[s.position + i - 1] = s.shift + s.gain \times s.curve[i], \forall i : 1 \leq i \leq 5}$
validity:	$\mathsf{T^I}$

Measure similarity *expresses how well the wave shape* curve *approximates the source signal in that position. The formula approximatively maps* curve *onto the data space in* position*. Finally, a significant validity information to be associate with the time series could be the interval of time in which the samples have been taken. Thus, a suitable temporal theory we may use for the pattern type validity component could be* T^I*.* □

From the previous examples clearly emerges that, even if all pattern type formulas express the relationship between elements in the pattern space (i.e., pattern structures) and elements in the source data space (i.e., elements belonging to the data source of a pattern), they can be quite different one from the others. Indeed, depending on the information they express and on the nature of the pattern type they are associated with, a suitable formalism, powerful enough, has to be chosen. For instance, the formula used in the definition of Cluster pattern type (see Example 2) is a polynomial equation. Indeed, a linear equation formula would not be suitable to express the equation of a circle. On the other side, the linear polynomial theory is required in the definition of InterpolatingLine pattern type (see Example 3). It seems, therefore, that different formalisms and logical theories could be chosen to express pattern formulas. A similar consideration holds for the pattern validity component. Therefore, the pattern model we will introduce in Chapter 4 will be parametric with respect to the language adopted to express pattern formulas and to the temporal theory chosen to model pattern validity information.

3.3.2 Patterns

Patterns are the knowledge units we are interested in and they are instances of a specific pattern type. Thus, they are record values with *identifiers*, which plays the same role of

OIDs in the object model and which are automatically handled by the PBMS. We call *pid* the unique identifier associated with a pattern instance.

In details, a pattern p, instance of a certain pattern type pt, besides being uniquely identified by its *pid*, is characterized by the following elements:

- A *structure* component that positions the pattern within the pattern space defined by its pattern type, which is an instance of the structure schema of the corresponding pattern type;
- A *data source* component that identifies the specific dataset, conforming the pt's data source schema, the pattern is related to. We remark that, in the case of an a-posteriori pattern, this is the raw dataset from which the pattern has been extracted. Moreover, pattern data sources can be modeled in two different ways: an intensional one and an extensional one. Under an intensional approach, a pattern data source is viewed as the result of a query over certain storage structure containing source data. Thus, a suitable query language have to be used in order to formulate the query producing the dataset we are interested in. On the other side, under an extensional approach, a pattern data source is represented by extensively enumerating its elements. In the proposed framework, we assume the data source is intensionally represented.
- A *measure* component, consisting in a value for the measure schema defined in the pattern type specification, evaluating the quality of the datasource representation achieved by the pattern.
- An instantiated *formula* representing the effective relationship between the data represented by the pattern (i.e., a subset of its datasource) and the pattern structure.
- A *validity period*, which is a formula of the temporal theory $TIME$ chosen in the pattern type definition.

In the following, we present some examples of patterns, instances of pattern types defined in Section 3.3.1. In particular, we present two examples of common data mining patterns (i.e., an instance of the 'AssociationRule' pattern type and an instance of the 'Cluster' pattern type) and a non data mining pattern (i.e., an instance of the 'TimeSeries' pattern type).

Example 5 *Consider the pattern type* `AssociationRule` *defined in Example 1, and suppose that the source dataset is a relational table Sales which stores data related to the sales transactions in a sport shop: Sales(transactionId, article, quantity), univocally identified by the attribute transactionId. Using an extended SQL syntax to denote the dataset and a*

natural language specification for a periodic time expression, an example of an instance of `AssociationRule` *is:*

pid:	*115*
structure:	$< head = \{'Boots'\}, body = \{'Socks', 'Hat'\} >$
data source:	*'SELECT SETOF(article) AS transaction FROM sales GROUP BY transactionId'*
measure:	$< confidence = 0.75, support = 0.55 >$
formula:	$\forall x((x \in \{`Boots'\} \vee x \in \{`Socks', 'Hat'\}) \Rightarrow x \in transaction)$
validity:	*'every year during winter time'*

In the formula component, variable x *ranges over the source space; the values given to head and body within the structure are used to bind variables* head *and* body *in the formula of pattern type* `AssociationRule`*. We outline that, we have expressed the periodic expression stating when the rule is valid in natural language. In a more formal way, we should use a suitable formula of the temporal theory* T^P *to express such an information. For more details about this theory and the proposed formalism we refer interested readers to Chapter 6.* □

Example 6 *Consider the pattern type* `Cluster` *defined in Example 2, a possible pattern istance of this type is the following:*

pid:	337
structure:	TUPLE(center:TUPLE(CX:2,CY:3), rad:4,2)
data source:	users(X,Y)
measure:	AvgIntraClusterDistance: 0.9
formula:	$(\mathsf{users.X} - 2)^2 + (\mathsf{users.Y} - 3)^2 \leq (4, 2)^2$
validity:	*[01-03-2006, 31-03-2006]*

In the pattern 337*,* users *is a relational view concerning observations about users behavior during their navigation in a certain Web site. It may contain, for example, for each user, the age and the total number of her navigation sessions during a day. The pattern formula is obtained by instantiating the pattern type formula with structure component values and data source component variable names of the pattern instance. It thus provides the exact equation used to identify the approximated set of points represented by the specific cluster at hand. Finally, the validity period associated with pattern* 337 *corresponds to the time period to which collected Web data refer to.* □

Example 7 *Consider the pattern type* `TimeSeries` *presented in Example 4. A possible pattern, extracted from a dataset which records the hourly-detected levels of the* Colorado river*, is as follows:*

pid:	*456*
structure:	$< curve = [y = 0, y = 0.8, y = 1, y = 0.8, y = 0],$ $position = 12, shift = 2.0, gain = 1.5 >$
data source:	*'colorado.txt'*
measure:	$< similarity = 0.83 >$
formula:	$\{samples[12] = 2.0, samples[13] = 3.2, samples[14] = 3.5,$ $samples[15] = 3.2, samples[16] = 2.0\}$
validity:	$[time_{fs}, time_{ls}]$

$time_{fs}$ and $time_{ls}$ represent the hour of the first music sample taken and the hour of the last sample taken, respectively. □

3.3.3 Classes

A *class* is a collection of semantically related patterns and constitutes the key concept in defining a pattern query language. A class is defined for a given pattern type and contains only patterns of that type. They are fundamental in the query processing within the pattern based management system. Indeed, pattern queries are posed over pattern classes and returns other collection of patterns.

Example 8 *The* Apriori *algorithm described in [AS94] could be used to generate patterns of type* AssociationRule*, defined in Example1, from a relational data set containing sale transactions collected in a supermarket. The relation is named* SalesTRS *and has schema (trID, item1, item2, ..., itemN). All the generated patterns could be inserted in a class called* SaleRules *defined for* AssociationRule*. The collection of patterns associated with the class can be later extended to include also rules generated from a different dataset (e.g* SalesTRS_2 *with the same schema of* SalesTRS*) representing, for instance, the sales transaction recorded in a different store.* □

3.3.4 Hierarchical Relationships Between Pattern Types

The expressivity of the model for representing patterns can be improved by considering hierarchies that may exist between patterns and pattern types [RBC+03]; in this way, it is possible to address the modeling requirement concerning complex pattern definition (see Section 3.1, requirement 3).

At least two different hierarchical relationships between pattern types can be useful:

- The *composition* relationship provides the ability to define complex pattern types (and, hence, patterns) by specifying that instances the structure of a certain pattern

type may refer to some (sub-)components which are of a different pattern types. From a conceptual point of view it builds a *part-of* hierarchy between the two pattern types.

- The *refinement* relationship supports the definition of patterns representing other patterns. In particular, the refinement relationship describes the relationship that may exist between a pattern instance of a certain pattern type and other patterns (instances of different pattern types) that may appear in its data source. Intuitively, the refinement relationship addresses the possibility to specify that a pattern of a certain type is obtained by mining other existing patterns.

Another oimportant relationship is given by the definition of an *is-a* hierarchy between pattern types and patterns. We, however, do not deal with this issue in the context of the proposed framework, in order to focus on more specific pattern features.

Example 9 *Consider the pattern type* `AssociationRule` *(see Example 1). The user may be interested in modeling mono-dimensional clusters of association rules. Each cluster can be viewed as a group of association rules sharing some properties and represented by a representative rule. The structure of the pattern type modeling clusters of rules consists of a single element of type association rule. Thus, it exploits the composition relationship. Moreover, since a single cluster of rules represents an association rule dataset (i.e., the datasource is a set of patterns of type* `AssociationRule`*), a refinement relationship exists between this new pattern type and the pattern type* `AssociationRule`*. A possible definition of the pattern type* `ClusterOfRules` *is the following:*

name:	ClusterOfRules
structure:	representative: AssociationRule
data source:	SET(rule: AssociationRule)
measure:	TUPLE(deviationOnConfidence: REAL, deviationOnSupport: REAL)
formula:	rule.ss.head=representative.ss.head
validity:	T^U

In the previous definition, a standard dot notation is adopted to denote pattern type components. Moreover, the temporal theory T^U *has been adopted to represent validity information associated with patterns, instances of this pattern type. The validity period of a cluster of rules can be interpreted, for instance, as the union of the validity periods associated with each rule in the cluster.* □

3.3.5 Pattern Extraction and Measure Evaluation

As already discussed, we want to deal with both a-priori patterns and a-posteriori patterns. Concerning a-posteriori patterns, there could be many different mining algorithms

to generate them. For instance, concerning association rules, there are many different implementations of the Apriori algorithm - first proposed by Agrawal et Al. [AS94]. Moreover, there are some other algorithms exploiting different paradigms to extract association rules, such as the one proposed in [HPY00]. For a survey on association rule mining algorithms we refer the reader to [HGN00]. In order to be able to trace the information concerning the mining process used to generate certain patterns, information concerning the used mining methods is modeled and stored in the PBMS. In particular, for each pattern type we formally define each possible extraction process as a *mining function* which takes as input a data source, applies a certain computation to it, and returns a set of patterns, instances of the pattern type.

We remark that, according to what stated in Section 3.3.2, we associate with each pattern a temporal information concerning its validity and we assume a possible temporal validity information available also over source data. Starting from this point, the mining function defined for a specific pattern type may take advantage in exploiting the temporal information associated with surce data in order to determine the validity information for extracted patterns. Nevertheless, this topic is only partially addressed in pattern management literature.Indeed, with the advent and the consolidation of temporal databases technology [TCG+93, BJW00], several approaches have been proposed for dealing with source data having temporal validity information associated with in the context of data mining and machine learning, especially dealing with temporal association rule extraction and sequence patterns [RS02]. However, most of these cases concern time series analysis [HD04, CSD98] or the issue of extracting relationships between events belonging to different temporal sequences [BWJL98]. Unfortunately, patterns resulting by applying these approaches usually are not associated with a validity time information. Concerning information retrieval, the previous topic has not been considered, since in general information retrieval applications do not produce kinds of patterns (e.g., document vectors) having a sensible validity information which depends on the possible temporal information associated with underlying data (i.e., text documents). A similar remark holds for mathematical patterns (such as interpolating lines). Restricting our attention to association rule data mining, temporal data mining is an active research fields and several approaches to mine rules from transactions associated with a temporal information exist [LNWJ03].[2] In general, all the proposals consider a timestamp associated with transactional data and they produce association rules with periodicity information [LNWJ03] or with interval validity information [VV05, AR00] or both [CP00].

In conclusion, once mined by using a suitable (temporal) mining function, patterns are inserted in the system and, since we want to maintain patterns and underlying data synchronized, we need to describe how pattern measures are computed with respect to a

[2]We outline that, a survey of temporal data mining research work is out of the scope of this thesis, here we simply recall several interesting data mining approaches which have been recently proposed in literature addressing the temporal rule discovery problem.

certain raw dataset. The function computing the values for measures of patterns with a certain pattern type over a certain dataset is called *measure function* and, as in the case of the mining function, it is stored within the PBMS.

We remark that the measure function is particularly interesting when dealing with manipulation issues, in order to re-compute pattern measures and to determine whether the quality of the representation of a certain dataset achieved by a pattern changes over the time accordingly to changes occurred in raw data.

3.4 Pattern Languages

As we have already discussed in Section 3.1 (requirements 6 and 7), in order to address manipulation and querying issues, our PBMS has to provide at least two distinct languages: a *Pattern Query Language (PQL)* and a *Pattern Manipulation Language (PML)*. In the remainder of this section, we briefly point out which are the basic issues in pattern manipulation (Section 3.4.1) and querying (Section 3.4.2) that we plan to support in our framework. The formal definition of languages for patterns will be presented in Chapter 5.

3.4.1 Pattern Manipulation

The following manipulation features have to be provided by the Pattern Manipulation Language.

Mining support Giving specific support for patterns resulting from a mining process is a fundamental issue for all PBMSs. According to what stated in Section 2.1, patterns (possibly of the same type) can be the result of several different mining algorithms. Therefore, the capability of supporting a-posteriori patterns mined by using different mining functions is a key pattern manipulation aspect any PBMS has to considered. Moreover, tracing the information concerning the mining function used to create certain patterns could be useful for other manipulation purposes concerning pattern care and update (see below).

Example 10 *Consider association rules concerning food sales. The rules generated by a run of the* Apriori *algorithm [AS94] with minimum support and confidence thresholds fixed at* 0.5 *are a-posteriori rules which once mined have to be inserted in the PBMS. Similarly, association rules generated by applying the* FP-growth *algorithm [HPY00] are a posteriori patterns which need to be inserted in the PBMS.* □

A-priori pattern support As already discussed, a user may be interested not only in patterns generated from source data by using some mining tools (a-posteriori patterns) but also in patterns she knows in advance (a-priori patterns) and wants to use in combination with other existing patterns or to check their relevance with respect to some data source. Therefore, the pattern insertion from scratch in the system is a manipulation operation to be considered.

Example 11 *Consider association rules concerning food sales. It may happen that the user may have a previous experience in food sale and she may know that the rule 'Egg, sugar → flour' holds; in this case it is an a-priori rule (i.e., it has not been mined from a specific raw dataset) which has to be inserted in the PBMS.* □

Update support Since source data change with high frequency, an important issue for a pattern management system consists in determining whether existing patterns, after a certain time, still represent the data source from which they have been generated or another existing source data set and, possibly, being able to change pattern information when the quality of the representation changes. Thus, issues concerning pattern synchronization and validation with respect to a given data set are very important. In particular, sometimes, given a pattern, it is important to be able to recompute its associated measures against a certain dataset - which may either be the one over which the pattern has been generated or another one - and, according to the results, being able to update the old pattern (i.e., to synchronize it with the raw dataset) or to modify its validity information (i.e., to validate it).

Example 12 *Consider the a-priori association rule concerning food sales of the Example 11. The user may be interested in testing whether this rule is verified - and with what measure values - by data stored in a certain dataset D where she collects raw data concerning sale transactions. Thus, she wants to synchronize the pattern with source data in D.* □

Class operation support Since, according to what previously stated (see Section 3.3.3), patterns are collected into classes in order to be queried, manipulation operations supporting the insertion and the removal of a pattern into or from a class defined for the proper pattern type are required.

Example 13 *Consider the association rule of the Example 5 having pid* = 115 *and assume the classes 'winter_AR', concerning all rules which hold in the winter time, and 'sales_AR', collecting all rules extracted from 'Sales' dataset, have been defined in the system. A possible manipulation would consist in inserting the specific rule in both classes 'winter_AR' and 'sales_AR'. When the user is no more interested in analyzing winter time association rules, he may want to remove association rules* 115 *from the class 'winter_AR', but not from 'sales_AR' class.* □

Pattern removal support When the user is no more interested at all in a pattern, besides removing it from one or more classes it belongs to in order to prevent their querying, he meeds to be able to remove the pattern from the system. Therefore, a pattern deletion operation is required. However, in a pattern management system, deletion issues have to be addressed in a special way in order to not create inconsistency in the system. Indeed, according to the modeling choices supporting advanced pattern requirements (such as, pattern hierarchies) and class management (see Section 3.1), aspects concerning pattern hierarchies and multi-class membership have to be taken into account in designing deletion features provided by the PBMS. In particular, pattern hierarchies have to be preserved, thus, whenever a pattern shares a hierarchical relationship with other patterns it cannot be removed from the system. Thus, different deletion operations with different behaviors should be provided, as shown by the following example.

Example 14 *Consider again the association rule of the Example 5 having pid* $= 115$*. According to the situation depicted in Example 13, such rule exists in the system and it is a member of the class 'sales_AR'. By invoking a deletion operation under a conservative approach (i.e., a restricted deletion) will not produce any effect, since the rule still belongs to the 'sales_AR' class. On the other side, by adopting an extended approach, the deletion operation will remove the rule from the class 'sales_AR' and from the system.* □

3.4.2 Pattern Querying Issues

In the context of the proposed framework, we want to cope with, the following querying issues.

Pattern Retrieval The capability to retrieve patterns according to some user specified condition is a fundamental issue a pattern query language has to address. As shown in the following example, query conditions may involve any possible pattern component.

Example 15 *Consider the* `AssociationRule` *pattern type (see Example 1) and assume C is a class defined for it. The following are examples of simple queries the user may be interested in:*

- *find all association rules belonging to C having support between* 0.3 *and* 0.75*;*
- *find all association rules belonging to C that are valid on March 12, 2006;*
- *find all association rules belonging to C whose body contains 'Boots' or whose confidence is greater than* 0.7*;*

- *find all association rules belonging to C having at least 2 items in the body;*
- *find all association rules belonging to C having a certain degree of similarity with a another fixed pattern.* □

Concerning pattern similarity, we recall that, as we have already discussed in Chapter 2, an important characteristic of a pattern query language is the ability to check similarity conditions based on pattern structure and measures (see Section 2.2.4) as shown, for example, in the last query in the above list. However, as pointed out in Section 3.1, we do not cope with this issue in the context of this thesis. Indeed, a similarity-based pattern retrieval would allow users to analyze the differences existing among the underlying data sets and it would be very useful whenever she wants to measure differences of patterns describing evolving data or data extracted from different sources. For instance, in monitoring monthly sales of a supermarket or in analyzing differences of data characteristics across several sets of data (customers transactions, reactions to chemical/biological substances), whenever the pattern similarity results high, there is no need to perform a thorough (and costly) analysis on actual data. However, the question *How is it possible to define pattern similarities?* is an hot topic for researchers and there is no general agreement on a possible general pattern similarity function. In general, with pattern similarity function we mean a function taking two patterns, possibly of the same type, and returning a value between 0 and 1 representing how similar the two patterns are. Even if various specific approaches have been defined for specific types of patterns, only two general approaches exist for pattern similarity independently of the considered pattern type. The first approach has been defined by Ganti et Al. [GGR99]. The second one has been defined in the context of the PANDA project [BCN+04]. We point out that the specific topic concerning pattern similarity definition is a complex research issue and it is out of the scope of the activities of this thesis.

Pattern combination A fundamental issue a pattern query language has to address concerns the capability to combine together patterns, possibly of different types, producing new patterns. An important application of this kind of querying operation is shown in the following example.

Example 16 *Consider two classes c_1 and c_2 defined for the pattern type* AssociationRules *of the Example 1. Suppose we want to generate new association rules by transitive closure, i.e., informally, given two rules $A \rightarrow B \in c_1$ and $B \rightarrow C \in c_2$, we want to generate a new rule like $A \rightarrow C$. The resulting rules are the combination, i.e. the join, of each pair of association rules, where the first input rule belongs to c_1 and the second rule belongs to c_2 such that the body of the first pattern matches with the head of the second pattern.* □

Explore Pattern Hierarchies According to what stated in Section 3.3.4, patterns can share hierarchical relationships. Thus, an important issue when querying hierarchical patterns concerns the capability to move up and down their hierarchical relationships. Therefore, dedicated pattern query operators to explore the composition and refinement hierarchies are required (see Example 17).

Example 17 *Consider the refinement relationship existing between patterns of type* ClusterOfRules *and patterns of type* AssociationRule *according to the definition given in the Example 9. Let C be a class of type* ClusterOfRules *containing rule clusters. It would be useful to retrieve the set of association rules from which the clusters in class C have been mined. This correspond to move down in the refinement relationship. On the other side, given a class C' of* AssociationRule *patterns, it would be useful to retrieve the patterns of type* ClusterOfRules *refining at least one rule in C'. Similarly, it would be useful to retrieve the set of association rules which are representative of patterns in C.* □

Cross-over queries Besides operators to retrieve patterns, in a knowledge management context it is also important to support operations binding patterns with raw data. Such operations are usually known as *cross-over queries* since for their execution two different systems - the PBMS and the system managing source data - have to be used. A typical example of query involving a cross-over process is when the user wants to retrieve elements represented by a pattern, i.e., elements in the data source of a certain pattern (see Example18). However, more complex types of cross-over queries may be useful to perform a deeper knowledge analysis identifying portions of the source data space effectively covered by a certain pattern or vice-versa which patterns in the pattern space represents a certain dataset (see Example 18).

Example 18 *Consider the association rule of the Example 5 having pid* $= 115$ *and assume to be interested in determining all articles belonging to its data source. This query is posed to the PBMS, which has to query the DBMS in order to solve it. In particular, locally to the DBMS the query defining the data source of the association rule* 115 *(i.e., 'SELECT SETOF(article) AS transaction FROM sales GROUP BY transactionId'), has to be execute, then, the obtained result is passed to the PBMS which outputs the response to the user.*

Other examples of cross-over queries are:

- *Determine whether a certain association rule is suitable to represent a certain data set D;*
- *Starting from D, determine the association rules in the class winter_AR representing elements in D with quality measures greater then some specified thresholds.* □

Chapter 4

$\mathcal{EPM}$: an Extended Pattern Model

This chapter is dedicated to the formal definition of the logical models for data and pattern representation. Concerning the data model, we assume raw data are modeled as (object) relational data, according to the model presented in [AB95]. Concerning the pattern model, we formalize the one informally introduced in Chapter 3. After presenting the basic concepts of pattern types, patterns, and classes, hierarchical relationships among patterns will be defined and issues concerning the mining process and the quality measure evaluation will be discussed, leading to the formalization of mining and measure functions.

As already discussed in Section 3.1, a fundamental issue in modeling patterns is the ability to identify the subset of raw data effectively represented by a pattern. To this end, the proposed pattern model equips each pattern with a logical constraint formula, expressing (in a possibly approximate way) the semantics of the pattern with respect to raw data. In general, several logical constraint theories can be used to express such formula. For instance, linear or polynomial constraint theories over $\mathbb{R}$ can be exploited. However, simpler, but less expressive, theories (such as the gap-order constraint theory) can also be used. The choice of a certain logical theory for pattern formulas impact the expressive power and the overall complexity of the whole framework for patterns. Thus, in order to make the proposed framework effectively usable, a trade-off between expressive power and complexity has to be reached. Theoretical issues concerning this topic will be discussed in Chapter 6.

The models we present support the modeling of validity temporal information. Temporal information is assigned to each tuple of raw data and to each pattern. In general, valid time can be represented in several ways. For example, it can be modeled as a time instant, a time interval or as a periodicity constraint. Since the choice of a specific temporal theory depends on the reference application domain and on data semantics, the proposed models are parametric with respect to the chosen temporal theories. For the sake of simplicity, we

however deal with single granularity time management. The proposed models can however be easily extended to cope with multigranular temporal information [BJW00]. Possible alternatives for temporal theories to be used in the pattern model will be presented in Chapter 6 Section 6.2.2. Also in this case, the choice of a specific theory will lead to specific results in terms of expressive power and the complexity of the resulting framework. Such issues will be discussed in Chapter 6.

Modeling valid time information for patterns gives rise to the definition of two distinct notions of pattern validity: a temporal one and a semantic one. *Temporal validity* specifies that a pattern is assumed to be reliable in a certain period of time based on some application dependent requirement. On the other side, a pattern is *semantic validity* with respect to a data source if it represents such data source with a certain quality level.

The remainder of this chapter is organized as follows. Section 4.1 is dedicated to the logical model for raw data, whereas Section 4.2 is focused on the logical model for non hierarchical patterns. The model is called *Extended Pattern Model* ($\mathcal{EPM}$). We call it *Extended Pattern Model* ($\mathcal{EPM}$) in order to point out that it extends existing proposals to cope with additional features, supporting the requirements listed in Section 3.1. In Section 4.3 $\mathcal{EPM}$ is extended in order to model hierarchical patterns. Section 4.4 contains a digression concerning temporal mining functions. Finally, pattern validity and safety issues are discussed in Section 4.4.

4.1 Source Data Model

In what follows, we introduce a formal logical model for representing source data. The model we are going to present is based on the one introduced by Abiteboul and Beeri in [AB95] for representing complex values data, extended with temporal information.[1] To this purpose, we first present the types allowed by the model and the corresponding domains. Then, we introduce database schemas and instances used for modeling source data.

4.1.1 Types

Since we assume to model source data by means of a complex value model, both *atomic* and *complex* types are allowed for modeling complex data components.

The set of *atomic data types* (AT) contains traditional atomic data types, such as Integer, Real, Boolean, Character, String, Timestamp, and Date belong.

[1] We remark that the choice of presenting a complex values model instead of a complex objects one is only due to simplification purposes.

Definition 3 (*Atomic types*) The set of atomic type $\mathcal{AT}$ is defined as $\mathcal{AT} = \{Integer, Real, Boolean, Character, String, Timestamp, Date\}$. □

It is important to outline that we assume atomic types $Timestamp$ and $Date$ are not used to codify pattern validity information; rather they are built-in basic types useful in specifying additional temporal raw data components. In the following, whenever no ambiguity arises, we identify an atomic type with its name (e.g. 'Integer' is the name of the type Integer which domain is the set of all possible integer numbers).

Complex data types are obtained by combining atomic types through the usage of type constructors SET and TUPLE[2] and by introducing the ability to assign names to defined types. We assume that names are selected from a set of type name labels $\mathcal{L}$. The overall type set is therefore defined as follows.

Definition 4 (*Types*) Let $\mathcal{AT}$ be the set of atomic types introduced in Def 3. Let $\mathcal{L}$ be a set of type name labels. The set of types $\mathcal{T}$ is inductively defined as follows:

1. $\top$ is a type and it is called 'root' type;
2. $\forall t_i \in \mathcal{AT}$, $t_i \in \mathcal{T}$ and it is an unnamed type;
3. if $t \in \mathcal{T}$, then $SET(t) \in \mathcal{T}$ is an unnamed type;
4. if $t_1, t_2, ..., t_n \in \mathcal{T}$, $n \geq 1$ and $a, a_1, ..., a_n \in \mathcal{L}$, then $TUPLE(a_1 : t_1, ..., a_n : t_n) \in \mathcal{T}$ and it is an unnamed type;
5. if $t \in \mathcal{T}$ is an unnamed type (see above), $A \in \mathcal{L}$, and $\nexists t' \in \mathcal{T}$ such that $A : t' \in \mathcal{T}$, then $A : t \in \mathcal{T}$ and it is a named type;
6. if $A : t \in \mathcal{T}$, then $A \in \mathcal{T}$ and it is a named type. □

Item 6 specifies that the identifier of a named type can be used to identify the type itself in type definition. Given a named type $a : TUPLE(a_1 : t_1, ..., a_i : t_i, ..., a_n : t_n)$, in the following $a.i$ denotes the type of the i-th components i.e., t_i.

4.1.2 Domains for Data Types

In the following, domains for the types previously introduced are presented. For each atomic type, we assume an infinite, countable set of values as its domain. The set of

[2]We remark that the given definition can be easily extended in order to deal with other constructors such as, for instance, BAG, LIST, and ARRAY.

values for an atomic type $t \in \mathcal{AT}$ is denoted by $\|t\|$. Domains for complex types are then inductively computed starting from the domains of their components. More precisely, the domain of set types is defined as the powerset of the domains of the constituent types whereas the domain of tuple types is defined as the product of their constituent types. The set of allowed values for a type $t \in \mathcal{T}$, i.e., its domain, is denoted by $dom(t)$.

Definition 5 Let $t \in \mathcal{T}$. $dom(t)$ is inductively defined as follows:

- $\perp \in dom(t)$, $\perp$ denotes a null value, which we assume can be specified for each domain;
- if $t \in \mathcal{AT}$, then $dom(t) = ||t||$;
- if $t = A : t'$, then $dom(t) = dom(A : t') = dom(A)$;
- if t is a set type (i.e. $t = SET(t')$), then $dom(t) = \{\{\}\} \cup \{\{v_1, ..., v_j\} | j \geq 1 v_i \in dom(t'), i = 1, ..., j\}$;
- if t is a tuple type (i.e. $t = TUPLE(A_1 : t_1, ..., A_k : t_k)$), then $dom(t) = \{< A_1 : v_1, ..., A_k : v_k > | v_i \in dom(t_i), i = 1, ..., k\}$. □

In the remainder of this paper, we use the notation $e :: t$ to express in a concise way that expression e has type t. Concerning the tuple type, we use the dot notation to access tuple components, i.e. attributes. Given a tuple r, in the following $r.i$ denotes the value of the i-th component of r.

We also notice that, according to the proposed definition, domains of named types, having distinct names but referring the same unnamed type, coincide. Thus, names are just labels for referring specific type definition.

4.1.3 Database

Based on the types and domains introduced in the previous sections, we can now introduce the notions of *database schema* and *database instance*. Since we assume that raw data can be associated to temporal validity information, we slightly extend the formal approach of [AB95] to cope with valid time. To this purpose, we consider a set $\mathcal{RN}$ of relation name strings and a temporal theory $\mathcal{V}$, and we define relations as sets of complex values tuples including a temporal attribute, modeling valid time information. Here, with temporal theory we mean a first-order theory whose constraints represent temporal information. The domains for temporal attributes are all the formulas that can be expressed in $\mathcal{V}$. Examples of logical theories that can be used to model valid time information will be

presented in more details in Chapter 6. We refer interested readers to [BFGM03, BJW00] for technical aspects and details concerning temporal data models.

Definition 6 (*Database Schema*) Let $\mathcal{V}$ be a logical temporal theory. Let $\mathcal{RN}$ be a set of relation names. 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: (i) $\widehat{D_i} \in \mathcal{T}$, $i = 1, ..., k$; (ii) $\widehat{R_i} \in \mathcal{RN}$, $i = 1, ..., n$; (iii) $T_i : SET(TUPLE(S_1^i, ..., S_m^i, \mathcal{V})) \in \mathcal{T}$, $i = 1, ..., n$, $S_j^i \in \mathcal{T}$, $j = 1, ..., m$, and T_i is defined upon data types $\widehat{D_1}, \ldots, \widehat{D_k}$. □

Definition 7 (*Database*) Let $\widehat{DB} = \langle[\widehat{D_1}, \ldots, \widehat{D_k}], [\widehat{R_1} :: T_1, \ldots, \widehat{R_n} :: T_n]\rangle$ be a database schema. An instance of $\widehat{DB}$, i.e., a *database* over $\widehat{DB}$, is a pair

$$DB = \langle[D_1, \ldots, D_k], [R_1, \ldots, R_n]\rangle$$

where D_i's are domains for data types $\widehat{D_i}$'s and R_i's are values for data types T_i's. □

According to Def. 7, each R_i has the following form:

$$R_i = \{r_j | r_j =< v_1, ..., v_n, w >\}$$

where, v_i is a legal value for type S_i, $i = 1, ..., n$, and w is a legal formula for the chosen temporal theory $\mathcal{V}$.

As discussed in Chapter 3, in the remainder of this thesis we assume that patterns are extracted or in general related to source databases represented according to Def. 7. Such databases can be local to the PBMS or remote, accessible through the network by means of a URI (Universal Resource Identifier). To simplify the presentation of the pattern model, in the following we however assume that raw data belong to a single *Universal Database (UDB)*. The Universal Database constitutes a sort of wrapping of many different compliant data repositories containing source data.

4.2 Extended Pattern Model

In what follows, we formally present the *Extended Pattern Model* ($\mathcal{EPM}$) , informally introduced in Section 3.3. According to what informally stated in Chapter 3, $\mathcal{EPM}$ is an object-relational model which relies on three basic concepts: *pattern type*, *pattern*, and *class*. The key notion is the pattern, that formally codifies the knowledge information unit

we want to represent. Each pattern is characterized by a certain type which defines its syntactic structure. Moreover, the notion of pattern class is introduced to model collections of patterns of the same pattern type. Classes constitute the fundamental unit over which queries can be executed. $\mathcal{EPM}$ addresses all the requirements identified in Chapter 3 and, in particular, it includes temporal information management and handles hierarchies between patterns, which will be introduced in Section 4.3. Moreover, it provides the usage of a query over the Universal Database to identify the source database over which patterns have been extracted or in general are related.

Since different application contexts may lead to different requirements, the definition of $\mathcal{EPM}$ is highly parametric. More precisely, $\mathcal{EPM}$ is parametric with respect to: (i) the query language $\mathcal{Q}$ used to specify the pattern data source; (ii) the logical theory $\mathcal{F}$ used to express the pattern formula; (iii) the temporal theory $\mathcal{V}$ used to model the validity information associated with patterns. For this reason, the Extended Pattern Model is usually denoted by $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, to point out the chosen languages. The notation $\mathcal{EPM}$ is however used for the sake of simplicity when no ambiguity arises.

Examples of logical theories that could be used for $\mathcal{F}$ are: the *equality/disequality constraint theory over* $\mathbb{Z}$ or the *polynomial constraint theory over* $\mathbb{R}$. On the other hand, concerning $\mathcal{V}$ we may choose, for instance, the *time interval theory over* $\mathbb{Z}$. Possible options for $\mathcal{F}$ and $\mathcal{V}$ will be presented in Section 6.2 of Chapter 6.

On the other side, the choice for the query language $\mathcal{Q}$ depends on the data model used for the Universal Database. For instance, assuming raw data in the Universal Database are simply relational, then relational algebra or tuple calculus can be used as $\mathcal{Q}$. Differently, whenever the Universal Database relies on a comple value data model, as the one introduced in Section 4.1, $\mathcal{Q}$ is a complex value language like the one proposed in [AB95]. For more details concerning $\mathcal{Q}$ options, we refer the reader to Chapter 6.

In the rest of this chapter, we assume that:

- $\mathcal{Q}$ identifies a query language for the underlying Universal Database, we call $\mathcal{Q}$-query a generic query expressed using $\mathcal{Q}$;
- $\mathcal{F}$ identifies a logical theory with constraints for complex values;
- $\mathcal{V}$ identifies a theory modeling temporal information.

In the following, the notion of (basic) pattern type is first formally defined (Section 4.2.1). Then, domains for pattern types are introduced and used to formalize the notion of patterns (Section 4.2.2) and classes (Section 4.2.3). The formalization of concepts concerning pattern management issues is presented in Section 4.3.4.

4.2.1 Pattern Type

Within $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, 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. Indeed, as we will see in Section 4.2.2, each pattern is an *instance* of a specific pattern type. In the following, we introduce the formal definition of a Pattern Type by taking into the account modeling requirements and the intuitive concept of pattern type introduced in Chapter 3. We start by introducing the notion of *Basic Pattern Type*, that does not take into account pattern hierarchies (Section 4.2.1). The concept of *Hierarchical Pattern Type* will be formalized in Section 4.3.

Among the pattern type components, the definition of the following ones is parametric with respect to the choices of $\mathcal{Q}$, $\mathcal{F}$, and $\mathcal{V}$:

- *Data source.* The pattern data source represents the overall data set the pattern is related to. Formally, it corresponds to a query over the Universal Database (see Section 4.1.3). We notice that, since Basic Pattern Type are not hierarchically defined, in general, it corresponds to a complex value query. In defining a pattern type, we just specify the type of the query result. When dealing with a-posteriori patterns, it corresponds to the raw data set from which a pattern has been mined. For a-priori patterns, it corresponds to the data set the user knows is represented by the pattern.
- *Formula.* The pattern formula is a formula expressed using the logical theory $\mathcal{F}$, describing in an intensional, and possibly approximated way, the subset of data effectively represented by a pattern. The formula may depend on information contained in the pattern structure. Since such information is not known at pattern type definition time, in the pattern type we just specify a formula whose free variables range over the pattern data set and over the pattern structure.
- *Validity period.* The pattern validity period is a formula of the temporal theory $\mathcal{V}$. In the pattern type, we just specify the reference temporal theory, i.e., $\mathcal{V}$.

Formally, a pattern type can be defined as a named complex type, defined upon types introduced in Section 4.1.1 and special types related to the components discussed above.

Definition 8 (*Basic pattern type*) Let $\mathcal{BPTN}$ be a set of labels. Let $\mathcal{Q}$ be a query language for the Universal Database, i.e. a query language for complex value databases. Let $\mathcal{F}$ be a logical theory with constraints for complex values. Let $\mathcal{V}$ be a temporal theory. Let $ext(\mathcal{F})$ be the set of formulas that can be expressed in $\mathcal{F}$. Let $t_s, t_d, t_m \in \mathcal{T}$, $Name \in \mathcal{L}$, and $t_f \in ext(\mathcal{F})$. A basic pattern type (*bpt*) for $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ is a named type of the form

$$Name : TUPLE(s : t_s, d : t_d, m : t_m, f : t_f, v : \mathcal{V})$$

such that:

- t_d is a type of the form $SET(TUPLE(t_1, ..., t_n))$, $t_1, ..., t_n \in \mathcal{T}$, corresponding to the schema of a source data set which can be obtained as output of a $\mathcal{Q}$-query;
- t_m is a type of the form $TUPLE(t_1, ..., t_k)$, $t_1, ..., t_k \in \mathcal{AT}$, defining the type of measures which quantify the quality of the source data achieved by a pattern instance of the pattern type being defined, often $t_1, ..., t_k$ are atomic types representing numerical values;
- $t_f \equiv \varphi(\tau, \psi)$, where φ is a well formed formula of the theory $\mathcal{F}$, and τ, ψ are the only free variables in φ. τ has type $TUPLE(t_1, ..., t_n)$, thus, implicitly, ranges over the domain of data source elements, whereas ψ has type t_s (thus it ranges over the pattern structure domain).

We denote with $\mathcal{BPT}$ the set of all possible basic pattern types. □

Notice that the formula component used in the pattern type definition expresses (possible in an approximated way) the correlation between elements in the pattern structure and elements in its data source. As we will see later, the variable representing the pattern structure will be instantiated inside patterns, istances of the given pattern type. The following examples show how some typical data mining patterns can be represented in $\mathcal{EPM}$

Example 19 *Let $\mathcal{Q}$ be a query language for complex values, as the one presented in [AB95], $\mathcal{FO}$ be a first order logical theory for complex values, and $\mathcal{V}$ be the temporal theory of intervals over $\mathbb{Z}$ (T^I). The basic pattern type modeling Association Rule, informally introduced in Example 1, can be modeled in $\mathcal{EPM}(\mathcal{Q}, \mathcal{FO}, \mathsf{T}^I)$ as a named tuple of the form:* $\mathsf{AssociationRule} : \mathsf{TUPLE}(\mathsf{s} : \mathsf{t_s}, \mathsf{d} : \mathsf{t_d}, \mathsf{m} : \mathsf{t_m}, \mathsf{f} : \mathsf{t_f}, \mathsf{v} : \mathsf{T^I})$*, where:*

$t_s \equiv$ TUPLE(head:SET(STRING), body:SET(STRING))
$t_d \equiv$ SET(transaction:SET(STRING))
$t_m \equiv$ TUPLE(confidence:REAL, support:REAL)
$t_f \equiv \forall x :: String(x \in \mathsf{s} :: \mathsf{t_s}.\mathsf{head} \vee x \in \mathsf{s} :: \mathsf{t_s}.\mathsf{body} \Rightarrow x \in \mathsf{trs} :: \mathsf{SET}(\mathsf{String}))$

According to what stated in the Example 1, the structure schema is a tuple with two components: the head and the body, which are sets of strings representing products. The data source schema specifies that association rules are constructed from a set of sale transactions, each defined as a set of strings, representing products. The measure schema includes two common real measures to assess the relevance of a rule: its confidence (what percentage of transactions including the head also include the body) and its support (what percentage of the whole set of transactions include both the head and the body). The formula represents

(exactly, in this case) the pattern/data set relationship by associating each rule with the set of transactions which support it. The only free variables involved in the formula component are `s` *and* `trs`*.* `s` *has type STRING and it ranges over elements in dom(t_s.head) and dom(t_s.body).* `trs` *has type SET(STRING) and it ranges over elements in dom(t_d). Finally, the temporal theory T^I has been chosen for the validity period associated with the pattern type (see Section 6.2.2 for more details about T^I). In this way, each pattern of type AssociationRule will be associated with a validity information of the form $[\tau_s, \tau_e]$, where τ_s and τ_e are two valid formulas of the temporal theory T^I.* □

Example 20 *Let $\mathcal{Q}$ be a query language for complex values, as the one presented in [AB95], $\mathcal{FP}$ be the logical theory with polynomial constraints over $\mathbb{R}$, and $\mathcal{V}$ be the temporal theory of intervals over $\mathbb{Z}$ (T^I). The pattern type modeling circular shaped clusters of 2D points informally introduced in the Example 2 can be modeled in $\mathcal{EPM}(\mathcal{Q}, \mathcal{FP}, \mathsf{T}^I)$ as a named tuple of the form:* $\mathsf{Cluster} : \mathsf{TUPLE}(\mathsf{s} : \mathsf{t_s}, \mathsf{d} : \mathsf{t_d}, \mathsf{m} : \mathsf{t_m}, \mathsf{f} : \mathsf{t_f}, \mathsf{v} : \mathsf{T^I})$*, where:*

$t_s \equiv$ TUPLE(center: TUPLE(cx: REAL,cy: REAL), rad: REAL)
$t_d \equiv$ SET(point: TUPLE(x: REAL, y: REAL))
$t_m \equiv$ AvgIntraClusterDistance: REAL
$t_f \equiv$ $(p :: t_e.x - \mathsf{s} :: \mathsf{t_s}.\mathsf{center.cx})^2 + (\mathsf{p} :: \mathsf{t_e}.\mathsf{y} - \mathsf{s} :: \mathsf{t_s}.\mathsf{center.cy})^2 \leq \mathsf{s} :: \mathsf{t_s}.\mathsf{rad}^2$
where $t_e =$ `TUPLE(x: REAL, y: REAL)`

The formula represents (approximatively, in this case) the pattern/data set relationship by associating each cluster with the set of 2D points internal to the circle defined by the elements in the pattern structure. In this case f is a formula of the theory $\mathcal{FP}$ whose only free variables are `s` *and* `p`*.* `s` *has type* `TUPLE(cx:REAL,cy:REAL))` *and it ranges over possible structure values.* `p` *has type* `TUPLE(x: REAL, y: REAL))` *and it ranges over elements in dom(t_d). We notice that in this case the formula represents the source elements represented by the pattern in an approximated way. We outline that the equation of the circle is a polynomial expression.* □

In the previous example, in order to be able to express the equation of the circle approximating the cluster of points we used a polynomial expression in the pattern formula. However, with a slight loss in precision, we may also approximate a 2D circle with the convex hull containing all points in the circle. In this case, we may use a simpler logical theory for expressing the pattern formula, such as the logical theory with linear constraints over $\mathbb{R}$, denoted by $\mathcal{FL}$. The pattern type, named *ConvexCluster*, instance of the $\mathcal{EPM}(\mathcal{Q}, \mathcal{FL}, \mathsf{T}^I)$ model, represented according to this choice is presented in the following example.

Example 21 *Let $\mathcal{Q}$ be a query language for complex values, as the one proposed in [AB95], $\mathcal{FL}$ be the logical theory with linear constraints over $\mathbb{R}$, and $\mathcal{V}$ be the temporal theory of*

intervals over $\mathbb{Z}$ *(*T^I*). The pattern type modeling convex clusters of 2D points modeled in* $\mathcal{EPM}(\mathcal{Q}, \mathcal{FL}, \mathsf{T}^I)$ *is a named tuple of the form:* $\mathsf{ConvexCluster} : \mathsf{TUPLE}(\mathsf{s} : \mathsf{t_s}, \mathsf{d} : \mathsf{t_d}, \mathsf{m} : \mathsf{t_m}, \mathsf{f} : \mathsf{t_f}, \mathsf{v} : \mathsf{T^I})$*, where:*

$t_s \equiv$ SET(vertex: TUPLE(vx: REAL,vy: REAL))
$t_d \equiv$ SET(point: TUPLE(x: REAL, y: REAL))
$t_m \equiv$ AvgIntraClusterDistance: REAL
$t_f \equiv$ convex_pol(s :: $\mathsf{t_s}$, p :: TUPLE(x : REAL, y : REAL))

In this case, the cluster is represented by the convex region containing all the 2D points belonging to it, i.e. their convex hull. Thus, the cluster structure is represented by the vertexes of such region, whereas the pattern data source components specifies that the pattern is extracted from a set of 2D points The formula represents (approximatively, in this case) the pattern/data relationship by associating each convex cluster with the set of 2D points internal to the convex hull defined by the convex linear polygon, which is characterized by the vertexes in the pattern structure. Therefore, f is a formula of the theory $\mathcal{FL}$*, whose only free variables are* `s` *and* `p`*.* `s` *has type* `TUPLE(vx:REAL,vy:REAL))` *and it ranges over elements in* $dom(t_s)$*.* `p` *has type* `TUPLE(x: REAL, y: REAL))` *and it ranges over elements in* $dom(t_d)$*.* convex_pol(s :: $\mathsf{t_s}$, p :: TUPLE(x : REAL, y : REAL)) *is a constraint-based representation of the polygon having as vertexes the elements in s, i.e. it is the conjunction of linear constraints describing the edges of the convex polygon with vertexes elements in s. Variables in the resulting formulas ranges over points.* □

4.2.2 Patterns

Within $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, a (basic) pattern can be defined as an instance, i.e., a legal value, of a specific (basic) pattern type. Formally, it is a tuple value with identifier, which is completely handled by the PBMS as usual in object-relational systems.

When defining the instances of a pattern type $pt_{name} : TUPLE(s : t_s, d : t_d, m : t_m, f : t_f, v : \mathcal{V})$, the following three pattern type components require some additional considerations:

- *Data source.* As we saw in Section 4.2.1, the type of the data source component corresponds to a set of tuples. In defining the domain for such component, we assume that the data set corresponding to this type is defined in an intensional way as a query expressed in $\mathcal{Q}$ over the Universal Database, having t_d as result type. Notice that taking into account an extensional modeling for the pattern data source would be an unfeasible modeling choice since this would require to associate with each pattern the entire raw data set it represents, which, in general, is quite huge.
- *Formula.* At the instance level, the formula specified in the pattern type has to

be instantiated, obtaining a new formula in $\mathcal{F}$, obtained from the original one by replacing the variable representing the pattern structure with its actual value. The only free variable in the formula associated with patterns is the one ranging over the data source component.

- *Validity period.* Any valid formula for $\mathcal{V}$ can be chosen as a value for the validity period.

Before introducing the formal pattern definition, we define special domains according to the previous considerations.

Definition 9 Let $\mathcal{Q}$ be a query language for the Universal Database. Let $\mathcal{F}$ be a logical theory with constraints for complex values. Let $\mathcal{V}$ be a temporal theory. Let $t_d \in \mathcal{T}$ such that $t_d \equiv SET(TUPLE(t_1, ..., t_n))$, $t_1, ..., t_n \in \mathcal{T}$. We define:

- $dom_i(t_d) = \{q | q$ is a $\mathcal{Q}$-query, having t_d as result type$\}$;
- $dom(\mathcal{V}) = \{\nu | \nu$ is a well formed formula for $\mathcal{V}\}$. □

Basic patterns (or simply patterns) can now be formally defined as follows.

Definition 10 (*Patterns*) Let $\mathcal{L}$ be a set of labels. Let $\mathcal{Q}$ be a query language for complex values, as the one proposed in [AB95]. Let $\mathcal{F}$ be a logical theory with constraints for complex objects. Let $ext(\mathcal{F})$ be the set of formulas that can be expressed in $\mathcal{F}$ and let $t_f \in ext(\mathcal{F})$. Let $\mathcal{V}$ be a temporal theory. Let $\mathcal{OID}$ be a set of pattern identifiers. Let $pt \in \mathcal{BPT}$, $pt \equiv pt_{name} : TUPLE(s : t_s, d : t_d, m : t_m, f : t_f, v : \mathcal{V})$ be a pattern type in $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$. The domain for pt is defined as follows:

$$dom(pt) \subseteq \mathcal{OID} \times dom(t_s) \times dom_i(t_d) \times dom(t_m) \times ext(\mathcal{F}) \times dom(\mathcal{V})$$

such that, given $p \in dom(pt)$, $p.f$ is obtained from t_f be replacing the free variable of type t_s with $p.s$. p is called pattern. Given a pattern $p \in dom(pt)$, we denote with $p.pid$ its first component, i.e., its identifier. □

From the previous definition, it follows that the formula associated with a pattern is always derived from the one specified in the pattern type the pattern is an instance of. In the following examples, some pattern instances of the pattern types introduced in Examples 19 and 20, are presented.

Example 22 *Consider the pattern type* `AssociationRule` *for* $\mathcal{EPM}(\mathcal{Q}, \mathcal{FO}, \mathsf{T}^I)$*, defined in Example 19. Suppose the Universal Database contains a relational table which stores data related to the sales transactions of a sport shop, having the following schema: Sales(transactionId, article, quantity). Let* $\mathcal{Q}$ *be an SQL-like query language. The following pattern is an instance of the pattern type* `AssociationRule`*:*

pid: 77
s: $< head = \{'Boots'\}, body = \{'Socks', 'Hat'\} >$
d: *'SELECT SETOF(article) AS transaction*
FROM sales
GROUP BY transactionId'
m: $< confidence = 0.75, support = 0.55 >$
f: $\forall x(x \in \{'Boots'\} \vee x \in \{'Socks','Hat'\} \Rightarrow x \in trs)$
v: *[01-03-2006, 30-06-2006]*

Note that, in the formula component, the variable `s`*, ranging over the source space, has been binded to the pattern structure and terms* `s.head` *and* `s.body` *have been replaced by* $\{'Boots'\}$ *and* $\{'Socks' ,'Hat'\}$*.* □

Example 23 *Consider the pattern type* `Cluster` *for* $\mathcal{EPM}(\mathcal{Q}, \mathcal{FP}, \mathsf{T}^I)$ *defined in Example 20. Suppose the Universal Database contains data concerning observations about users behavior during their navigation in a certain Web site and assume it contains a view, named* `users`*, with schema users(*<u>*userId*</u>*, age, tot_session), listing for each user, her age and the total number of her navigation sessions during a day. The following pattern is an instance of the pattern type* `Cluster`*:*

pid: 137
s: $< center :< cx : 2, cy : 3 >, rad : 4, 2 >$
d: *'SELECT age AS X , tot_session AS Y*
FROM users'
m: $< AvgIntraClusterDistance : 0.9 >$
f: $(p.X - 2)^2 + (p.Y - 3)^2 \leq (4, 2)^2$
v: *[10-03-2006, 10-04-2006]*

The data set represented by the pattern is the set of pairs X, Y *- representing, respectively, age and total number of user sessions in a day - extracted from the view* `users` *by an SQL query. The pattern measure component contains the effective average intra-cluster distance. Finally, in the pattern formula component, the variable* `s` *have been binded to the pattern structure and terms* `s.center.cx` *and* `s.center.cy` *have been replaced by values* 2 *and* 3 *as well as the term* `s.rad` *has been replaced with* 4.2*. It thus provides the circle equation used to identify the approximated set of points represented by the cluster.* □

Since patterns have been defined as tuples with identifiers, pattern equality rises several

interesting issues, as in the case of object equality in the object relational context. In particular, two different notions of pattern equality have to be considered: a first one, called *pattern identity*, is based on pattern identifiers, and a second one, called *value-based equality* or simply *equality*, based on pattern components.

More formally, *pattern identity* states that two $\mathcal{EPM}$ patterns are identical if and only if their identifiers are equal. More formally, pattern identity can be defined as follows.

Definition 11 (*Pattern identity*) Let p_1 and p_2 be two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ patterns. p_1 and p_2 are identical if and only if $p_1.pid = p_2.pid$. □

Pattern identity is a very strong notion of equality. For pattern data source and formula components, equality is a too strong criteria, that can be relaxed by considering equivalence (see Section 5.2.1.1). We notice that, an alternative approach (that we call extensional) would require checking equality between two query results or sets of values satisfying the formula. However, two $\mathcal{EPM}$ patterns *value-based equal* if and only if their corresponding components are equals, except for pattern identifiers and validity periods. Notice that pattern validity period component is not considered in determining value-based equality, since it may happen that patterns expires, but their structural and semantical properties remain the same. Since the extensional approach for checking pattern equality would requires to access the Universal Database and it may produce a huge quantity of raw data to be involved in the equality check, in order to be able to check pattern equality locally to the PBMS, we consider intensional equality for pattern data source and formula components. Concerning pattern data sources, the equality is therefore based on $\mathcal{Q}$-queries equivalence, and, concerning pattern formulas, the equality check consists in verifying the equivalence of the two pattern formulas according to the logical theory $\mathcal{F}$ (see Section 6.2.3 for additional details).

More formally, the notion of value-based equality between patterns can be defined as follows.

Definition 12 (*Value-based equality*) Let $p_1 \equiv< pid_1, s_1, d_1, m_1, f_1, v_1 >$ and $p_2 \equiv< pid_2, s_2, d_2, m_2, f_2, v_2 >$ be two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ patterns. p_1 and p_2 are value-based equal if and only if the following conditions hold: (i) $p_1.s = p_2.s$; (ii) $p_1.d$ is equivalent to $p_2.d$; (iii) $p_1.m = p_2.m$; (iv) $p_1.f$ is equivalent to $p_2.f$ with respect to the logical theory $\mathcal{F}$. □

Example 24 *Consider the pattern type 'AssociationRule' for* $\mathcal{EPM}(\mathcal{Q}, \mathcal{FO}, \mathsf{T}^I)$ *defined in Example 19. Let* p_1 *be the pattern with* $pid = 77$ *introduced in Example 22 and* p_2 *be the following one:*

pid:	*103*
structure:	$< head = \{'Boots'\}, body = \{'Socks', 'Hat'\} >$
data source:	*'SELECT SETOF(article) AS transaction FROM sales GROUP BY transactionId'*
measure:	$< confidence = 0.75, support = 0.55 >$
formula:	$\forall y(y \in \{'Boots'\} \vee x \in \{'Socks','Hat'\} \Rightarrow y \in trs)$
validity:	*[01-01-2006, 01-07-2006]*

Both patterns p_1 and p_2 are instances of the `AssociationRule` *pattern type. They are value-based equal (indeed, $p_1.s = p_2.s$, $p_1.d$ is equivalent to $p_2.d$, $p_1.m = p_2.m$, and $p_1.f \equiv_{\mathcal{F}} p_2.f$), but not identical ($77 \neq 103$).* □

We notice that, value-based pattern equality can be checked also over patterns of different types.

According to Def. 10, $\mathcal{EPM}$ patterns are associated with a unique identifier ($p.pid$); thus, a dereferencing mechanism is required in order to obtain the pattern itself whenever a pattern identifier is given. For this reason, we consider a *dereferencing function*, denoted by $\gg$: $\mathcal{OID} \longrightarrow \bigcup_{pt \in \mathcal{PT}} dom(pt)$, that given a pattern identifier returns the entire pattern it refers to, i.e. $\gg (o) =< o, s, d, m, f, v >$ instance of the pattern type $pt \in \mathcal{PT}$.

Since patterns have been defined as tuples, path expressions can be used to directly access a pattern component. The following definition formalizes the concept of path expressions in an inductive way.

Definition 13 $O.A_i$ is a valid path expression if and only if one of the following conditions holds:

- $O \in dom(t)$, $t \in \mathcal{PT}$, and $t = TUPLE(..., A_i : t_i, ...)$;
- $O \in OID$ and $\exists pt \in \mathcal{PT}$, $\exists p \in dom(pt)$ such that $p.pid = O$ and A_i is a pattern component of p.

By induction, we say that $O.A_1. ... .A_i$ is a path expression if and only if $O.A_1. ... .A_{i-1}$ is a path expression denoting a value of type $t = TUPLE(..., A_i : t_i, ...)$ or an object identifier $o \in \mathcal{OID}$ such that $\exists p \in dom(pt)$, $pt \in \mathcal{PT}$, having o as pattern identifier. □

Example 25 *Consider the pattern presented in Example 22 and the pattern presented in Example 23. Let p be the first pattern and q be the second one. The following are examples of valid path expressions: $p.s$, $77.s$, $p.s.head$, $q.s.rad$, $q.m.AvgIntraClusterDistance$, and $137.s.center.cy$* □

4.2.3 Classes

A *pattern class* over a pattern type for $\mathcal{EPM}$ 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. In the following, we first introduce the notion of *pattern class schema* and, then, we define the *pattern class*.

Definition 14 (*Pattern Class Schema*) Let $\mathcal{PTN}$ be the set of all pattern type names. A *Pattern Class Schema* has the form $\widehat{Cname}$ :: pt such that $Cname$ is a unique identifier among all classes and $pt \in \mathcal{PTN}$ is a pattern type name. □

An instance of a pattern class schema $\widehat{Cname}$ is called *pattern class* and it can be formally defined as follows.

Definition 15 (*Pattern Class*) Let $\widehat{Cname}$ :: pt be a pattern class schema for the pattern type pt. A *Pattern Class* $Cname$, instance of $\widehat{Cname}$, is a set $S(pt)$ such that $S(pt)$ is a finite set of patterns instances of pattern type pt. □

We notice that classes are homogeneous collection of patterns, i.e., they contain patterns which are instances of the same pattern type. On the other side, a single pattern can belong to different classes (defined for its pattern type).

4.3 Hierarchical Extended Pattern Model

As already informally discussed in Chapter 3, patterns can share various hierarchical relationships. Introducing hierarchies in $\mathcal{EPM}$ rises several interesting issues concerning pattern types and patterns.

In the following, we first consider the impact of hierarchies over pattern type definition (Section 4.3.1). Then, we focus on the impact of hierarchies over patterns and classes (Section 4.3.2). Finally, pattern hierarchies are formally defined (Section 4.3.3).

4.3.1 Hierarchical Pattern Types

Intuitively, dealing with pattern type hierarchies means to be able to refer a pattern type inside the definition of another pattern type. There are two different ways to refer a pattern type when creating a pattern type: a direct one and a non direct one. The direct one relies

on the usage of pattern type names, whereas the non direct one involves references to pattern types. Values for pattern type references are pattern identifiers. The usage of identifiers in the pattern model increases the re-usage of previously detected knowledge in defining new patterns. The pattern model is therefore a kind of complex object model.[3] Formally, within $\mathcal{EPM}$, pattern hierarchies can be modeled by extending the type system with pattern type names and references to pattern types as pointed out by the following definition.

Definition 16 (*Extended Types*) Let $\mathcal{AT}$ be the set of atomic types introduced in Def 3. Let $\mathcal{L}$ be a set of type name labels. Let $\mathcal{PTN}$ be the set of pattern type names introduced in Def 8. The set of extended types $\mathcal{T}_e$ is inductively defined as follows:[4]

1. $\top$ is a type and it is called 'root' type;
2. $\forall t_i \in \mathcal{AT} \cup \mathcal{PTN}$, $t_i \in \mathcal{T}_e$ and it is an unnamed extended type;
3. if $t \in \mathcal{PTN}$, then $ref(t) \in \mathcal{T}_e$ and it is an unnamed extended type;
4. if $t \in \mathcal{T}_e$, then $SET(t) \in \mathcal{T}_e$ and it is an unnamed extended type;
5. if $t_1, t_2, ..., t_n \in \mathcal{T}_e$, $n \geq 1$ and $a_1, ..., a_n \in \mathcal{L}$, then $TUPLE(a_1 : t_1, ..., a_n : t_n) \in \mathcal{T}_e$ and it is an unnamed extended type;
6. if $t \in \mathcal{T}_e$ is an unnamed extended type (see above) and $A \in \mathcal{L}$ is a name, then $A : t \in \mathcal{T}_e$ and it is a named extended type.
7. if $A : t \in \mathcal{T}_\urcorner$, then $A \in \mathcal{T}_\urcorner$ and it is a named type. □

Before introducing the formal definition of a hierarchical pattern type, we define domains for the newly introduced types.

Definition 17 Let $\mathcal{OID}$ be a set of pattern identifiers. We define:

$$dom(ref(t)) = \{o | o \in \mathcal{OID} \wedge \exists p \in dom(t)\, p.pid = o\}$$

. □

[3]The choice of an object model for patterns provides more flexibility in defining pattern concepts. On the other hand, we choose a complex values model for data in order to simplify the presentation of the proposed concepts, since source data modeling is not a primary issue of this work.

[4]We notice that the definition of $\mathcal{T}_e$ is similar to the definition of $\mathcal{T}$ (Def. 4), the only difference is that the induction is based on $\mathcal{AT} \cup \mathcal{PTN}$ instead of $\mathcal{AT}$.

By using $\mathcal{T}_e$, we can formally define a pattern type including hierarchies, i.e., a *Hierarchical Pattern Type* as follows.

Definition 18 (*Hierarchical Pattern Type*) Let $\mathcal{PTN}$ be a set of pattern type names. Let $\mathcal{Q}$ be a pattern query language. Let $\mathcal{F}$ be a logical theory with constraints for objects. Let $\mathcal{V}$ be a temporal theory. Let $ext(\mathcal{F})$ be the set of formulas that can be expressed in $\mathcal{F}$. Let $t_s \in \mathcal{T}_e$, $t_d \in \mathcal{T}_e$, $t_m \in \mathcal{T}$, $Name \in \mathcal{PTN}$, and $t_f \in ext(\mathcal{F})$. A hierarchical pattern type pt for $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ is a named type of the form :

$$Name : TUPLE(s : t_s, d : t_d, m : t_m, f : t_f, v : \mathcal{V})$$

such that:

- $t_s \in \mathcal{T}_e$ and $Name$ does not appear in t_s;[5]
- t_d has one of the following types:
  - **(i)** $SET(TUPLE(t_1, ..., t_n))$, $t_1, ..., t_n \in \mathcal{T}$;
  - **(ii)** $SET(t)$, where $t \in \mathcal{PTN} - \{Name\}$ and t is a name for a pattern type in $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$;
- t_m is a type of the form $TUPLE(t_1, ..., t_k)$, $t_1, ..., t_k \in \mathcal{AT}$, defining the type of measures which quantify the quality of the source data achieved by a pattern instance of the pattern type being defined, often $t_1, ..., t_k$ are atomic types representing numerical values;
- $t_f \equiv \varphi(\tau, \psi)$, where φ is a well formed formula of the theory $\mathcal{F}$ and τ, ψ are the only free variables in φ. τ has type $TUPLE(t_1, ..., t_n)$ or $t \in \mathcal{PTN} - \{Name\}$, thus, implicitly ranges over the domain of data source elements, whereas ψ has type t_s (then, it ranges over the pattern structure).

We denote with $\mathcal{PT}$ the set of all possible pattern types. □

We notice that, with respect to the non hierarchical pattern type definition (Def. 18), in this case, the Universal Database may contain patterns. Then, $\mathcal{Q}$, which must be a query language for the Universal Database, must be a query language for data and pattern bases. Moreover the logical theory with constraint for modeling hierarchical pattern type formula, i.e. $\mathcal{F}$, has to be a theory for complex objects.

[5]Note that, by imposing the type of the pattern structure being in $\mathcal{PT} - \{Name\}$ we force the structure schema not to be recursive.

Example 26 *let $\mathcal{Q}$ be a query language for complex objects, $\mathcal{FE}$ be the logical theory for objects with equality constraints, and $\mathcal{V}$ be the temporal theory of unions of intervals over $\mathbb{Z}$ (T^U). The hierarchical pattern type modeling Clusters of Association Rules informally introduced in the Example 9 can be modeled in $\mathcal{EPM}(\mathcal{Q}, \mathcal{FE}, \mathsf{T}^U)$ as a named tuple of the form:* $\mathsf{ClusterOfRules : TUPLE(s : t_s, d : t_d, m : t_m, f : t_f, v : T^U)}$*, where:*

$t_s \equiv$ TUPLE(representative: AssociationRule)
$t_d \equiv$ SET(rule: AssociationRule)
$t_m \equiv$ TUPLE(deviationOnConfidence: REAL, deviationOnSupport: REAL)
$t_f \equiv$ r::rule.s.head=p::representative.s.head

The structure of the pattern type `ClusterOfRules` *consists in a single element of type* `AssociationRule` *(see Example 19), called 'representative'. Note that also the data source is a set of patterns of type* `AssociationRule`. □

The impact of hierarchies in the definition of a hierarchical pattern type is twofold. It may concern the structure of the pattern type or its data source. In the first case, it indicates that a pattern instance of this pattern type is composed of other patterns. In other words, it means that within the structure of a pattern other patterns may appear as sub-components. In the second case, this means that a pattern instance of this pattern type is, for example, obtained by mining other patterns. In this case, the data source of the pattern is a collection of patterns of a different pattern type. This motivates the type of the pattern type data source which has been defined as $SET(t)$, where $t \in \mathcal{PTN}$ is a pattern type different from the one under definition.

The following definition introduces a notation for specifying both situations.

Definition 19 Let $nome_1 : pt_1, nome_2 : pt_2 \in \mathcal{PT}$. We say that pt_1 refers pt_2 through z, $z \in \{s, d\}$, where z is either the structure or the data source component, (denoted by $pt_1 \rightarrowtail_z pt_2$) if and only if one of the following conditions holds: (i) $name_2$ appears in $pt_1.z$, or (ii) ref($name_2$) appears in $pt_1.z$. □

Example 27 *Consider the* `AssociationRule` *pattern type (see Example 19) and the* `ClusterOfRules` *pattern type (see Example 26). Then, $ClusterOfRules \rightarrowtail AssociationRule$* □

4.3.2 Hierarchical Patterns

Within $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, a hierarchical pattern can be defined as an instance, i.e., a legal value, of a specific hierarchical pattern type. Hierarchical patterns can therefore be formally defined similarly to non hierarchical ones, as follows.

Definition 20 (*Hierarchical Patterns*) Let $\mathcal{L}$ be a set of labels. Let $\mathcal{Q}$ be a query language for complex object databases. Let $\mathcal{F}$ be a logical theory with constraints for complex objects. Let $ext(\mathcal{F})$ be the set of formulas that can be expressed in $\mathcal{F}$ and let $t_f \in ext(\mathcal{F})$. Let $\mathcal{V}$ be a temporal theory. Let $\mathcal{OID}$ be a set of pattern identifiers. Let $pt \in \mathcal{PT}$, $pt \equiv pt_{name} : TUPLE(s : t_s, d : t_d, m : t_m, f : t_f, v : \mathcal{V})$ be a pattern type in $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$. Let $\mathcal{Q}'$ be a pattern query language for $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ pattern bases. The domain for pt is defined as follows:

$$dom(pt) \subseteq \mathcal{OID} \times dom(t_s) \times dom_i(t_d) \times dom(t_m) \times ext(\mathcal{F}) \times dom(\mathcal{V})$$

such that, given $p \in dom(pt)$, $p.f$ is obtained from t_f be replacing the free variable of type t_s with $p.s$. p is called pattern. Given a pattern $p \in dom(pt)$, we denote with $p.pid$ its first component, i.e., its identifier. □

The previous definition concerning hierarchical relationships between pattern types (Def. 19) can be easily extended to patterns as follows.

Definition 21 Given two patterns $p_1 \in dom(pt_1)$ and $p_2 \in dom(pt_2)$, we say that p_1 refers p_2 through z, where z is either the structure or the data source component, i.e., $z \in \{s, d\}$, (denoted by $p_1 \rightarrowtail_z p_2$) if and only if $pt_1 \rightarrowtail_z pt_2$ and one of the following conditions holds: (i) if $name_2$ appears in $pt_1.z$, then p_2 appears in $p_1.z$, or (ii) if ref($name_2$) appears in $pt_1.z$, then $p_2.pid$ appears in $p_1.z$. □

We point out that, when considering hierarchical patterns, the notion of value-based pattern equality can be specialized in a shallow equality and in a deep equality. As usual in object databases, shallow equality coincides with the one previously introduced (see Def. 12), whereas deep equality requires a value-based equality for all the corresponding patterns referred in the structure or in the data source components of the original two hierarchical patterns.

Finally, we observe that the definition for $\mathcal{EPM}$ pattern class schema (Def. 14) and pattern classes (Def. 15) remain the same even if we consider pattern types with hierarchies. The same holds for definitions concerning $\mathcal{EPM}$ pattern base schema (Def. 26) and pattern base instance (Def. 27).

4.3.3 Relationships among Pattern Types and Patterns

According to the informal definition presented in Section 3.3.4, two different pattern type hierarchies can be defined: the *composition* hierarchy and the *refinement* hierarchy. The composition hierarchy models references to other pattern types inside the structure; on the

other hand, the refinement hierarchy models references to other pattern types in the data source.[6]

More precisely, the $\mathcal{EPM}$ *composition* relationship addresses the possibility of defining the structure schema of a certain pattern type based on other already existing pattern types. From a conceptual point of view, it models a *part-of* hierarchy between two pattern types. In more details, the composition relationship between a pattern type $pt_1 \in \mathcal{PT}$ and another pattern type $pt_2 \in \mathcal{PT}$ means that the type of the structure schema component of pt_1 refers pt_2. Of course, a composition hierarchy between two pattern types induces a hierarchy over pairs of patterns, instances of those pattern types.

Definition 22 (*Composition relationship*) Let $pt_1, pt_2 \in \mathcal{PT}$. There exists a composition relationship between pt_1 and pt_2 (denoted by $pt_1 \hookrightarrow pt_2$) if $pt_1 \rightarrowtail_s pt_2$. In this case, we say that pt_2 composes pt_1. Let $p_1 \in dom(pt_1)$ and $p_2 \in dom(pt_2)$. There exists a pattern composition relationship between p_1 and p_2, denoted by $p_1 \hookrightarrow p_2$ if and only if $p_1 \rightarrowtail_s p_2$ In the following, we denote with $\hookrightarrow^*$ the transitive closure of $\hookrightarrow$. □

The $\mathcal{EPM}$ *refinement* relationship addresses the possibility of specifying that a pattern of a certain type is associated with a data source containing other patterns. Note that this relationship is called 'refinement' in order to emphasize that, by moving from a pattern type to the pattern type that appears in its data source, the level of details in representing knowledge increases, as shown in Fig. 4.1. More specifically, a refinement relationship between a pattern type $pt_1 \in \mathcal{PT}$ and another pattern type $pt_2 \in \mathcal{PT}$ models the fact that the type of the data source schema component of pt_1 (i.e., $pt_1.ds$) refers pt_2. Of course, a refinement hierarchy between two pattern types induces a hierarchy over pairs of patterns, instances of those pattern types.

Definition 23 (*Refinement relationship*) Let $pt_1, pt_2 \in \mathcal{PT}$. There exists a refinement relationship between pt_1 and pt_2 (denoted by $pt_1 \Rightarrow pt_2$) if $pt_1 \rightarrowtail_d pt_2$. In this case, we say that pt_2 refines pt_1. Let $p_1 \in dom(pt_1)$ and $p_2 \in dom(pt_2)$. There exists a pattern refinement relationship between p_1 and p_2, denoted by $p_1 \Rightarrow p_2$ if and only if $p_1 \rightarrowtail_d p_2$ In the following, we denote with $\Rightarrow^*$ the transitive closure of $\Rightarrow$. □

Example 28 *Consider the pattern type* `ClusterOfRules` *presented in Example 26 and the pattern type* `Association Rules` *introduced in the Example 19. There is a refinement*

[6]Actually, if we want to deal with patterns under a pure object-oriented paradigm, we would have to consider inheritance aspects. Therefore, a specialization relationship between pattern types would have to be considered in order to allow pattern types to be defined starting from existing ones, specializing some of their components. in this thesis, for the sake of simplicity, we do not take into account the possibility of defining a specialization hierarchy between pattern types.

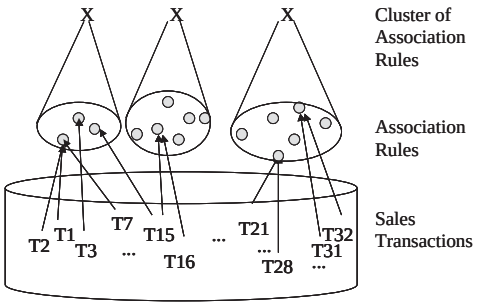


Figure 4.1: Refinement and composition hierarchies

relationship between `ClusterOfRules` *and* `AssociationRule`*, since the data source of a cluster of rules is a set of patterns of type* `AssociationRules`*. In addition, since the representative of each pattern of type* `ClusterOfRules` *is an association rule, there exists also a composition relationship between those two pattern types. These refinement and composition hierarchies are depicted in Fig. 4.1.* □

4.3.4 Pattern Base

In this section we exploit the $\mathcal{EPM}$ concepts introduced before (i.e., pattern type, pattern, and classes) to formally define the notions of *Pattern Base Schema* and *Pattern Base*.

As we have already discussed in Chapter 3, a-posteriori patterns are the most common patterns in many important domains (such as, for instance, data mining) and a full support for this kind of patterns is a fundamental requirement for any PBMS. To this end, in order to fully support the management of a-posteriori patterns according to the requirements discussed in Section 3.1, within an $\mathcal{EPM}$ pattern base, information concerning mining methods, used to extract patterns from raw data, and evaluation methods, used to assess the quality of pattern representation with respect to raw data, have to be represented. In details, we introduce the concept of *mining function* as an abstraction of the whole pattern extraction process and the concept of *measure function* as an abstraction for the computation of quality measures.

Based on the previous considerations, an $\mathcal{EPM}$ pattern base will then be defined as a storage structure, comprising pattern types and patterns defined over raw data and collected into pattern classes, where each pattern type is coupled with several possible mining and measure functions.

The remainder of this section is organized as follows. Section 4.3.4.1 introduces the concept of *mining function* supporting pattern extraction from raw data, whereas Section 4.3.4.2 presents the concept of *measure function*, which is used to evaluate pattern quality measures

over a certain data set. Notions of mining and measure functions are, then, used to formalize the notions of *Pattern Base Schema* and *Pattern Base* in Section 4.3.4.3.

4.3.4.1 The Mining Function

Most of the patterns we are interested in are the result of an extraction process, i.e., they are *a-posteriori patterns*. As already argumented in Chapter 3, in order to model the extraction phase, the notion of *mining function* is required. In details, a mining function is a function used to generate patterns - of a certain pattern type - from raw data. As we introduced in Section 3.3.5, a mining function is defined for a certain pattern type. However, several different mining functions can be defined for the same pattern type.

Given a pattern type $pt \in \mathcal{PT}$ with $pt.m = TUPLE(m_1 : t_1, ..., m_n : t_k)$ a mining function μ for pt takes as input a data set D of type $pt.t_d$ and k threshold values (one for each measure sub-component) $th_1, ..., th_k$, applies a certain computation over D, and returns a set of patterns instances of pt. Threshold values are used by μ to filter obtained patterns. Indeed, only those patterns having measure values better than the corresponding thresholds are retained; the others are discarded.

In order to establish whether a pattern measure value is better than another one, several ad-hoc comparison operators can be introduced. In particular, we assume a comparison operator θ_i exists for each pattern measure $pt.m.m_i$. The behavior of such operators is specialized with respect to the semantics of the pattern quality measure. For example, when considering association rules, a comparison operator for confidence values would choose as best value the highest one; on the other hand, when considering clusters of 2D points, better values for average intra distance measure would be those closer to zero.

A mining function μ is formally defined as follows.

Definition 24 (*Mining Function*) Let $\mathcal{Q}$ be a query language for object databases. Let $\mathcal{F}$ be a logical theory with constraints for complex objects. Let $\mathcal{V}$ be a temporal theory. Let $pt \in \mathcal{PT}$, $pt \equiv pt_{Name} : TUPLE(s : t_s, d : t_d, m : TUPLE(m_1 : t_1, ..., m_k : t_k), f : t_f, v : \mathcal{V})$ be a pattern type in $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$. Let $th_i \in dom(t_i)$, $i = 1, ..., k$. Let θ_i be a comparison operator for m_i, $i = 1, ..., k$. A mining function μ for pt within $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ is a function with signature:

$$\mu : dom(pt.t_d) \times dom(t_1) \times ... \times dom(t_k) \longrightarrow \wp(dom(pt))$$

such that $\forall p \in \mu(D, th_1, ..., th_k)$, $p.m_i \theta_i th_i$, $i = 1, ..., k$. When no threshold values are provided, we simply write $\mu(D)$. □

Example 29 *A mining function modeling the Apriori algorithm defined for* `AssociationRule` *pattern type (see Example 19) has the following signature:* $\mu_{Apriori} : dom(SET(transaction : SET(STRING))) \times [0,1] \times [0,1] \longrightarrow \wp(dom(AssociationRule))$. □

4.3.4.2 The Measure Function

As already discussed, measures assess the goodness of the representation of a certain data set achieved by a pattern. Within $\mathcal{EPM}$, the function computing measure values of patterns, instances of a certain pattern type over a certain data set, is called *measure function* (μ_m).

Within $\mathcal{EPM}$, a measure function μ_m is defined for a certain pattern type $pt \in \mathcal{PT}$. It takes in input a pattern p of type pt and a data set D in $dom(pt.t_d)$. The output of μ_m is a value for $pt.t_m$, i.e., a new tuple of measures for p.

Definition 25 (*Measure Function*) Let $\mathcal{Q}$ be a query language for complex object databases. Let $\mathcal{F}$ be a logical theory with constraints for complex objects. Let $\mathcal{V}$ be a temporal theory. Let $pt \in \mathcal{PT}$, $pt \equiv pt_{Name} : TUPLE(s : t_s, d : t_d, m : TUPLE(m_1 : t_1, ..., m_k : t_k), f : t_f, v : \mathcal{V})$, be a pattern type in $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$. Let $th_i \in dom(t_i)$, $i = 1, ..., k$. A measure function μ_m for pt is a total function with signature:

$$\mu_m : dom(pt) \times dom(pt.t_d) \longrightarrow dom(pt.t_m)$$

□

As we will see measure functions are required at different steps of the pattern manipulation process. In general, they are needed whenever we want to align the knowledge represented by a pattern with the underlying data.

Example 30 *Consider the pattern type* `AssociationRule` *defined in Example 19. A measure function* μ_{AR} *for this pattern type has the following signature:* $\mu_{AR} : dom(AssociationRule) \times dom(SET(transaction : SET(STRING))) \longrightarrow [0,1] \times [0,1]$. □

4.3.4.3 Pattern Bases

In the following, similarly to what stated in Section 4.1.3 concerning databases, the notions of *Pattern Base Schema* and *Pattern Base* within $\mathcal{EPM}$ are formalized.

Definition 26 (*Pattern Base Schema*) Let $\mathcal{Q}$ be a query language for complex object databases. Let $\mathcal{F}$ be a logical theory with constraints for complex objects. Let $\mathcal{V}$ be a temporal theory. Let $\mathcal{CN}$ be a set of class names. A *Pattern Base Schema* in $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, defined over a database schema $\widehat{DB}$, is defined as a tuple

$$\widehat{PB} = \langle [\widehat{D_1}, \ldots, \widehat{D_k}], [PT_1, ..., PT_h], [\widehat{PC_1} :: PT_1, \ldots, \widehat{PC_m} :: PT_m], [\mu_1, ..., \mu_r], [\mu_{m1}, ..., \mu_{ms}] \rangle$$

where:

- $\widehat{D_i}$ is a data type, $i = 1, ..., k$;
- $PT_i \in \mathcal{PT}$, $i = 1, .., h$, is defined upon data types $\widehat{D_1}, ..., \widehat{D_k}$;
- $\{PT_1, ..., PT_m\} \subseteq \{PT_1, ..., PT_h\}$;
- $\widehat{PC_i} \in \mathcal{CN}$, $i = 1, ..., m$;
- μ_i, $i = 1, ..., r$, is a mining function for a pattern type in $\{PT_1, ..., PT_h\}$;
- μ_{mi}, $i = 1, ..., s$, is a measure function for a pattern type in $\{PT_1, ..., PT_h\}$. □

Once introduced the concept of pattern base schema, we are able to define the concept of *Pattern Base* as an instance of a certain pattern base schema.

Definition 27 (*Pattern Base*) Let $\mathcal{Q}$ be a query language for an object databases. Let $\mathcal{F}$ be a logical theory with constraints for complex objects. Let $\mathcal{V}$ be a temporal theory. Let $\widehat{PB}$ be a pattern base schema in $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, defined over a database schema $\widehat{DB}$. A Pattern Base instance of $\widehat{PB}$ is a structure

$$PB = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$$

where

- D_i is a domain for $\widehat{D_i}$;
- $ext(PT_i)$ is a set of pattern instances for PT_i, $i = 1, .., h$;
- PC_i is a class extension for PT_i, $i = 1, ..., m$. □

According to Def. 27 a pattern base may contain patterns which are inserted into one or more classes and patterns not belonging to any classes. However, as discussed in Chapter 3, only patterns inserted in at least a class can be queried.

4.4 Pattern Validity and Safety

Based on definitions for pattern types and patterns in $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, it is possible to deal with semantic and temporal pattern validity issues in a parametric way with respect to the chosen temporal theory $\mathcal{V}$.

Independently from the chosen temporal theory $\mathcal{V}$ used in $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, the notion of *temporal validity* expresses that a pattern is *assumed* to be valid at a certain instant of time. Since, according to $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, a hierarchical pattern can refers other patterns in its structure, in the case it exploits a composition hierarchy , and/or in its data source, whenever it exploits a refinement hierarchy, (see Section 4.3.2), two different versions of the temporal validity notion have to be introduced: a *shallow temporal validity* and a *deep temporal validity*. The shallow temporal validity notion simply considers the pattern itself, while the deep temporal validity also considers, in a recursive way, all other patterns referred by the pattern. In the following, the two notions of temporal validity for patterns are formally defined.

Definition 28 (*Shallow Pattern Temporal Validity*) Let $pt \in \mathcal{PT}$ be a pattern type for $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ and $p \in dom(pt)$. Let t be an instant of time expressed according to $\mathcal{V}$. p is shallow temporally valid at instant t if t satisfies $p.v$. □

Definition 29 (*Deep Pattern Temporal Validity*) Let $pt \in \mathcal{PT}$ be a pattern type for $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ and $p \in dom(pt)$. Let t be an instant of time expressed according to $\mathcal{V}$. p is deep temporally valid at instant t if p is shallow temporally valid at t and $\forall p'$ s.t. $p \rightarrowtail_z p'$, $z \in \{s, d\}$, p' is deep temporally valid at t. □

We stress that the shallow and deep versions of the validity notions are different only for hierarchical patterns using composition or refinement relationships. For basic patterns those two validity notions coincide as shown in the folllowing example.

Example 31 *Consider the pattern presented in Example 22 of type* `AssociationRule`*, where* T^I *is the temporal theory chosen to model the validity information (see Example 19). It is both shallow and deep temporally valid in* 01-*Apr*-2006*, but it is not temporally valid in* 31-*Dec*-2005 *(neither in a shallow nor deep way).* □

In the remainder of this thesis, when no otherwise specified, when referring to temporal pattern validity, we mean the shallow pattern temporal validity.

Since raw data change with a high frequency, it may happen that a pattern, even if it is temporally valid at a certain instant of time, does not correctly represent raw data from

which it has been extracted or, in general, it is associated with. To this purpose, the following concept of *semantic validity* with respect to a certain data source is introduced. In details, an $\mathcal{EPM}$ pattern is semantically valid with respect to a certain data source D (possibly different from the one to which is has been mined) with respect to some thresholds if at a certain instant of time it represents the data set D with measures greater than the specified thresholds.

Definition 30 (*Pattern Semantic Validity*) Let $pt \in \mathcal{PT}$ be a pattern type such that $pt.m = TUPLE(m_1 : t_1, .., m_n : t_n)$ and $p \in dom(pt)$ an a-posteriori pattern, extracted by using a mining function μ. Let t be an instant of time expressed according to $\mathcal{V}$. Let θ_i be a comparison operator for m_i. p is semantically valid at time t with respect to D with thresholds $v_1, ..., v_n$, denoted by $D \models_t p$, if there exists $\overline{p} \in \mu(p.d(UDB), v_1, ..., v_n)$, computed at time t, such that: $p.s = \overline{p}.s$ and $p.m.m_i\ \theta_i\ v_i$, $i = 1, ..., n$. □

We remark that in the previous definition the element $p.d(UDB)$, given in input to the mining function μ, represents the result of the execution of query defining $p.d$ over the Universal Database (UDB).

Note that semantic validity can only be checked at an instant-by-instant base, since we do not know how the data source will change in the future and how it was in the past. Under this perspective, the semantic validity of patterns can be seen as a function of time. Indeed, by checking it periodically, we may monitor how pattern measures change along the time, an therefore how data representation achieved by the pattern changes.

At this point, by exploiting the previous notions of pattern validity, the notion of pattern *safety* can be introduced. A pattern is safe when it is both temporally and semantically valid at a certain instant of time as formalized in the following definition. Depending on which pattern validity notion we consider, we obtain a shallow version or a deep one of the safety notion.

Definition 31 (*Pattern Safety*) Let $pt \in \mathcal{PT}$ be a pattern type for $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, $pt = TUPLE(s : t_s, d : t_d, m : TUPLE(m_1, ..., m_n), f : t_f, v : \mathcal{V})$. Let $p \in dom(pt)$ with $p.m =< m_1 : v_1, ..., m_n : v_n >$. Let μ be a mining function for pt. Let $D \subseteq UDB$ and t be an instant of time expressed according to $\mathcal{V}$. p is {shallow,deep} safe at t if it is both {shallow, deep} temporally and semantically valid at instant t with respect to D and $v_1, ..., v_n$. □

Example 32 *Consider the pattern type* `AssociationRule` *defined in Example 19 and the instance with pid = 77 presented in Example 22, mined from data stored in table Sales(transactionId, article, quantity). Since such pattern has a validity period [01-03-2006,30-06-2006], we assume it is reliable from 01-03-2006 to 30-06-2006. In particular,*

as an example, the pattern is temporally valid on 22-04-2006. However, since raw data is continuously changing, it may be possible that, on 22-04-2006, no transaction in the pattern data source (say D) contains both 'Hat' *and* 'Boots'. *Thus, the support and the confidence of this pattern evaluated over D on 22-04-2006 is* 0. *Therefore, on 22-04-2006, the pattern is not semantically valid with respect to D, for any threshold values, and it is not safe.* □

We conclude this section by pointing out that according to our models, we associate with each pattern a temporal information concerning its validity (Section 4.2) and we assume a possible temporal validity information is specified also over raw data (Section 4.1). Both these two temporal information are modeled by using a suitable temporal theory (see Section 6.2.2 for possible temporal theory options). However, the temporal theory chosen to model temporal information associated with raw data and the one used for patterns may not coincide. For instance, it may happen that the temporal theory used for modeling raw data validity is T^I over $\mathbb{Z}$, whereas the temporal theory used for modeling pattern validity information is T^P over $\mathbb{Z}$.

For this reason, we introduce a notion of *temporal reasonableness* of a pattern type, in order to formalize the concept that it models patterns with temporal validity information expressed according to a temporal theory which is *reasonable* with respect to the one used for modeling source data validity. In order to define this notion, we may assume that temporal theories are ordered with respect to their expressive power, i.e., with respect to the set of theorems which are valid in a given theory. Given two temporal theories T_1 and T_2 we use the notation $T_1 \propto T_2$ to point out that the set of theorems of T_1 is contained in the set of theorems of T_2 (see Section 6.2.2 for additional details). Based on this ordering relation, we require that, within $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, the temporal theory $\mathcal{V}$ chosen for modeling pattern validity is not less expressive than the theory $\mathcal{W}$ used by the data source model, i.e., $W \propto V$. For instance, if $W = \mathsf{T}^U$ over $\mathbb{Z}$, $\mathcal{V}$ cannot be T^I over $\mathbb{Z}$, since T^I over $\mathbb{Z} \propto \mathsf{T}^U$ over $\mathbb{Z}$.

This consideration about validity information has an impact on the choice of the mining function, which must be able to deal with temporal validity information. Unfortunately, as pointed out in Section 3.3.5, up to now only few mining functions take care of this aspect. In most cases, time is considered as a data attribute and no validity information is generated for the extracted patterns. We however believe that the notion of temporal reasonableness is however significant in order to design new mining functions considering those aspects, which seem to be relevant from an application point of view.

Chapter 5

Languages for $\mathcal{EPM}$ Pattern Management

The aim of this chapter is to introduce languages for pattern querying and manipulation assuming patterns are represented according to the $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ model.

A *query language* for patterns modeled according to $\mathcal{EPM}$ is a notation for expressing queries involving patterns in the Pattern Base along with raw data in the Data Base, coupled with a mechanism for clearly identifing the semantics of these expressions. A query language for $\mathcal{EPM}$ patterns is called *Extended Pattern Query Language ($\mathcal{E}$PQL)*. An algebra for querying $\mathcal{EPM}$ patterns is called *Extended Pattern Query Algebra* and a calculus for querying $\mathcal{EPM}$ patterns is called *Extended Pattern Query Calculus*.

Moreover, patterns in the PBMS have to be manipulated, e.g. extracted from raw data or inserted in the system from scratch, updated or (re)aligned with raw data they represent, and deleted. A *pattern manipulation language* should support all these functionalities. According to what stated in Chapter 5, a manipulation language for $\mathcal{EPM}$ patterns is called *Extended Pattern Manipulation Language ($\mathcal{E}$PML)*.

In this chapter, after introducing in Section 5.1 a classification of possible queries types the user may pose over a PBMS, Extended Pattern Query Languages are introduced in Section 5.2. In particular, both an algebra named $\mathcal{E}$APQL (Section 5.2.2) and a calculus for querying patterns are presented $\mathcal{E}$CPQL (Section 5.2.3). Section 5.3 concludes the chapter by presenting the Extended Pattern Manipulation Language.

5.1 Query Types

Based on the nature of their input and output parameters, it is possible to present a classification of PBMS queries. We are interested in four different types of queries and we group these query types in two distinct classes based on their output parameter: *D-class* and *P-class*. Queries in *D-class* return as output data, whereas queries in *P-class* return as output patterns. Queries belonging to *D-class* are called *D-queries* and queries belonging to *P-class* are called *P-queries*. In the following, we denote with q_D a generic D-query and with q_P a generic P-query.

Within each class of queries, it is possible to discriminate between *simple queries* and *cross-over queries* with respect to their input. Queries taking as input both a data set and a pattern set and returning as output either a new data or pattern set are called *cross-over queries*. The others, i.e. the ones taking in input a data set and producing a data set or taking in input a pattern set and producing a pattern set, are called *simple queries*. Simple D-queries can be solved locally to the DBMS and, similarly, simple P-queries are processed locally to the PBMS. On the other side, cross-over queries (in both classes) require for their execution both systems.

D-queries and *P-queries* can be formally defined as follows.

Definition 32 (*D-query*) Let $\widehat{DB}$ be a database schema (see Def. 6) and $\widehat{PB}$ be an $\mathcal{EPM}$ pattern base schema defined over $\widehat{DB}$ (see Def. 26). A *simple D-query*, denoted by q_D^s, is a function with signature $2^{\widehat{DB}} \longrightarrow 2^{\widehat{DB}}$. A *cross-over* D-query, denoted by q_D^{co}, is a function with signature $2^{\widehat{DB}} \times 2^{\widehat{PB}} \longrightarrow 2^{\widehat{DB}}$. We call *D-query* either a simple or cross-over D-query. □

Definition 33 (*P-query*) Let $\widehat{DB}$ be a database schema (see Def. 6) and $\widehat{PB}$ be an $\mathcal{EPM}$ pattern base schema in defined over $\widehat{DB}$ (see Def. 26). A *simple* P-query, denoted by q_P^s, is a function with signature $2^{\widehat{PB}} \longrightarrow 2^{\widehat{PB}}$. A *cross-over* P-query, denoted by q_P^{co}, is a function with signature $2^{\widehat{DB}} \times 2^{\widehat{PB}} \longrightarrow 2^{\widehat{PB}}$. We call *P-query* either a simple or cross-over D-query. □

In Fig. 5.1 all types of *D-query* and *P-query* are sketched pointing out the interactions with the PBMS and the DBMS

The query languages we are going to present support the specification of all query types identified in this section.

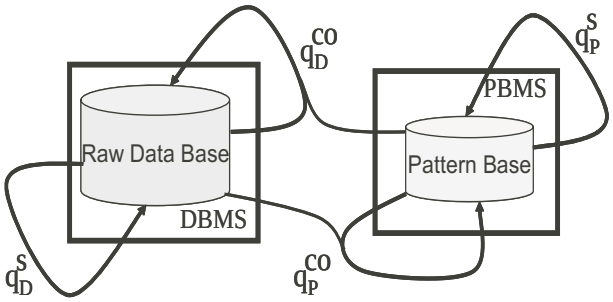


Figure 5.1: *D-queries* and *P-queries*

5.2 Pattern Query Languages for $\mathcal{EPM}$

In this section, we present two pattern query languages for $\mathcal{EPM}$. As usual in a database context, we focus on two different paradigms of query languages: a procedural one - i.e., an algebra - and a declarative one - i.e., a calculus. Due to their formal semantics and generality, the proposed algebraic operators can be used as the basis for the development of optimization techniques, based on query rewriting, for pattern query processing. On the other side, the calculus approach can be efficiently exploited when developing a user-friendly practical query language, since it does not require the user to know the internal system working. Both languages provide all the functionalities pointed out in Section 3.4 of Chapter 3. In Chapter 6, we will show that, under specific hypothesis, these languages are equivalent.

The proposed query languages provide querying features supporting not only the specification of selection conditions over patterns, but also the capability to combine patterns (possibly of different types) together and other advanced querying capabilities, by capitalizing on the peculiar semantic features of $\mathcal{EPM}$. More precisely, they support pattern validity analysis (from both a temporal and a semantic point of view), based on the notions introduced in Chapter 4, and hierarchy navigation, allowing the user to move along the refinement and the composition pattern hierarchies. Finally, they support both, cross-over D- and P- queries, as discussed in Section 5.1.

In the remainder of this section, we first introduce the predicates defined over patterns and pattern components (Section 5.2.1). Such predicates will then be used in defining the *Extended Pattern Query Algebra*, $\mathcal{E}APQL$, (Section 5.2.2) and the *Extended Pattern Query Calculus*, $\mathcal{E}CPQL$, (Section 5.2.3).

5.2.1 Pattern Predicates

In the remainder of this section, we first introduce predicates dealing with pattern components (Section 5.2.1.1) and, then, several other predicates defined over entire patterns are presented (Section 5.2.1.2). Some of the presented predicates are cross-over, since for their evaluation raw data access is required.

5.2.1.1 Predicates Dealing with Pattern Components

In this section predicates dealing with pattern components are introduced. For each predicate, we point out whether it is a cross-over one or a non cross-over predicate. Information concerning these predicates are summarized in Table 5.1.

Extensional source containment. It checks the containment between pattern data sources in an extensional way, i.e., by checking the containment of the datasets resulting from the execution of the $\mathcal{Q}$-queries used to describe the pattern data sources. It is denoted with the symbol $\subseteq^e$. More precisely, given two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ pattern data sources d_1 and d_2 of type t_d, $d_1 \subseteq^e d_2$ is true if and only if $d_1(UDB) \subseteq d_2(UDB)$, where $d_i(UDB)$ represents the dataset obtained by evaluating the query corresponding to d_i over the Universal Database (UDB). We notice that it is a cross-over predicate.

Intensional source containment. It checks the containment between pattern data sources, in an intensional way, i.e., by considering query definitions. It is denoted with the symbol $\subseteq^i$. In details, given two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ pattern data sources d_1 and d_2 of type t_d, $d_1 \subseteq^i d_2$ is true if and only if $d_1 \subseteq d_2$, where $\subseteq$ denotes the containment predicate for $\mathcal{Q}$-queries defined as follows.

Definition 34 (*$\mathcal{Q}$-query containment*) Given two $\mathcal{Q}$-queries Q_1 and Q_2, we say that Q_2 is contained in to Q_1, denoted by $Q_2 \subseteq Q_1$, if for any database D $Q_2(D) \subseteq Q_1(D)$, where $Q_i(D)$ denotes the dataset obtained by executing Q_i over D. □

Since intensional containment predicate acts over $\mathcal{Q}$-queries describing pattern data sources, it does not require any access to raw data; thus, it is not a cross-over predicate.

Extensional source equivalence. It checks the equivalence between pattern data sources in an extensional way, i.e., by checking the equivalence of the datasets resulting by the execution of the $\mathcal{Q}$-queries used to describe the pattern data sources. It is denoted with the symbol $=^e$. More formally, given two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ pattern data sources d_1 and d_2 of type t_d, $d_1 =^e d_2$ is true if and only if $d_1(UDB) = d_2(UDB)$. We notice that it is a cross-over predicate.

Intensional source equivalence. It checks the equivalence between pattern data sources in an intensional way, i.e., by considering query definition. It is denoted with the symbol $=^i$. Thus, given two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ pattern data sources d_1 and d_2 of type t_d, $d_1 =^i d_2$ is true if and only if intensional equivalence between $\mathcal{Q}$-queries describing them holds, i.e., $d_1 \equiv d_2$, where $\equiv$ denotes the equivalence predicate for $\mathcal{Q}$-queries defined as follows.

Definition 35 (*$\mathcal{Q}$-query equivalence*) Given two $\mathcal{Q}$-queries Q_1 and Q_2, we say that Q_1 is equivalent to Q_2, denoted by $Q_1 \equiv Q_2$, if and only if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. □

As in the case of the intensional containment predicate, it is a non cross-over predicate.

Pattern formula satisfaction. It checks whether the evaluation of a formula of the theory $\mathcal{F}$ associated with a pattern p of type pt in $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ over a certain data item returns true. More precisely, let $r \in dom(pt.d)$, $p.f(r)$ is true if and only if $p.f([r/x])$ evaluates to true.

Extensional pattern formula containment. It checks the containment between two pattern formulas in an extensional way. It is denoted with the symbol $\preceq^e$. Since it exploits an extensional approach, it consists in the evaluation of the containment of the two datasets containing raw data effectively represented by the patterns. Those datasets are subset of the pattern data sources to which the predicate is applied. Such subsets are obtained by evaluating each pattern formula over the corresponding pattern data source, as it were a query condition. More formally, given two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ pattern formulas f_1 and f_2 associated with patterns p_1 and p_2, respectively, $f_1 \preceq^e f_2$ is true if and only if $\{x | x \in p_1.d(UDB) \wedge f_1(x)\} \subseteq \{y | y \in p_2.d(UDB) \wedge f_2(y)\}$. Since it requires to execute two queries over pattern data sources, it is a cross-over predicate.

Intensional pattern formula containment. It checks the containment between two pattern formulas in an intensional way. It is denoted by the symbol $\preceq$ and it relies on logical implication between formulas in the logical theory $\mathcal{F}$. More formally, given two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ pattern formulas f_1 and f_2 associated with patterns p_1 and p_2, respectively, $f_1 \preceq f_2$ if and only if f_1 logically implies f_2 (i.e., $f_1 \rightarrow f_2$) in the context of the logical theory $\mathcal{F}$. It is quite obvious that it is a non cross-over predicate.

Extensional pattern formula equivalence. It checks the equivalence between pattern formulas in an extensional way, i.e., by checking the equality of the two datasets containing data in the pattern data sources satisfying the input formulas. It is denoted with the symbol $\equiv^e$. More formally, given two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ pattern formulas f_1 and f_2, $f_1 \equiv^e f_2$ is true if and only if $\{x | x \in p_1.d(UDB) \wedge f_1(x)\} = \{y | y \in$

$p_2.d(UDB) \wedge f_2(y)\}$ Since it requires to execute two queries over pattern data sources, it is a cross-over predicate.

Intensional pattern formula equivalence. It checks the equivalence between two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ pattern formulas in an intensional way. It is denoted by the symbol $\equiv^i$. Since pattern instantiated formulas are logical formulas of a certain theory $\mathcal{F}$, it relies on the logical equivalence defined for the logical theory $\mathcal{F}$. In details, given two $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ pattern formulas f_1 and f_2, $f_1 \equiv^i f_2$ is true if and only if f_1 is equivalent to f_2 (denoted by $f_1 \equiv f_2$) within the logical theory $\mathcal{F}$. It is a non cross-over predicates.

Formula satisfiability. It checks the satisfiability of pattern formulas according to the chosen logical theory $\mathcal{F}$ for $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ is provided. It is denoted by the symbol $\Downarrow$. We remark that this predicate relies on the satisfiability problem (SAT) for logical theories.

Temporal predicates Since $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ is parametric with respect to the temporal theory $\mathcal{V}$ chosen for representing pattern validity information, predicates defined over pattern validity period components are the ones offered by the temporal theory chosen in the pattern type definition.

For example, if $\mathcal{V}$ is the time interval theory T^{I} over $\mathbb{Z}$ (see Chapter 6 Section 6.2.2 for more details) pattern validity periods are closed interval over $\mathbb{Z}$ and predicates defined for intervals - such as equals, overlaps, meets, before - are provided [Sno95].

All the previous predicates deal with specific $\mathcal{EPM}$ pattern components. However, since, according to $\mathcal{EPM}$, pattern components may have a complex value type (e.g. the pattern structure and the pattern measure) the following additional predicates for complex values are provided, with the proper type restriction: (i) $=$(equality); (ii) $\in$(membership); (iii) $\subseteq$(set containment). We notice that, in presence of hierarchies, two versions of each of the previous predicates can be defined, one based on shallow equality and one based on deep equality. In the following, we use the supscript ^{s} or ^{d} to remark the type of equality we want to consider.

5.2.1.2 Predicates Dealing with Patterns

Besides the previous predicates over pattern components, several other predicates over patterns are supported. Details concerning these predicates are presented in the following and summarized in Table 5.2.

Identity. It checks whether two patterns are identical or not, according to the identity notion presented in Def. 11. It is a non cross-over predicate.

Shallow equality. It checks whether two patterns are shallow equal or not, according to Def. 12. It is a non cross-over predicate.

Deep equality. It checks whether two patterns are deep equal or not. It is a non cross-over predicate.

Weak subsumption. It checks whether a pattern p_1 weakly subsumes a pattern p_2, denoted by $p_1 \prec^w p_2$, i.e., if p_1 and p_2 have the same structure but p_1 represents a smaller set of data than p_2. More formally, let p_1 and p_2 two patterns defined according to $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, $p_1 \prec^w p_2$ is true if and only if the following conditions hold: (i) $p_1.s = p_2.s$; (ii) $p_1.d \subseteq^i p_2.d$; (iii) $p_1.f \preceq p_2.f$ within the logical theory $\mathcal{F}$. It is a non cross-over predicate.

Subsumption. It checks whether a pattern p_1 subsumes another pattern p_2, denoted by $p_1 \prec p_2$, i.e., checks whether p_1 and p_2 have the same structure but p_1 represents a smaller set of data than p_2, through their instantiated formulas. More formally, let p_1 and p_2 two patterns defined according to $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, $p_1 \prec p_2$ is true if and only if $p_1.s = p_2.s$ and $p_1.d_{\lceil p_1.f} \subseteq p_2.d_{\lceil p_2.f}$, where $d_{\lceil f}$ represents the set of source data items d satisfying the formula f. Since it requires the evaluation of the pattern instantiated formulas over the pattern data sources, it is a cross-over predicate.

Goodness. It checks whether a pattern p_1 is better of than a pattern p_2, having the same pattern type, with respect to their measures. It is denoted by the symbol $\nearrow$. More formally, $p_1 \nearrow p_2$ is true if and only if the following conditions hold: (i) p_1 and p_2 are two patterns of type pt defined according to $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$; (ii) $p_1 \prec^w p_2$; (iii) p_1 measures are better than p_2 measures, i.e., assuming that $pt.m = \langle m_1, ..., m_n \rangle$, $p_1.m_i \theta_i p_2.m_i$, $i = 1, ..., n$. Since $\nearrow$ relies on the weak subsumption predicate, which is a cross-over predicate, it is a cross-over predicate too.

Temporal validity. It checks whether a pattern p is temporally valid at a certain instant of time t. It is denoted by the symbol ω_T. Assuming p_1 is a pattern defined according to $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ and t is an instant of time expressed according to the temporal theory $\mathcal{V}$, $\omega_T(p_1, t)$ is true if and only if p_1 is temporally valid at time t, according to Def 28 (Section 4.4), i.e., t satisfies $p_1.v$ in $\mathcal{V}$. It is a non cross-over predicate.[1]

Semantic validity. It checks whether a certain pattern p_1 of type pt is semantically valid over a certain data source D, with respect to a some given thresholds $v_1, ..., v_n$, according to Def. 30 (Section 4.4). It is denoted with the symbol ω_S. Since ω_S involves a mining phase, which in general is performed over raw data, it can be

[1]We notice that two versions of this predicate are available: a first one checking for shallow pattern temporal validity, denoted by ω_T^S, and a second one checking for deep pattern temporal equality, denoted by ω_T^D. Whenever no otherwise specified, we refer with ω_T to the shallow version of the predicate, i.e. $\omega_T \equiv \omega_T^S$.

considered a cross-over predicate. However, it may be possible the mining is applied over patterns, whenever a refinement relationship is exploited in *pt* definition. In this last case, it is a non cross-over predicate, since the entire mining process is performed within the PBMS without accessing the underlying DBMS.

Composition check. It checks whether exists a composition hierarchy between two patterns, according to Def. 22. It is denoted with the symbol $\hookrightarrow$. it is a non-cross over predicate.

Refinement check. It checks whether exists a refinement hierarchy between two patterns, according to Def. 23. It is denoted with the symbol $\Rightarrow$. it is a non-cross over predicate.

5.2.2 $\mathcal{E}APQL$: an Extended Pattern Query Algebra

In this section, a procedural query language for $\mathcal{EPM}$ patterns, called Extended Algebraic Pattern Query Language ($\mathcal{E}APQL$) is introduced. We start by presenting an algebraic approach to pattern querying since it is quite intuitive and it allows us to focus on the semantics of each querying operator.

The proposed algebraic operators apply to pattern classes. Some of them, like set-based operators and selection, are quite close to their relational counterparts; others have been introduced in order to deal with complex values [AB95] (such as tuple and set constructors and destructors, nest and unnest). Nevertheless, some others, like join and projection, extend their traditional counterparts, in order to deal with patterns. They are, therefore, significantly different with respect to their relational or complex value version. We notice that, whenever a new pattern is created as result of an algebraic operation, a new $pid \in OID$ is generated for it by the function $newid()$.

Set based operators Since classes are sets of patterns, usual set operators such as *union* ($\cup$), *difference* ($\setminus$), and *intersection* ($\cap$) are defined for pairs of classes defined over the same pattern type. They take as input two pattern classes defined for a pattern type *pt* and they output a collection of patterns of the same type. Since, the criterion for equality/identity of the patterns can be any of the ones previously discussed in Section 4.2.2, to be more precise we denote with $\cup^{eq}$, $\setminus^{eq}$, and $\cap^{eq}$ the set-based operator versions relying on pattern equality notion (see Def. 12) and with $\cup^{id}$, $\setminus^{id}$, and $\cap^{id}$ the set-based operator versions relying on pattern identity notion (see Def. 11).

Selection ($\sigma_F(c)$) The selection operator allows one to select from a given class c the patterns satisfying a certain selection condition F. The pattern type of the output

Name	Type[a]	Symbol	Description	Note
Extensional source containment	$d_1 :: t_d$, $d_2 :: t_d$	$\subseteq^e$	$d_1 \subseteq^e d_2$ is true iff $d_1(UDB) \subseteq d_2(UDB)$	cross-over
Intensional source containment	$d_1 :: t_d$, $d_2 :: t_d$	$\subseteq^i$	$d_1 \subseteq^i d_2$ is true iff $d_1 \subseteq d_2$	non cross-over
Extensional source equivalence	$d_1 :: t_d$, $d_2 :: t_d$	$=^e$	$d_1 =^e d_2$ is true iff $d_1(UDB) = d_2(UDB)$	cross-over
Intensional source equivalence	$d_1 :: t_d$, $d_2 :: t_d$	$=^i$	$d_1 =^i d_2$ is true iff $d_1(UDB) \equiv d_2(UDB)$	non cross-over
Pattern formula satisfaction	$f :: t_f$	.f(r)	$p.f(r)$ is true iff $p.f[r/x]$, where x is the free variable in $p.f$ and $r \in dom(pt.d)$, is true.	non cross-over
Extensional pattern formula containment	$f_1 :: t_f$, $f_2 :: t_f$	$\preceq^e$	$f_1 \preceq^e f_2$ is true iff $\{x \mid x \in p_1.d(UDB) \wedge f_1(x)\} \subseteq \{y \mid y \in p_2.d(UDB) \wedge f_2(y)\}$	cross-over
Intensional pattern formula containment	$f_1 :: t_f$, $f_2 :: t_f$	$\preceq$	$f_1 \preceq f_2$ is true iff $f_1 \rightarrow f_2$ within $\mathcal{F}$	non cross-over
Extensional pattern formula equivalence	$f_1 :: t_f$, $f_2 :: t_f$	$\equiv^e$	$f_1 \equiv^e f_2$ is true iff $\{x \mid x \in p_1.d(UDB) \wedge f_1(x)\} = \{y \mid y \in p_2.d(UDB) \wedge f_2(y)\}$	cross-over
Intensional pattern formula equivalence	$f_1 :: t_f$, $f_2 :: t_f$	$\equiv^i$	$f_1 \equiv^i f_2$ is true iff $f_1 \equiv f_2$ within $\mathcal{F}$	non cross-over
Formula satisfiability	$f :: t_f$	$\Downarrow$	$\Downarrow f$ is true iff f is logical formula and $SAT(f)$	non cross-over
Temporal predicates	$v_1 :: \mathcal{V}$, $v_2 :: \mathcal{V}$	θ_T	$v_1 \theta_T v_2$ is true iff $v_1 \theta_T v_2$ evaluates to true in $\mathcal{V}$	non cross-over
Complex value equality	$e_1 :: t$, $e_2 :: t$	$=$	$e_1 = e_2$ is true iff complex values e_1 and e_2 are equal	it depends on the definition of e_1 and e_2
Complex value membership	$e_1 :: t$, $e_2 :: SET(t)$	$\in$	$e_1 \in e_2$ is true iff complex value e_1 belong to the complex object e_2	it depends on the definition of e_1 and e_2
Complex value set-containment	$e_1 :: SET(t)$, $e_2 :: SET(t)$	$\subseteq$	$e_1 \subseteq e_2$ is true iff complex value e_1 is contained in e_2	it depends on the definition of e_1 and e_2

Legenda:
t_d: type allowed for data sources
[a] t_f: type allowed for formulas
$\mathcal{V}$: temporal theory
$t \in \mathcal{T}$

Table 5.1: Predicates involving pattern components

patterns is the same of the input class c. Whenever no patterns in c satisfy F, the output is the empty class. Selection condition F is constructed starting from atomic formulas, which can be constructed by using predicates introduced in Section 5.2.1 and combining them using boolean operators, such as $\wedge$, $\vee$, or $\neg$. Allowed terms are constants of the proper type and path expressions (according to Def. 13), starting from a variable p representing the generic pattern belonging to the input class.

Therefore, let c be a class of pattern type $pt \in \mathcal{PT}$ for $\mathcal{EPM}$ and let F be a selection condition as defined above. The selection operator applied over c is defined as follows:

$$\sigma_F(c) = \{\overline{p} | \overline{p} \in c \wedge F[p/\overline{p}])\}$$

where $F[p/\overline{p}]$ represents F in which p has been replaced by $\overline{p}$.

Name	Symbol	Description	Note
Identity	$=$	$p_1 = p_2$ is true iff $p_1.pid = p_2.pid$, according to Def 11(Section 4.2.2)	non cross-over
Shallow equality	$=^s$	$p_1 =^s p_2$ is true iff $p_1.s = p_2.s$ and $p_1.d =^i p_2.d$ and $p_1.m = p_2.m$ and $p_1.f \equiv^i p_2.f$	non cross-over, since it uses the intensional pattern formula equivalence predicate
Deep equality	$=^d$	$p_1 =^d p_2$ is true iff $p_1 =^s p_2$ and $\forall \overline{p_1}$ s.t. $p_1 \rightarrowtail \overline{p_1}$, $\forall \overline{p_2}$ s.t. $p_2 \rightarrowtail \overline{p_2} : \overline{p_1} =^d \overline{p_2}$	non cross-over, since it uses the intensional pattern formula equivalence predicate
Weak subsumption	$\prec^w$	$p_1 \prec^w p_2$ is true iff $p_1.s = p_2.s$ and $p_1.d \subseteq^i p_2.d$ and $p_1.f_1 \prec p_2.f_2$ within $\mathcal{F}$	non cross-over
Subsumption	$\prec$	$p_1 \prec p_2$ is true iff $p_1.s = p_2.s$ and $p_1.d_{\lceil p_1.f} \subseteq p_2.d_{\lceil p_2.f}$, where $d_{\lceil f}$ represents the set of source data items d satisfying the formula f	cross-over
Goodness	$\nearrow$	$p_1 \nearrow p_2$ is true iff $p_1 \prec^w p_2$ and, assuming that $pt.m = \langle m_1, ..., m_n \rangle$, $p_1.m_i \theta_i p_2.m_i$, $i = 1, ..., n$, where θ_i is the 'better of' predicate over pt measures	cross-over
Temporal validity	ω_T	$\omega_T(p_1, t)$ is true iff t is an instant of time expressed according to $\mathcal{V}$, and p_1 is temporally valid at time t according to Def. 28(Section 4.4)	non cross-over
Semantic validity	ω_S	$\omega_S(p_1, D, < v_1, ..., v_n >)$ is true iff p_1 is semantically valid with respect to D and $v_1, ..., v_n$, according to Def. 30(Section 4.4)	cross-over (whenever the pattern type of p_1 does not exploit refinement) non cross-over (whenever in the pattern type of p_1 a refinement is exploited)
Composition check	$\hookrightarrow$	$p_1 \hookrightarrow p_2$ is true iff $p_1 \rightarrowtail_s p_2$ according to Def. 22)	non cross-over
Refinement check	$\Rightarrow$	$p_1 \hookrightarrow p_2$ is true iff $p_1 \rightarrowtail_d p_2$ according to Def. 23)	non cross-over

Table 5.2: Predicates involving patterns defined according to $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$

Measure projection ($\pi^m_{lm}(c)$) The measure projection operator over a class c, defined for pt, reduces the measures of the input patterns by projecting out some measure components, according to the projection list lm. Such an operation is useful whenever the user is interested in pattern processing not involving some of the pattern measures. For instance, when dealing with association rules, an interesting study concerning the relative weight of rules with respect to the observed transactions would focus on the support measure rather than on the confidence measure. In such cases, the user may restrict her attention on association rules support, projecting out the confidence component of the pattern measure. Note that no modification is required for the pattern formula and for the validity period of the pattern, since measures do not appear in them.

The measure projection operation produces as output patterns of a new pattern type pt' which has all components, except the measure one, equal to pt. The measure component of pt' is obtained from the one in pt by removing all components do not appearing in lm. More formally, let $pt = TUPLE(s : t_s, d : t_d, m : TUPLE(m1 : t_{m_1}, ..., m_k : t_{m_k}), f : t_f, v : \mathcal{V})$. Then, $pt' = TUPLE(s : t_s, d : t_d, m : t'_m, f : t_f, v : \mathcal{V})$, where $t'_m \in \mathcal{T}$ and $t'_m = TUPLE(m_1 : t_{m_1}, ..., m_h : t_{m_h})$ and $lm = \{m_1, ..., m_h\}$.

Let c be a class of pattern type pt and lm be a list of attributes appearing in $pt.m$, the measure projection operator is formally defined as follows:

$$\pi^m_{lm}(c) = \{< newid(), s, d, \pi_{lm}(m), f, v > | \exists p\, c(p)\ s.t.\ (p =< pid, s, d, m, f, v >)\}$$

In the previous definition, $\pi_{lm}(m)$ is the usual relational projection applied over the measure component, which is a tuple.

We outline that no projection has been defined over the data source, since in this case the structure and the measures would have to be recomputed and therefore, the pattern will change. On the other side, projection of the pattern structure has to be treated in a special way, due to the fact that the type of the structure component is not predefined. This motivates the introduction of a dedicated reconstruction algebraic operator, defined as follows.

Reconstruction ($\lambda_q(c)$) The reconstruction operator allows us to manipulate the pattern structure of the patterns belonging to a class c, defined for the pattern type pt, according to an input query q. This operation is useful for example for projecting out some components from the structure or for nesting or unnesting data, or even to select patterns with respect to inner components of their structure.

Since the type of the pattern structure is any type admitted by the type system, we define reconstruction based on a query over the pattern structure, defined by applying complex value algebraic operators similar to the ones defined in [AB95], extended with a dereferencing operator, in order to deal with pattern references (not allowed in [AB95]).

More formally, let c be a class of pattern type pt. Let $I, I_1, I_2, ...$ be sets of values of type $\tau, \tau_1, \tau_2, ...$ Let q be a query constructed by using the following operators defined according to [AB95].

Projection When dealing with tuple elements we may be interested in projecting out some attributes of the tuple. Thus, a *projection* operation is considered.

- Let I be a set of type $\tau =< B_1 : \tau_1, ..., B_k : \tau_k >$. Then, $\pi_{B_1,...,B_l}(I)$ is a set of type $< B_1 : \tau_1, ..., B_l : \tau_l >$ such that $\pi_{B_1,...,B_l}(I) = \{< B_1 : v_1, ..., B_l : v_l > | \exists v_{l+1}, ..., v_k$ such that $< B_1 : v_1, ..., B_l : v_l, B_{l+1} : v_{l+1}, ..., B_k : v_k > \in I\}$

Dereferencing operator Whenever a composition relationship is exploited we may deal with tuple elements having some attributes of type OID, representing pattern references. In such cases, we may be interested in dereferencing the identifier, in order to get the corresponding value.

- Let I be a set of type $\tau =< B_1 : \tau_1, ..., B_k : \tau_k, C_1 : ref(\sigma_1), ..., C_h : ref(\sigma_h) >$. Then, $\gg_{C_1,...,C_j}(I)$ is a set of type $< B_1 : \tau_1, ..., B_k : \tau_k, C_1 : \sigma_1, ..., C_j : \sigma_j >$ such that $\gg_{C_1,...,C_j}(I) = \{< B_1 : v_1, ..., B_k : v_k, C_1 : (\gg$

$w_1), ..., C_j : (\gg w_j) > | \exists w_{j+1}, ..., w_h \in \mathcal{OID}$ such that $< B_1 : v_1, ..., B_k : v_k, C_1 : w_1, ..., C_j : w_j, C_{j+1} : w_j, ..., C_h : w_h > \in I\}$

Constructors We may be interested in constructing complex elements starting from simple ones. Thus, we consider the following operations, *tup_create* and *set_create*. Let I be a set of type τ.

- If $A_1, ..., A_n$ are distinct attributes, $tup\_create_{A_1,...,A_m}(I_1, ..., I_m)$ is of type $< A_1 : \tau_1, ..., A_m : \tau_m >$ and $tup\_create_{A_1,...,A_m}(I_1, ..., I_m) = \{< A_1 : v_1, ..., A_m : v_m >, i = 1, ..., m\}$.
- $set\_create(I)$ is a set of type $\{\tau\}$ and $set\_create(I) = \{I\}$.

Destructors We may be interested in elements inside a complex element. Thus, we consider two destructor operations, *tup_destroy* and *set_destroy*.

- If $\tau = \{\tau_1\}$, then $set\_destroy(I)$ is a set of type τ_1 and $set\_destroy(I) = \{w | \exists v \in I, w \in v\}$.
- If I is of type $< A : \tau_1 >$, $tup\_destroy(I)$ is a set of type τ_1 and $tup\_destroy(I) = \{v | < A : v > \in I\}$.

Nest and unnest Let I_1 be a set of type t_1 such that $t_1 =< A_1 : \tau_1, ..., A_k : \tau_k, B : \{< A_{k+1} : \tau_{k+1}, ..., A_n : \tau_n >\} >$. Let I_2 be a set of type t_2 such that $t_2 =< A_1 : \tau_1, ..., A_k : \tau_k, A_{k+1} : \tau_{k+1}, ..., A_n : \tau_n >$. Then,

- $unnest_B(I_1)$ is a set of type t_2 and $unnest_B(I_1) = \{< A_1 : v_1, ..., A_n : v_n > | \exists y < A_1 : v_1, ..., A_k : v_k, B : y > \in I_1 \wedge < A_{k+1} : v_{k+1}, ..., A_n : v_n > \in y\}$;
- $nest_{B=(A_{k+1},...,A_n)}(I_2)$ is a set of type t_1 and $nest_{B=(A_{k+1},...,A_n)}(I_2) = \{< A_1 : v_1, ..., A_k : v_k, B : \{< A_{k+1} : v_{k+1}, ..., A_n : v_n >\} > | < A_1 : v_1, ..., A_n : v_n > \in I_2\}$.

At this point, the reconstruction operator is defined as follows:

$$\lambda_q(c) = \{< pid, q(s), d, m, f, v > | p =< pid, s, d, m, f, v > \in c\}.$$

Note that the formula of the resulting patterns has not to be updated since the only variables appearing in it come from the data source, which is not changed by the operation.

Assuming $pt = TUPLE(s : t_s, d : t_d, m : t_m, f : t_f, vp : \mathcal{V})$ is the pattern type over which c is defined, $\lambda_q(c)$ produces as output patterns of pattern type $pt' = TUPLE(s : t'_s, d : t_d, m : t_m, f : t_f, vp : \mathcal{V})$ where t'_s is the type of the result of q.

Example 33 *Let $AR1$ be a class defined for the pattern type* `AssociationRule`. *Assume you want to obtain new patterns containing only the head as structure component and support as measure starting from association rules belonging to class $AR1$*

having at least confidence 0.7 *and whose body contains 'Bread'. The following is an algebraic query specifying the previous request:*

$$\lambda_{\pi_{p.s.head}}((\pi^{m}_{support}(\sigma_{'Bread' \in p.s.body \wedge p.m.confidence > 0.75}(AR1))))$$

□

Join $(C_1 \bowtie_{F,c} C_2)$ The join operation provides a way to combine patterns belonging to two different classes, C_1 and C_2, according to a join predicate F and a composition or joining function j, that defines the pattern type of the result of the join operation, specified by the user.

More precisely, let c_1 and c_2 be two classes over two pattern types $pt_1 = TUPLE(s_1, d_1, m_1, f_1, v_1)$ and $pt_2 = TUPLE(s_2, d_2, m_2, f_2, v_2)$. A join predicate F is a condition using any predicate defined over a component of patterns in c_1 and a component of patterns in c_2 (see Section 5.2.1). A join composition function j for pattern types pt_1 and pt_2 is a 5-tuple of functions $j = (j_s, j_d, j_m, j_f, j_v)$, one for each pattern component. For example, function j_s takes as input two structure values of the correct 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 $p_1 =< pid_1, s_1, d_1, m_1, v_1 > \in c_1$ and $p_2 =< pid_2, s_2, d_2, m_2, v_2 > \in c_2$, $j(p_1, p_2)$ is defined as the pattern $p =< pid_p, s_p, d_p, m_p, v_p >$ instance of the pattern type pt_{join} defined as follows $pt_{join} = TUPLE(j_s(s_1, s_2), j_d(d_1, d_2), j_m(m_1, m_2), j_f(f_1, f_2), j_v(v_1, v_2))$.

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

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

The general definition of join can be customized by considering specific composition operators having a predefined semantics. For instance, it can be useful to construct new patterns referring to the union of two distinct pattern data sources. Such new patterns represent data in at least one of the two data sources. The special join operation implementing this semantics is called *union join* ($\bowtie^{\cup}$). In some sense, patterns generated by the union join can be viewed as a 'relaxation' of the input patterns. At the same way, it could be useful to generate new patterns representing data belonging to the intersection of the data sources of the input patterns. The *intersection join* ($\bowtie^{\cap}$) operation has this behavior. Therefore, in some sense, patterns resulting by an intersection join application can be considered a 'restriction' of the input ones.

The specific composition functions used to define the union and intersection joins can be defined as follows.

union join $c_1 \bowtie^{\cup}_{F,j} c_2$, with joining function j formed by the following components:
$j_{pid} = newpid()$
$j_s = tup\_create_{t^1_s, t^2_s}(p_1.s, p_2.s)$[2]
$j_d = p_1.d \cup p_2.d$
$j_m = tup\_create_{t^1_m, t^2_m}(p_1.m, p_2.m)$[3]
$j_f = p_1.f \vee p_2.f$
$j_v = p_1.v \cup p_2.v$

intersection join $c_1 \bowtie^{\cap}_{F,j} c_2$ with joining function j formed by the following components :
$j_{pid} = newpid()$
$j_s = tup\_create_{t^1_s, t^2_s}(p_1.s, p_2.s)$
$j_d = p_1.d \cap p_2.d$
$j_m = tup\_create_{t^1_m, t^2_m}(p_1.m, p_2.m)$
$j_f = p_1.f \wedge p_2.f$
$j_v = p_1.v \cap p_2.v$

Example 34 *Consider two classes c_1 and c_2 defined for the pattern type* `AssociationRules` *(see Example 19). Suppose we want to generate new association rules by transitive closure, i.e., informally, given two rules $A \rightarrow B \in c_1$ and $B \rightarrow C \in c_2$, we want to generate rule $A \rightarrow C$. Such rules can be generated through a join operation $c_1 \bowtie_{F,j} c_2$ applied to c_1 and c_2 with join predicate $F \equiv c_1.s.body = c_2.s.head$ and using a specific join function j. A reasonable join function generates the new structures as described above. Data sources can be combined by considering the natural join between the input sources since the new rule will refer only to the tuples belonging to both datasets. Null values can be assigned to measures if we do not want to recompute them on the new dataset. Finally, expressions can be combined by considering the intersection of the input ones. The resulting patterns are still of type* `AssociationRule`. □

Decomposition. The decomposition operator allows one to navigate the composition relationship, returning for each pattern the component patterns having a given pattern type. More formally, consider two pattern types pt_1,$pt_2 \in \mathcal{PT}$ such that $pt_1 \hookrightarrow pt_2$, let c be a class over pattern type pt_1. The decomposition operation is denoted by $\mathcal{C}_{pt_2}(c)$ and it is formally defined as follows:
$\mathcal{C}_{pt_2}(c) = \{p | \exists p'\ p' \in c \wedge p' \hookrightarrow p \wedge p \in ext(pt_2)\}$.

Example 35 *Let CR be a class over pattern type* `ClusterOfRules` *defined in Example 26. The execution of the operation $\mathcal{C}_{AssociationRule}(CR)$ returns a set of patterns*

[2]Note that, $p_1 \in dom(pt_1)$, thus t^1_s is the type of $pt_1.s$ and $p_2 \in dom(pt_2)$, thus t^2_s is the type of $pt_2.s$.
[3]Similarly to the case of the structure component, t^1_m is the type of $pt_1.m$ and t^2_m is the type of $pt_2.m$.

of pattern type `AssociationRule` *containing the representative rule of each cluster of rules in* CR. □

Drill-Down. The drill-down operator allows one to navigate the refinement relationship between patterns, from a pattern to some of the patterns it refines. More formally, consider two pattern types pt_1,$pt_2 \in \mathcal{PT}$ such that $pt_1 \Rightarrow pt_2$, let c be a class over pattern type pt_1. The drill-down operation, denoted by $\delta_{pt_2}(c)$, returns the following set:
$\delta_{pt_2}(c) = \{p | \exists p'\, p' \in c \wedge p' \Rightarrow p \wedge p \in ext(pt_2)\}$.

Roll-Up. The roll-up operator allows one to navigate the refinement relationship between patterns, from a pattern p to some of the patterns obtained by refining p. More formally, consider two pattern types pt_1,$pt_2 \in \mathcal{PT}$ such that $pt_1 \Rightarrow pt_2$, let c be a class over pattern type pt_1. The roll-up operator, when applied to c, returns the pattern instances of pattern type pt_2 refining at least one pattern belonging to c. The roll-up operation is formally defined as follows:
$\rho_{pt_1}(c) = \{p | \exists p'\, p' \in c \wedge p \Rightarrow p' \wedge p \in ext(pt_1)\}$.

Example 36 *Consider the refinement relationship existing between* `ClusterOfRules` *and* `AssociationRule` *(see Example 26). It is possible to navigate from patterns of type* `ClusterOfRules` *to the correlate instances of the pattern type* AssociationRule *and vice versa. Let* CR *be a class of type* `ClusterOfRules` *and let* AR *be a class of type* `AssociationRule`.

- $\delta_{AssociationRule}(CR)$ *returns a set of association rules refined by clusters in* CR;
- $\rho_{\mathsf{ClusterOfRules}}(AR)$ *returns a set of patterns of type* `ClusterOfRules` *refining at least one pattern in* AR. □

Drill-through ($\gamma(c)$) The drill-through operator allows one to navigate from a pattern to its data source. Such operator, thus, takes as input a pattern class c, defined for the pattern type pt, and returns a raw data set or a pattern set, depending on whether the patterns are hierarchical or not. The type of the output is the type of the result of the query intensionally describing the data sources of the patterns in the input class c.

More formally, let c be a class of pattern type pt which contains patterns $p_1, \ldots, p_n$. Then, the drill-through operator, denoted by $\gamma(c)$, is formally defined as follows:

$$\gamma(c) = \bigcup_{p \in c} p.d(UDB).$$

Example 37 *Let CR be a class of type* `ClusterOfRules`. *The expression $\gamma(CR)$ returns a set of association rules represented by at least one cluster in CR; $\gamma(\gamma(CR))$ returns a set of transactions represented by at least an association rule in the result set of $\gamma(CR)$.* □

Data covering ($\theta_d(c, D)$) Given a pattern p and a dataset D, sometimes it is important to determine whether p represents D or not. In other words, we wish to determine the subset S of D represented by p.[4]

Extended to a class c, this operation corresponds to determining the subset S of D containing elements (raw data or patterns) represented by at least one pattern in c. To determine S, we use the pattern formula as a query over the dataset D.

Let c be a class, defined for the pattern type pt, and D a dataset with schema $t_D \in \mathcal{T}_e$, compatible with the type of the data source of patterns of type pt. The data covering operator, denoted by $\theta_d(c, D)$, returns a new dataset corresponding to all elements in D represented by a pattern p in c. More formally:

$$\theta_d(c, D) = \{e \mid e \in D \wedge \exists p \in c \; p.f(e)\}$$

We notice that there are two main differences between the drill-through operator and the data covering one. Both operators provides a way to navigate from the pattern space to the data space; however, the drill-through operator is based on data source information (thus, it determines the input data set the pattern is associated with) whereas the data covering operator is based on the formula (thus, it determines the set of elements represented by a pattern). Moreover, the data covering operator selects patterns representing a give set of data items; on the other hand, the drill-through operator returns the overall input data source.

Pattern covering ($\theta_p(c, D)$) Sometimes it can be useful to have an operator that, given a class c and a dataset D, returns all patterns in c representing D (a sort of inverse data covering operation).

More formally, let c be a pattern class defined for pattern type pt and D a dataset with schema $t_P \in \mathcal{PT}$ compatible with the type of the data source of patterns of type pt. The pattern_covering operator, denoted by $\theta_p(c, D)$, returns a set of patterns corresponding to all patterns in c representing D and it is defined as follows:

$$\theta_p(c, D) = \{p | p \in c \wedge \forall t \in D \; p.f(t)\}.$$

[4]We remark that in real implementations of the algebra, the input dataset in the data covering and pattern covering operations can be intensionally described through a query. We did not formalize this aspect for the sake of simplicity.

It is easy to show that pattern covering is a derived operation, as pointed out in the following proposition.

Proposition 1 $\theta_p(c, D) = \{p \mid p \in c \wedge \theta_d(\{p\}, D) = D\}$

Example 38 *Consider the class $AR1$ of type* AssociationRule *(see Example 19) and the class CR of type* ClusterOfRule *(see Example 26). The operation $\theta_p(CR, AR1)$ returns a set of patterns of type* ClusterOfRules *refining at least one association rule in $AR1$.* □

Table 5.3 presents several examples of $\mathcal{E}APQL$ queries.

5.2.3 $\mathcal{E}CPQL$: the Extended Pattern Query Calculus

In this section, a declarative query language for $\mathcal{EPM}$ patterns, Extended Calculus Pattern Query Language ($\mathcal{E}CPQL$) is introduced.

$\mathcal{E}CPQL$ is a many sorted calculus, based on the data and pattern types introduced in Chapter 4. In the following, we first present concepts concerning types and values, variables, terms, predicates, and formulas used to formalize the notion of $\mathcal{E}CPQL$ query.

Types and values. The sorts involved in the calculus for patterns are the ones corresponding to types in $\mathcal{OID} \cup \mathcal{T}_e \cup \mathcal{PT}$ (see Section 4.3), i.e., we consider one sort for each $t \in \mathcal{T}_e \cup \mathcal{PT}$ and a specific sort for OIDs. Values for those types belong to the domains introduced in Section 4.2.2. Values are denoted by $v, v_i, ...$ For each domain $dom(t)$, $t \in \mathcal{OID} \cup \mathcal{T}_e \cup \mathcal{PT}$, $v_i \in dom(t)$ is a constant of sort t. Constants of atomic types are called *atomic constants*.

Variables. We assume that for each sort $t \in \mathcal{OID} \cup \mathcal{T}_e \cup \mathcal{PT}$ there exists a countable set of variables V_t of sort t. We use the notation $x, x_i, y, y_i, ...$ to indicate variables in V_t.

Terms. Terms are defined as follows inductively as follows:

- Base cases:
  - all the atomic constants and variables are terms of the language;
  - $o \in \mathcal{OID}$, $\gg (o) \in dom(pt)$, where $pt \in \mathcal{PT}$, are terms of the language;
  - any path expression defined according to Def. 13 is a term of the language.
- Inductive step:
  - the results of the application of complex value constructors, i.e. {} and <>, to terms of the language are still terms of language.

Atomic formulas. Atomic formulas for $\mathcal{E}CPQL$ can be constructed by exploiting all predicates introduced in Section 5.2.1 applied over terms of the right type. Furthermore, as usual in the (object-) relational context, the name of any relation in the Universal Data Base and, similarly, the name of any $\mathcal{EPM}$ pattern class belonging to the PBMS can be used as predicates in constructing atomic formulas for $\mathcal{E}CPQL$. More precisely, assuming that p is a term of type $pt \in \mathcal{PT}$ and that $c \in \mathcal{CN}$ is a pattern class name defined over pt, then the expression $c(p)$ is a legal $\mathcal{E}CPQL$ atomic formula, expressing the fact that the pattern p belongs to class c. Similarly, if t is a data tuple term and $R \in \mathcal{RN}$ is a relation name, then the expression $R(t)$ is a legal $\mathcal{E}CPQL$ atomic formula, expressing the fact that the data tuple t belongs to relation R. Moreover, for all $pt \in calPT$, $p \in ext(pt)$ is an atomic formula, expressing the fact that the pattern p belongs to the extension of the pattern type pt. In a similar way, $p.d(t)$ and $p.f(t)$ are $\mathcal{E}CPQL$ atomic formula, expressing the fact that the data tuple t belongs to $p.d(UDB)$ and that t makes $p.f$ true, respectively.

Non atomic formulas. Non atomic formulas are constructed starting from atomic formulas by applying the usual logical connectives, $\wedge$, $\vee$, $\neg$, and $\rightarrow$. Furthermore, quantifiers $\forall$ and $\exists$ can be used to construct quantified formulas. Therefore, if $\psi(x)$ is a formula, also $\forall x\psi(x)$ and $\exists x\psi(x)$ are formulas. In the first case ($\forall x\psi(x)$), x is a universally quantified variable; in the second case ($\exists x\psi(x)$), x is an existential quantified variable. Moreover, the use of parenthesis is allowed in order to explicitly express the scope of a quantifier. Thus, $\forall x(\psi(x))$ and $\exists x(\psi(x))$ are formulas of the language. As usual, variables not involved in the scope of a quantifier are called *free variables*.

Informally, a pattern query can be viewed as an expression describing a collection of data or patterns or simply pattern identifiers satisfying certain properties. More formally, based on the previous concepts, an $\mathcal{E}CPQL$ query can be defined as follows.

Definition 36 ($\mathcal{E}CPQL$ *query*) A $\mathcal{E}CPQL$ query is an expression

$$\{x|\beta(x)\}$$

where β is a formula with exactly one free variable x of sort $t \in \mathcal{T}_e \cup \mathcal{PT}$. The result of a $\mathcal{E}CPQL$ query is a collection of data or patterns or simply pattern identifiers satisfying the formula β. □

Notice that, when using a cross-over predicate in β, the resulting query is a cross-over query, otherwise it is a simple query.

Concerning the query classification presented in Section 5.1, we point out that $\mathcal{E}CPQL$-queries returning a dataset as output, i.e., queries of the form $\{x|\beta(x)\}$ where x is of sort $t \in \mathcal{T}_e$, are D-queries, whereas $\mathcal{E}CPQL$-queries returning as output pattern set, i.e., queries of the form $\{x|\beta(x)\}$ where x is of sort $t \in \mathcal{PT}$, are P-queries.

Example 39 *Let AR1 and AR2 be two classes of type* AssociationRule *(see Example 19) and let CR be a class of type* ClusterOfRules*(see Example 26). Some examples of queries expressed by using the proposed calculus for patterns are shown in the third column of Table 5.3.* □

5.3 $\mathcal{E}PML$: the Extended Pattern Manipulation Language

The *Extended Pattern Manipulation Language* ($\mathcal{E}PML$) provides primitives for generating patterns from raw data, inserting patterns in the PBMS from scratch, deleting and updating patterns. In particular, according to requirements pointed out in Chapter 3, all these operations have to be defined by taking into account differences between a-posteriori and a-priori patterns, temporal validity issues, and pattern hierarchies. Furthermore, the $\mathcal{E}$PML has to support class management, i.e., it has to provide primitives to insert and delete patterns from classes.

In the following, $\mathcal{E}$PML primitives are presented grouped into four different classes: (a) insertion operations, (b) deletion operations, (c) update operations, and (d) operators for classes. We assume such operations are executed over a pattern base $PB = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$, where each $D_i \in \mathcal{T}$ and each $PT_i \in \mathcal{PT}$.

5.3.1 Insertion Operations

To cope with both a-posteriori and a-priori patterns, two different types of insertions are supported: *extraction* and *direct insertion*. In particular, extracting patterns means to apply a mining function to an input raw data (or pattern) set to generate new patterns, which have to be inserted in the PBMS. Thus, the extraction operation provides specific support for a-posteriori patterns. On the other side, by using the direct insertion operation, a-priori patterns can be supported. Indeed, when a user knows a pattern but she does not know the mining process used to generate it (or she does not care about it) and she wishes to insert this pattern in the system, then a direct insertion operation has to be used.

Due to temporal management, all insertion operators have to cope with validity period settings. In case of extracted patterns (i.e., a-posteriori ones), the validity period of the resulting patterns can be assigned by the mining function used to mine patterns (see Section 4.4) or directly specified by the user. In many cases mining functions do not directly deal with temporal information. Indeed, in general, mining algorithms do not assume raw data are temporized, as in the case of K-Means clustering methods for 2D point distribution [HK01]. Moreover, even when they consider temporized data, the output pattern

they produce may not have any temporal information associated with. The most common example of this case is the standard Apriori mining technique, proposed by Agrawal and Srikant [AS94], generating association rules from a set of timestamped transactions. In all such cases a default setting, specified by the user, for the pattern validity period can be considered. On the other side, concerning a-priori patterns, the validity period has to be set by the user, according to the chosen temporal theory, as well as any other pattern component.

Besides extraction and direct insertion, there is at least one other approach to generate patterns from raw data. It corresponds to the *recomputation* operation. Indeed, given some patterns, we may be interested in establishing whether they can represent a specified dataset, possibly different from the one they are associated with, computing the new measures. The resulting patterns have the same structure than the input ones but their data source and measures are, in general, different.

A mining process or a recomputation process may result in a big quantity of patterns. However, in some cases, the user may be interested only in some of them. For instance, when extracting association rules the user may want to made permanent in the system only association rules with specified minimum support and confidence. In order to support the specification of these filtering criteria, $\mathcal{E}$PML operations allow the user to specify a *condition*, based on which the generated pattern set is filtered and only those patterns satisfying the condition are effectively inserted in the PBMS, the others are discarded. Such a condition can be represented as an $\mathcal{E}APQL$ selection condition (see Section 5.2.2).

We remark the fact that the $\mathcal{EPM}$ pattern type of the patterns a user wants to insert in the PBMS has to already exist in the system before executing the insertion operation. In details, insertion operations can be described as follows.

Extraction ($\mathcal{E}(pt, D, cond, \mu, pr)$). It generates patterns of a specific pattern type $pt \in \{PT_1, ..., PT_h\}$, defined in PB, from a raw dataset $D \in dom_i(pt.t_d)$ by applying a specific mining function μ defined for pt in PB. A filtering condition *cond*, corresponding to an $\mathcal{E}$PQL selection predicate, can be specified and only patterns satisfying *cond* are effectively inserted in the system. A pattern validity value, $pr \in dom(\mathcal{V})$, can be explicitly specified by the user and, in this case, it is assigned to the extracted a-posteriori patterns.

The $\mathcal{E}(pt, D, cond, \mu, pr)$ operation generates a new pattern base $PB' = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(pt), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$ where:

$$ext(pt) = ext(pt) \cup \{p | p \in \mu(D) \wedge cond(p)\}$$

All the other structures of the pattern base remain the same.

Direct Insertion ($\mathcal{I}(pt, d, s, pr)$). It allows the user to insert in the PBMS patterns from scratch, therefore it provides support for a-priori patterns. It takes as input a pattern

type $pt \in \{PT_1, ..., PT_h\}$, a data source $d \in dom(pt.t_d)$, a structure $s \in dom(pt.t_s)$, and inserts in the PBMS the pattern generated from those data, by setting a user defined validity value $pr \in dom(\mathcal{V})$ for the new pattern. Measures are not specified since the user may not know their values at insertion time. Measures can then be computed by using synchronization (see below).

The $\mathcal{I}(pt, d, s, pr)$ operation generates a new pattern base $PB' = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(pt), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$ where:

$$ext(pt) = ext(pt) \cup \{< newpid(), s, d, \perp, pt.f[\tau/s], pr >\}$$

All the other structures of the pattern base remain the same.

Recomputation ($\mathcal{R}(pt, cond, d, \mu_m, pr)$). It generates new patterns from already existing ones, by recomputing their measures over a given raw dataset. More precisely, given a pattern type $pt \in \{PT_1, ..., PT_h\}$, for all instances of pt in PB, the measures of those patterns over a raw dataset d are computed, accordingly to some input measure function μ_m defined for pt. New patterns are created and inserted into the system. In the case a filtering condition *cond* is specified, only those patterns satisfying *cond* are inserted in the PBMS. A pattern validity value, $pr \in dom(\mathcal{V})$, can be explicitly specified by the user and, in this case, it is assigned to the recomputed patterns.

The $\mathcal{R}(pt, cond, d, \mu_m, pr)$ operation generates a new pattern base $PB' = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(pt), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$ where:

$$ext(pt) = ext(pt) \cup \{< newpid(), p.s, p.d, \mu_m(p, D), p.f, pr > | p \in dom(pt) \wedge cond(p)\}$$

All the other structures of the pattern base remain the same.

5.3.2 Deletion Operations

Once a user is no more interested in a certain pattern, she may want to be able to remove it from the system. However, according to the logical model presented in Chapter 4, in deleting patterns two problems have to be faced: (i) a pattern, at a certain instant of time, may belong to more than one pattern class; (ii) a pattern may share hierarchical relationships with other patterns.

Concerning the first problem, recall that only patterns belonging to classes can be queried by the user. Thus, it may be possible that the user does not want to remove a pattern from the system if it is still queriable (i.e., it is contained at least in one class). On the other hand, in some cases, the user could be interested in removing the pattern from the system anyway.

Concerning the second problem, we assume that if a pattern is involved in a composition or refinement relationship with other patterns, it cannot be removed in order to maintain referential integrity.[5]

In summary, two types of deletion operations to remove a pattern from the PBMS according to the user requirements are provided.

Deletion_restricted ($\delta_R(pt, cond)$). The instances of pattern type $pt \in \{PT_1, ..., PT_h\}$ satisfying condition *cond* are removed from the PBMS if they belong to no class and they are not referred by any other patterns neither in the structure (i.e., by composition) nor in the data source (i.e., by refinement). More formally, the last condition means that does not exist an $\mathcal{EPM}$ pattern $\overline{p}$ of type $\overline{pt}$ s.t. $\overline{p} \hookrightarrow^* p$ or $\overline{p} \Rightarrow^* p$.[6]

The $\delta_R(PT_i, cond)$ operation generates a new pattern base $PB' = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(pt), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$ where:

$$ext(pt) = ext(pt) - \{p | p \in dom(pt) \wedge cond(p) \wedge \neg \exists c\, c(p) \wedge \neg \exists \overline{p} (\overline{p} \hookrightarrow^* p \vee \overline{p} \Rightarrow^* p)\}$$

All the other structures of the pattern base remain the same.

Deletion_extended ($\delta_E(pt, cond)$). It deletes instances of pattern type $pt \in \{PT_1, ..., PT_h\}$ satisfying condition *cond* which are not referred in the structure or data source of any other pattern.

The $\delta_E(pt, cond)$ operation generates a new pattern base $PB' = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(pt), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$ where:

$$ext(pt) = ext(PT_i) - \{p | p \in dom(pt) \wedge cond(p) \wedge \neg \exists \overline{p} (\overline{p} \hookrightarrow^* p \vee \overline{p} \Rightarrow^* p)\}$$

All the other structures of the pattern base remain the same.

5.3.3 Update operations

Pattern update operations cope with the synchronization and validation of patterns with respect to their data source component. According to the pattern logical model presented in Chapter 4, we assume that only measures and the validity period can be changed, by using update operators. Due to the fact that a modification of the pattern structure or data source component carries with it an important change in the pattern semantics, it

[5] We remark that other approaches are however possible. For example, referenced patterns can be deleted in cascade. We do not further discuss this issue for the sake of simplicity.

[6] The symbol $*$ denotes the transitive closure of the hierarchical relationship.

would correspond to the creation of a new pattern. As in the case of insertion operations, each update operator is equipped with a *cond* parameter, representing a filtering condition. Note that the existence of a refinement relationship involving patterns under update has no impact over the proposed update operators. Within $\mathcal{E}$PML, the following update operators are available.

Synchronize ($\mathcal{S}(pt, cond, \mu_m)$). It allows the user to re-compute the measure values associated with the patterns, in order to reflect modifications that occurred in the data source. Note that, only measures are modified, the pid, structure, the data source, the pattern formula, and the validity period of the patterns do not change. Thus, synchronization can be seen as a special case of recomputation, applied against the pattern data source. In detail, it makes patterns safe *without changing* their validity period. More precisely, it allows the user to re-compute the measure values associated with temporally valid patterns, instances of a pattern type $pt \in \{PT_1, ..., PT_h\}$ and satisfying a given condition *cond*, in order to reflect modifications that occurred in its data source, by using a measure function μ_m, defined for pt in PB, specified as input. Only patterns that are temporally valid are synchronized since all the others, by definition of safety, cannot become safe.

The $\mathcal{S}(pt, cond, \mu_m)$ operation generates a new pattern base $PB' = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(pt), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$ defined as follows($current\_time \in dom(\mathcal{V})$ is the system time at which the synchronization operation is invoked):

$$\begin{aligned} ext(pt) = & [ext(pt) - \{p | p \in dom(pt) \wedge cond(p) \wedge \omega_T(p, current\_time)\}] \cup \\ & \{< p.pid, p.s, p.d, \mu_m(p, p.d), p.f, p.v > | p \in dom(pt) \wedge cond(p) \wedge \\ & \neg\omega_T(p, current\_time)\} \end{aligned}$$

All the other structures of the pattern base remain the same.

Validate ($\mathcal{V}(pt, cond, \mu_m)$). It makes $\mathcal{EPM}$ patterns safe *by changing* their validity period. More precisely, it first recomputes the measure values associated with temporally valid patterns, instances of a pattern type $pt \in \{PT_1, ..., PT_h\}$, satisfying a given condition *cond*, by using the specified measure function μ_m defined for pt in PB. If such measure values are better than the ones associated with patterns before validation, patterns are semantically valid. Thus, similarly to synchronization, if patterns are temporally valid, measures are modified, and the validity period is left unchanged. On the other hand, if measures are worst than before or the pattern is not temporally valid,, the validity period of the pattern is changed. We assume the temporal theory $\mathcal{V}$ chosen to model pattern validity within $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ provides a special constant value, called *current_time*, representing the actual instant of time and another constant value, called EOT, representing the end of time. Therefore, by using such special values we set the end of the pattern validity to *current_time*

and a new pattern is created, with the same structure and dataset than the previous one, but with the new measures. Its validity period is set to start at *current_time* and to end at EOT. Since, after validation, patterns are semantically valid within their validity period, it is possible to use the validity period as an approximation of the periods in which a pattern is semantically valid.[7]

The $\mathcal{V}(pt, cond, \mu_m)$ operation generates a new pattern base $PB' = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(pt), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$ defined as follows:

$$
\begin{aligned}
A =& \{p | p \in dom(pt) \wedge cond(p) \wedge \mu_m(p, p.d) \theta p.m \wedge \omega_T(p, current\_time)\} \\
B =& \{< p.pid, p.s, p.d, \mu_m(p, p.d), p.f, p.v > | \\
& \quad p \in dom(pt) \wedge cond(p) \wedge \mu_m(p.d) \theta p.m \wedge \omega_T(p, current\_time)\} \\
C =& \{p | p \in dom(pt) \wedge cond(p) \wedge \neg(\mu_m(p, p.d) \theta p.m \vee \omega_T(p, current\_time))\} \\
D =& \{< newpid(), p.s, p.d, p.m, p.f, v_{[current\_time, \infty]} > | \\
& \quad p \in dom(pt) \wedge cond(p) \wedge \neg(\mu_m(p, p.d) \theta p.m \vee \omega_T(p, current\_time))\} \\
ext(pt) =& (ext(pt) - A \cup B) - C \cup D
\end{aligned}
$$

In the previous expression, we denoted with $v_{[current\_time, \infty]}$ the formula of the temporal theory $\mathcal{V}$ representing the time interval $[current\_time, \infty]$.

All the other structures of the pattern base remain the same.

Since, according to $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, a validity information conforming the temporal theory $\mathcal{V}$ is associated with each pattern, it may be useful for the user to be able to (re-)set this information once the pattern is already inserted in the system. Therefore, besides the previous update operators, $\mathcal{E}$PML provides the following operator to change a pattern validity period by setting it to a new value belonging to $\mathcal{V}$. As in the previous cases, the *New_Period* operator is equipped with a *cond* parameter, representing a filtering condition.

New_Period ($\mathcal{N}_P(pt, cond, pr)$). It is used to update a set of patterns, instances of a pattern type $pt \in \{PT_1, ..., PT_h\}$ defined in PB according to $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$, and satisfying the user-defined filtering condition, *cond*, by setting a new user specified value, $pr \in dom(\mathcal{V})$, for their validity periods. Note that this operator does not recompute the measure values of the patterns. Such an operator is useful in order to be able to support pattern temporal characteristics management, since it allows the user to maintain up-to-date the validity information associated with patterns. Indeed, by using this manipulation operator the user may change the pattern validity, by assessing that a pattern is expired, yet, or that its expiration time is delayed.

[7] Notice that here we consider shallow temporal validity.

The $\mathcal{N}_P(PT_i, cond, pr)$ operation generates a new pattern base $PB' = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(PT_i), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$ where:

$$ext(PT_i) = [ext(PT_i) - \{p | p \in dom(PT_i) \wedge cond(p)\}] \cup \{< p.pid, p.s, p.d, p.m, p.f, pr > | p \in dom(PT_i) \wedge cond(p)\}$$

All the other structures of the pattern base remain the same.

5.3.4 Operators for Classes

According to $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ (see Chapter 4), a pattern has to be inserted in at least one class in order to be queried. Therefore, two $\mathcal{E}$PML operations supporting the insertion and the removal of a pattern into or from a class are provided. We remark that deletion from class operator removes patterns from a class, but leaves them in the system and in all classes it belong. Note that the existence of any hierarchical relationship involving patterns under manipulation has no impact over the proposed operators for pattern classes.

Insertion_Into_Class ($\mathcal{I}_\mathcal{C}(p.pid, c)$). It allows one to insert a pattern p into a specific class $c \in \{PC_1, ..., PC_h\}$ in PB. More formally, let $PT_i \in \{PT_1, ..., PT_h\}$ be the pattern type for which the class PC_i is defined (i.e., $\widehat{PC_i} :: PT_i$). The execution of the operation $\mathcal{I}_\mathcal{C}(p.pid, PC_i)$ generates a new pattern base $PB' = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(PT_h)], [PC_1, ..., PC_i, ..., PC_m] \rangle$ where:

$$PC_i = PC_i \cup \{p | p \in dom(PT_i) \wedge p.pid\}$$

All the other structures of the pattern base remain the same.

Deletion_From_Class ($\mathcal{D}_\mathcal{C}(cond, c)$). It allows the user to remove the patterns satisfying a given condition *cond* from a given class $c \in \{PC_1, ..., PC_h\}$ in PB. More formally, let $PT_i \in \{PT_1, ..., PT_h\}$ be the pattern type for which the class PC_i is defined (i.e., $\widehat{PC_i} :: PT_i$). The execution of the operation $\mathcal{D}_\mathcal{C}(cond, PC_i)$ generates a new pattern base $PB' = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(PT_h)], [PC_1, ..., PC_i, ..., PC_m] \rangle$ where:

$$PC_i = PC_i - \{p | p \in dom(PT_i) \wedge cond(p)\}$$

All the other structures of the pattern base remain the same.

5.3.5 A Final Example

In the following, we present a conclusive example concerning the usage of $\mathcal{E}PML$ operators in a possible market-basket scenario.

Example 40 *Let Q_{Rome} and Q_{Milan} be two intensional descriptions of two different datasets concerning sales in stores in Rome and Milan. Let $\mu_{aPriori}$ be the mining function corresponding to the* A-Priori *algorithm for computing association rules and let μ_{MAR} be a measure function for pattern type* AssociationRule*. Let AR_R, AR_M, and AR classes containing association rules. The following are some examples of interesting PML operations:*

1. *A set of association rules about sales in a department store in Rome are extracted and inserted into AR_R:*

   - $for\,all\,p.pid \in \mathcal{E}(AssociationRules, Q_{Rome}, true, \mu_{aPriori}, null)$
     $do\{\mathcal{I}_C(p.pid, AR_R)\}$

   Note that, no validity period is specified for the extracted patterns.

2. *New patterns SP' are generated by recomputing SP over the dataset concerning sales in a department store in Milan, intensionally described by Q_{Milan}. Then, they are inserted into both classes AR_M and AR.*

   - $for\,all\,p.pid \in \mathcal{R}(SP, Q_{Milan}, \mu_{MAR})$
     $do\{\ \mathcal{I}_C(p.pid, AR_M)$
     $\quad\mathcal{I}_C(p.pid, AR)\}$

3. *Association rules with confidence value lower than 0.5 not belonging to any class are removed from the system.*

   - $\delta_R(AR, p.m.confidence < 0.5)$

4. *The association rules representing transaction in Q_{Rome} can be synchronized to reflect changes in their data source by using the following operation:*

   - $\mathcal{S}(AssociationRules, Q_{Rome} \subseteq^i p.d, \mu_{MAR})$ □

Query description	**$\mathcal{E}$APQL query**	**$\mathcal{E}$CPQL query**
Find all association rules in class AR1 whose confidence is greater than 0.7:	$\sigma_{p.m.confidence>0.7}(AR1)$	$\{p \mid \exists p'\ AR1(p') \wedge p'.m.confidence > 0.7 \wedge p = p'\}$
Find the association rules in class AR2 mined from a dataset containing sales in Rome, intensionally described by the query *Q_Rome*	$\sigma_{Q\_Rome \subseteq^i p.d}(AR2)$	$\{p \mid \exists p'\ AR2(p') \wedge Q_{Rome} \subseteq^i p'.d \wedge p = p'\}$
Find all association rules in class AR1 whose body contains 'Bread' or whose confidence is greater than 0.75	$\sigma_{'Bread' \in\ p.s.body\ \vee\ p.m.confidence>0.75}(AR1)$	$\{p \mid \exists p'\ AR1(p') \wedge ('bread' \in p.s.body \vee p.m.confidence > 0.5) \wedge p = p'\}$
Find association rules in class AR2 whose body contains 'Hat' which are temporally valid on March,1 2006:	$\sigma_{'Hat' \in p.s.body \wedge \omega_T(p,'01-mar-06')}(AR2)$	$\{p \mid \exists p'\ AR2(p') \wedge' hat' \in p.s.body \wedge \omega_T(p,'01\text{-}mar\text{-}06') \wedge p = p'\}$
Retrieve all rules in $AR1$ with just support measure associated with, i.e., projecting out the confidence value	$\pi^m_{support}(AR1)$	$\{p =< newpid(), s, d, m, f, v > \mid \exists p_1 =< pid_1, s_1, d_1, m_1, f_1, v_1 > \wedge AR1(p_1) \wedge p.s = p_1.s_1 \wedge p.d = p_1.d_1 \wedge p.f = p_1.f_1 \wedge p.v = p_1.v_1 > \wedge p.m = \pi_{support}(p_1.m_1)\}$
Find rules containing 'Bread' in the head or having a support greater than 0.6, belonging to the dataset by which patterns in class CR have been mined	$\sigma_{'Bread' \in\ p.s.head\ \vee\ p.m.support>0.6}($ $\delta_{AssociationRule}(CR))$	$\{p =< newpid(), s, d, m, f, v > \mid \exists p_1\, CR(p_1) \wedge \exists p_2\, p_2 =< pid_2, s_2, d_2, m_2, f_2, v_2 > \wedge p_1.d.rule = p_2 \wedge' bread' \in p_2.s.head \wedge p_2.m.support > 0.6 \wedge p = p_2\}$
Retrieve new patterns containing only head as structure components and support as measure starting from association rules belonging to class $AR1$ having at least confidence 0.7 and whose body contains 'Bread'	$\lambda_{\pi_{p.s.head}}((\pi^m_{support}($ $\sigma_{'Bread' \in p.s.body \wedge p.m.confidence>0.75}(AR1))))$	$\{p =< pid, s, d, m, f, v > \mid \exists p_1 =< pid_1, s_1, d_1,\ m_1, f_1, v_1 > (\exists p_2 =< pid_2, s_2, d_2, m_2, f_2, v_2 > \wedge AR1(p_2) \wedge 'bread' \in p_2.s.body \wedge p_2.m.confidence > 0.75 \wedge p_1.pid_1 = newpid() \wedge p_1.s_1 = p_2.s_2 \wedge p_1.d_1 = p_2.d_2 \wedge p_1.m_1 = \pi_{support}(p_2.m_2) \wedge p_1.f_1 = p_2.f_2 \wedge p_1.v_1 = p_2.v_2) \wedge p.pid = p_1.pid_1 \wedge p.s = \pi_{head}(p_1.s) \wedge p.d = p_1.d_1 \wedge p.m = p_1.m_1 \wedge p.f = p_1.f_1 \wedge p.v = p_1.v_1\}$
Retrieve all representative of clusters of rules refining at least an association rule belonging to class $AR1$ mined from a dataset containing all sales in Rome, denoted by *Q_Rome*	$\mathcal{C}_{AssociationRule}($ $\rho_{ClusterOfRules}(\sigma_{Q\_Rome \subseteq^i p.d}(AR1)))$	$Q3 \equiv \{p / \exists p_1\ AR1(p_1) \wedge Q_{Rome} \subseteq^i p_1.d \wedge p = p_1\}$ $Q2 \equiv \{p / \exists p_2\ \exists p_1 \in Q3 \wedge p_2 = p_1 \wedge p = p_2\}$ $\{p / \exists p_4\ \exists p_3 \in Q2 \wedge p_3 \hookrightarrow p_4 \wedge p = p_4\}$

Table 5.3: $\mathcal{E}$APQL and $\mathcal{E}$CPQL query examples. In the queries, $AR1$ and $AR2$ are two classes of type `AssociationRule` and CR is a class of type `ClusterOfRules`.

Chapter 6

Formal Results about the Proposed Framework

In this chapter, we present some results concerning the expressive power and the complexity of proposed framework for pattern management. In particular, Section 6.1 focuses on the equivalence between the two proposed pattern query languages: the algebra ($\mathcal{E}$APQL) and the calculus ($\mathcal{E}$CPQL). To this purpose, first some safety restrictions for the proposed calculus will be introduced and, then, the equivalence between such restricted calculus and the algebra is proved. Section 6.2 presents a complexity analysis of the proposed framework for pattern management.

6.1 Equivalence Results about Languages for Patterns

The aim of this section is to state an equivalence result between the pattern algebra and calculus presented in Chapter 5. This constitutes an important issue for the development of any data or pattern management framework. Indeed, in general, query languages proposed for human users are based on the declarative approach (i.e., the calculus), whereas their internal system implementations are based on the algebraic approach for efficiency and optimization issues.

The equivalence result we are going to present is similar to the one proposed for complex values query languages by Abiteboul and Beeri [AB95]. They start by observing that, as in the case of traditional relational algebra and calculus, complex value algebra and calculus are not equivalent. Indeed, there exist some complex values calculus query which cannot be expressed with the algebra for complex values. For instance, by using complex calculus, it is possible to express queries having as result an infinite relation.

The key to prove that the two approaches are equivalent is based on the concept of *domain independence* for query languages. This notion is primary aimed at making query results independent from the domain values, but simply dependent from the database content. Unfortunately, the domain independence is a semantical property which cannot be checked in a static way. Therefore, in [AB95] sufficient conditions for domain independence, known as *safety conditions*, have been proposed, requiring calculus formulas to be range restricted with respect to finite domains of values. By exploiting these safety conditions, a safe variant of the complex value calculus can be provided, which is then equivalent to the complex values algebra [AB95].

A similar argumentation applies to our case. Indeed, $\mathcal{E}APQL$ and $\mathcal{E}CPQL$ defined in Chapter 5 are not equivalent. As an example, let $pt \in \mathcal{PT}$ be a pattern type and let pb be a pattern base (Def. 27) containing among the others a pattern class c defined for pt (i.e., $\widehat{c} :: pt$), consider the $\mathcal{E}CPQL$ query: $\{p | \exists p' \neg c(p') \wedge p = p'\}$. It generates an infinite result set containing all patterns $p \in dom(pt)$ not belonging to c. Such kinds of queries cannot be expressed in $\mathcal{E}APQL$.

In the following, we rely on the approach presented in [AB95] to discuss equivalence between $\mathcal{E}APQL$ and $\mathcal{E}CPQL$. As a preliminary consideration, we notice that, according to the $\mathcal{EPM}$ model, patterns are complex objects and not complex values as in [AB95], due to the fact that they have associated a unique pattern identifier, which can be used to define hierarchical patterns. Therefore, in order to rely on the proof presented in [AB95], we first apply a preprocessing step replacing each pattern identifier with the corresponding value. It is easy to prove that this processing phase can be executed in a polynomial time with respect to the dimension of the pattern base.

After this initial step, we may apply the technique proposed for complex values to our case. To this purpose, according to what done in [AB95], we introduce a safe calculus, defined by imposing several safety restrictions over $\mathcal{E}CPQL$. Such a *safe calculus*, named $\mathcal{E}CPQL^{SS}$, is indeed a *strictly safe calculus*, which will be demonstrated equivalent to $\mathcal{E}APQL$.

We recall that, intuitively, a formula is safe range when each variable appearing in it is bounded, in the sense that it ranges within the active domain of the query itself or in the active domain of the input database or pattern base. Moreover, a formula is strictly safe range when it is safe range and it does not contain any (extensional) inclusion predicate. The formalization of the previous notions are briefly recalled in the folowing preliminaries section.

The remainder of this section is organized as follows. In Section 6.1.1.1, we introduce some preliminary formal notions. In Section 6.1.2 a strictly safe range version of the pattern calculus ($\mathcal{E}CPQL^{SS}$) is presented. Finally, in Section 6.1.3 the equivalence between $\mathcal{E}APQL$ and $\mathcal{E}CPQL^{SS}$ is demonstrated.

6.1.1 Preliminaries

In this section, several preliminary notions are presented. In particular, in Section 6.1.1.1, the concept of *domain independence* for pattern bases queries is introduced by extending the well known notion of domain independent query proposed in [AB95]. Then, Section 6.1.1.2 introduces notions of *active domains* for pattern bases and pattern queries.

6.1.1.1 Domain Independence

The notion of domain independent database query is primary aimed at making query results independent from the domain values, but dependent from the database content (i.e., data in relations).

Similarly to what has been done in the database context, we say that a query q over a pattern base is domain independent if, for any pattern base structure, changing the domains (but keeping them large enough to contain all atomic entries appearing in the pattern base or in the query) does not change the query result. Since a pattern base PB is defined over a database DB and over a given set of pattern type extensions (see Def. 26 and 27), this means that q has to be domain independent with respect to DB's domains. This concept is formalized by the following definition.

Definition 37 (*Domain independent pattern base query*) Let $\widehat{PB} = \langle[\widehat{D_1}, \ldots, \widehat{D_k}], [PT_1, \ldots, PT_h], [\widehat{PC_1} : PT_1, \ldots, \widehat{PC_m} : PT_m], [\mu_1, \ldots, \mu_r], [\mu_{m1}, \ldots, \mu_{ms}]\rangle$ be a pattern base schema where $\widehat{D_1}, \ldots, \widehat{D_k}$ are data types of the data base schema $\widehat{DB}$ over which the PBMS is defined. Let $PB = \langle[D_1, \ldots, D_k], [ext(PT_1), \ldots, ext(PT_h)], [PC_1, \ldots, PC_m]\rangle$ be one of its pattern base instances. An $\mathcal{E}PQL$ query q with signature $q : 2^{\widehat{DB}} \times 2^{\widehat{PB}} \longrightarrow 2^{\widehat{DB}} \times 2^{\widehat{PB}}$ is *domain independent* if, for any other pattern base instance $PB' = \langle[D'_1, \ldots, D'_k], [ext(PT_1)', \ldots, ext(PT_h)'], [PC_1, \ldots, PC_m]\rangle$, $q(PB) = q(PB')$. □

We notice that in the previous definition PB and PB' differ only in the domains over which pattern types are defined and in the pattern type extensions, but the content of classes does not change. Based on the previous definition, the notion of domain-independent query language for patterns can be formalized as follows.

Definition 38 A pattern query language $\mathcal{L}$ is *domain independent* if any query q that can be expressed in $\mathcal{L}$ is *domain independent*. □

Since it is undecidable whether a given calculus query is domain independent (we refer interested readers to [AB95] for a technical discussion about this topic), syntactic conditions

that ensure domain independence have to be used. Usually, such conditions are known as *safety conditions* and they rely on the notion of *safe range query* or *safe range formula*. We will propose safety conditions for $\mathcal{E}CPQL$ formulas in the following Section 6.1.2.

6.1.1.2 Active Domains

In this section, notions of active domains for databases and database queries proposed in [AB95] are extended to pattern bases and pattern base queries. Such notions will be then exploited in provind the equivalence result. Within the following definitions, we assume $DB =< [D_1, ..., D_k], [R_1, ..., R_n] >$ is a data base instance of the database schema $\widehat{DB}$ according to Def. 7 and $PB = \langle [D_1, \ldots, D_k], [ext(PT_1), ..., ext(PT_h)], [PC_1, \ldots, PC_m] \rangle$ is a pattern base instance of the pattern base schema $\widehat{PB}$ according to Def. 27.

Definition 39 (*Pattern base active domain*) The *active domain of a pattern base* instance PB, called $adom(PB)$, is the set of all constants occurring in PB. The *active domain of a pattern class* instance C of a pattern base PB, called $adom(C)$, is the set of all constants occurring in C. □

Note that, since PB is formed by the pattern classes $C_1, ..., C_m$: $adom(PB) = adom(C_1) \cup ... \cup adom(C_m)$.

Definition 40 The *active domain of an $\mathcal{E}APQL$ query* q, called $adom(q)$, is the set of all constants occurring in q. The *active domain of a $\mathcal{E}CPQL$ formula* φ, called $adom(\varphi)$, is the set of all constants occurring in φ. □

We use the notation $adom(q, PB)$ as an abbreviation for $adom(q) \cup adom(PB)$ and $adom(\varphi, PB)$ as an abbreviation for $adom(\varphi) \cup adom(PB)$. Furthermore, we use the notation X_{adom} to indicate the active domain of an element X.

6.1.2 $\mathcal{E}CPQL^{SS}$: the Strictly Safe Calculus for Patterns

In order to make a formula *safe*, the usual approach is to require that each variable is related to a range formula. Concerning databases, the simplest forms of range formulas are $R(x)$, where R is a name of a database relation. Similarly, concerning pattern bases, the simplest forms of range formulas are $C(x)$, where C is a name of a pattern class in the pattern base. However, since we allow nested structures, other range restriction options have to be taken into account.

Furthermore, since we want to make our formulas *strictly safe* - i.e., safe and do not containing any extensional inclusion predicate - we do not allow (extensional) containment predicates. In particular among predicates introduced in Section 5.2.1, predicates checking extensional containment between pattern data sources and formula (i.e., $\subseteq^e$ and $\preceq^e$) are not allowed. Moreover, concerning predicates over complex value components of a pattern, the set containment predicate (i.e., $\subseteq$) is not allowed.

In the following, according to the approach proposed in [AB95], we give some conditions to establish whether a variable in a pattern calculus formula ϕ is range restricted. As in [AB95], we consider only formulas in which universal quantifier ($\forall$) does not occur and the connective $\rightarrow$ is not used. Note that, we do not loose in generality since formulas of the form $\forall x f(x)$ can always be translated into $\neg\exists x \neg f(x)$ or, similarly, $A \rightarrow B$ correspond to $\neg A \vee B$.

Let ϕ be a $\mathcal{E}CPQL$ formula and assume a partial ordering on its variables is given. We assume that the free and bound variables are distinct. We introduce the notion of *range restricted variable* for both free and bound data and pattern variables as follows.

Definition 41 (*Range restricted variable*) A variable x of type $s \in \mathcal{T}_e \cup \mathcal{PT}$ is range restricted in ϕ, relative to the given ordering, if the type of x is the empty tuple type, or one of the following holds:

- ϕ is one of the following atomic formulas:

  (B1) $R(x)$, where x has type $s \in \mathcal{T}_e$;

  (B2) $C(x)$, where x has type $s \in \mathcal{PT}$;

  (B3) $x\delta t$, where x has type $s \in \mathcal{T}_e$, t is a $\mathcal{T}_e$ term, and δ is an allowed predicate;

  (B4) $x\theta t$, where x has type $s \in \mathcal{PT}$, t is a $\mathcal{PT}$ term, and θ is an allowed predicate;

  (B5) $x = \{y|\psi(y)\}$, where all variables that are used in ψ (including y) precede x in the ordering.

- ϕ is one of the following non atomic formulas constructed over subformulas:

  (CL1) ϕ is a conjunct of two formulas, and x is restricted in at least one of them;

  (CL2) ϕ is a disjunct of two formulas, and one of the following holds:
  - x is free in it, in which case it must appear and be restricted in both disjuncts,
  - x is bound in it, in which case it appears in precisely one of the disjuncts, and is restricted in the disjunct where it appears;

  (CL3) ϕ is obtained from a formula in which x is restricted by adding an existential quantifier (on x, or on another variable);

(CL4) $x = \neg\psi$, where ψ is a formula in which x is both bound and restricted. □

Clearly, the empty tuple type needs no range restriction: its only value is $\perp$, anyhow.

The five base cases, (B1)-(B5), deal with atomic formulas where x is directly restricted in ϕ, either because it is required to be in some database relation or pattern class (in (B1) or (B2)), or because it is involved in an atomic formula of the calculus involving a predicate and a calculus term constructed over constants and variables that precede x in the partial order (in (B3) and (B4)), or (in (B5)) because it is required to be the result of a calculus query expressed by a formula ψ, where all the variables in ψ precede x. We outline the importance of the assumption that the range restriction for a variable depends only on variables that precede it in the ordering. Indeed, concerning cases (B3) and (B4) if the term t in (B3) and (B4) contains variables, then the range restriction of x depends on these variables, in the sense that it restricts x only if these variables are properly restricted. Then these variables have their own range restrictions, and further, their restricting formulas do not depend on x directly, or even indirectly through other variables. A similar remark applies to the case (B5) concerning variables in ψ. The closure cases, (CL1)-(CL4), deal with non atomic formulas.

According to the previous definition, the notions of *strictly safe formulas* and *strictly safe queries* for $\mathcal{E}CPQL$ can be formalized as follows.

Definition 42 (*Strictly Safe $\mathcal{E}CPQL$ formula*) An $\mathcal{E}CPQL$ formula is *strictly safe with respect to a given partial ordering*, if all the variables appearing in it are range restricted. Furthermore, it is *strictly safe* if there is an ordering, such that it is strictly safe with respect to it. □

Definition 43 (*Strictly Safe $\mathcal{E}CPQL$ query*) An $\mathcal{E}CPQL$ query $\{t|\varphi(t)\}$ is *strictly safe* if the range restricted variables of φ is equal to the free variables in t. □

Finally, the strictly safe pattern calculus, denoted by $\mathcal{E}CPQL^{SS}$, is the calculus where all queries are strictly safe.

Definition 44 (*Strictly Safe $\mathcal{E}CPQL$*) The family of strictly safe $\mathcal{E}CPQL$ queries defines the Strictly Safe $\mathcal{E}CPQL$, denoted by $\mathcal{E}CPQL^{SS}$. □

6.1.3 Equivalence Results Between $\mathcal{E}APQL$ and $\mathcal{E}CPQL$

Once introduced the strictly safe pattern calculus $\mathcal{E}CPQL^{SS}$ we can state an equivalence result between $\mathcal{E}APQL$ and $\mathcal{E}CPQL^{SS}$. The result we are going to present is an extension

of the well-known equivalence result between the complex value algebra and the calculus, demonstated in [AB95]. The proof of the next Theorem is therefore quite similar to the one presented in [AB95, AHV95] and it is presented in Appendix B

Theorem 1 *The $\mathcal{E}CPQL^{SS}$ and $\mathcal{E}APQL$ are equivalent.*

Proof Sketch. The proof can be divided in two parts, showing that:

1. $\mathcal{E}APQL \subseteq \mathcal{E}CPQL^{SS}$: For each $\mathcal{E}APQL$ query q_A there exists an equivalent $\mathcal{E}CPQL^{SS}$ query q_C. q_C can be constructed by induction over the type of q_A.
2. $\mathcal{E}CPQL^{SS} \subseteq \mathcal{E}APQL$: For each $\mathcal{E}CPQL^{SS}$ query $q_C = \{x|\phi\}$, there exists an equivalent $\mathcal{E}APQL$ query q_A. q_A can be constructed by induction over the subformulas appearing in the formula ϕ and by exploiting the strictly safe range conditions given above. □

6.2 Pattern Framework Complexity

This section is aimed at analyzing the complexity of the pattern framework we have developed. Especially, we want to state some preliminary complexity results concerning the pattern query language we proposed in Chapter 5. Indeed, as we have already stressed in Chapter 4, the logical model for patterns we have proposed is parametric with respect to the query language used to describe the pattern data sources, with respect to the logical theory used to express the pattern formulas, and with respect to the temporal theory used for pattern validity information. Different choices concerning these parameters lead to different expressive power of $\mathcal{EPM}$ and different complexities in pattern querying. In this section, we focus on these complexity aspects, by investigating possible choices and their related impact over the complexity of the proposed query languages. Since they are equivalent, in the following we focus on the proposed algebra.

The remainder of this section is organized as follows. Preliminary concepts and notions used throughout the section are presented in Section 6.2.1. On the other side, Sections 6.2.2, 6.2.3, and 6.2.4 focus, respectively, on validity period, formula, and data source components of patterns. Finally, an overall complexity result for the proposed query language is discussed in Section 6.2.5.

6.2.1 Preliminaries

In Chapter 5, we have introduced several predicates dealing with special components of patterns, such as data source, formula, and pattern validity period, (see Table 5.1). Due

to their nature, evaluating such predicates is not a constant cost operation. Thus, in order to be able to evaluate the complexity of $\mathcal{E}$APQL, we need to identify the complexity of such predicates. To this end, in the following, for each predicate, we identify some related logical problems for which complexity results are known (see Table 6.1). In particular, we notice that:

- for pattern data source, the related logical problems concern: query evaluation and query containment along with set inclusion and containment tests;
- for pattern formula, the related logical problems are the decision problem for the logical theory $\mathcal{F}$, chosen to represent formula constraints, and evaluation of the query that can be associated with a given formula;
- for pattern validity period, the related logical problem is the evaluation problem for formulas of the chosen temporal theory $\mathcal{V}$.

All the cited reference problems are defined in Chapter 5 and in Appendix A.

With respect to the previous problems, different complexities can be considered. In the following, we refer to canonical complexity classes. We refer the reader to [Pap94] for more details about their formal definition. Concerning query evaluation, depending on the parameter with respect to which the complexity is evaluated, several different notions of complexity can be defined:

Data Complexity: it measures the computational complexity of a fixed query over variable size database inputs.

Expression Complexity: it measures the computational complexity when the size of the query may change and the database is fixed.

Combined Complexity: it measures the computational complexity when both the size of the query and the database may change.

Since, it is well known that, independently from the used data model, the database size is much more larger (usually, of some order of magnitude) than the size of the query, the data complexity is the more used complexity when dealing with database query languages.

In the following, for each pattern components referenced in Table 6.1, we present additional information concerning its definition and the complexity of the related problems.

Pattern component	Operation	Logical operation	Logical problem
Data Source	$p_1.d \subseteq^e p_2.d$	$\{x \vert p_1.d(x)\} \subseteq \{x \vert p_2.d(x)\}$	Query evaluation + Set containment
Data Source	$p_1.d \subseteq^i p_2.d$	$p_1.d \subseteq p_2.d$	Query containment
Data Source	$p_1.d =^e p_2.d$	$\{x \vert p_1.d(x)\} = \{x \vert p_2.d(x)\}$	Query evaluation + Set equality
Data Source	$p_1.d =^i p_2.d$	$p_1.d \subseteq p_2.d \wedge p_2.d \subseteq p_1.d$	Query equivalence
Formula	$p_1.f(\overline{d})$	$p_1.f([\overline{d}/\overline{x}])$, where $\overline{d}$ an assignement of values for free variables $\overline{x}$ in $p_1.f$	Formula evaluation
Formula	$p_1.f \preceq^e p_2.f$	$\{\overline{x} \vert x \in p_1.d(UDB) \wedge p_1.f(\overline{x})\} \subseteq \{\overline{x} \vert x \in p_2.d(UDB) \wedge p_2.f(\overline{x})\}\}$	Query evaluation + Set containment
Formula	$p_1.f \preceq p_2.f$	$\forall \overline{x}\, p_1.f(\overline{x}) \rightarrow p_2.f(\overline{x})$	Decision problem
Formula	$p_1.f \equiv^e p_2.f$	$\{x \vert x \in p_1.d(UDB) \wedge p_1.f(x)\} = \{y \vert y \in p_2.d(UDB) \wedge p_2.f(y)\}$	Query evaluation + Set equality
Formula	$p_1.f \equiv^i p_2.f$	$\forall \overline{x}\, p_1.f(\overline{x}) \leftrightarrow p_2.f(\overline{x})$	Decision problem
Formula	$\Downarrow (f)$	$SAT(f)$	Satisfiability
Validity period selection	$p_1.v\theta_T p_2.v$ where θ_T is a predicate for $\mathcal{V}$	$p_1.v\theta_T p_2.v$ is true in $\mathcal{V}$	Formula evaluation

Table 6.1: Logical problems corresponding to predicates over pattern components

6.2.2 Validity period

Temporal theories represent time as an arbitrary set of instants with an imposed order. Therefore, an instant of time is the minimum non-decomposable unit along the fixed time axis. Axioms may also be added to the temporal theory to characterize the density of the time line. A discrete model of time is isomorphic to natural numbers. Dense models are isomorphic to either rationals or reals. Continuous model of time are isomorphic to reals. Moreover, independently from the chosen temporal theory, it is always possible to identify a notion of granularity. Intuitively, the granularity of a temporal theory defines a countable set of granules such that each granule corresponds to an instant of time.

According to what introduced in Chapter 4, pattern valid time information can be modeled by several distinct temporal theories, depending on the application context. As already remarked, in this thesis we focus on single-granularity time modeling. In the following, we report some examples of temporal theories. All of them are defined over Integers ($\mathbb{Z}$) with the total order relation $<$. Notice that such theories relies on the constraint classes defined in Appendix A. Results concerning satisfiability and formula evaluation in the related constraint theories are reported in Table 6.2.

Time Interval Theory - T^I: it supports the definition of valid time period information. According to this theory, the symbolic representation of a valid period has the form $[t_s, t_e]$, where t_s and t_e are two instants of time at a fixed granularity (i.e., years,

months, dates, timestamps, etc.).[1]

Under this theory, each symbolic valid time interval $[t_s, t_e]$ corresponds to an infinite set of time instants. Therefore, it can be translated using different constraint theories over $\mathbb{Z}$. More precisely, we may use: (a) conjunctions of lower and upper bound constraints over $\mathbb{Z}$, or (b) gap order constraints over $\mathbb{Z}$ [Rev90, Rev93]. Using the first option, the interval $[t_s, t_e]$, where $t_s, t_e \in \mathbb{Z}$ with $t_s \leq t_e$, is represented as: $t_s \leq t \wedge t \leq t_e$. On the other side, using gap order constraints the interval $[t_s, t_e]$ is represented as: $t' - (t_e - t_s) \geq t$.

Union Interval Theory - T^U: it supports the definition of valid time periods which are the union of disjoint intervals of time. According to this theory, a valid period associated with a certain data has the symbolic form $\cup_{i=1}^{k}[t_{s_i}, t_{e_i}], k \geq 0$, where t_{s_i} and t_{e_i} are, respectively, the start and end points of the i-th valid time interval, expressed with the same granularity.

As in the former case, also this theory exploits combination of lower and upper bound constraints or gap order ones over $\mathbb{Z}$. Indeed, a finite union of valid time intervals can be translated into a disjunction of constraints, each one representing a single time interval.

Periodic Time Expression Theory - T^P: it supports the definition of periodic valid time information. A typical example of such an information occurs when a certain fact is valid in all working days from 8 am to 1 pm, within a specific interval of time $[t_s, t_e]$.

Formally, periodic valid time can be modeled through a *temporal constraint*. A temporal constraint Ξ has the form $\{(PC_1, GC_1), ..., (PC_m, GC_m)\}$ where each PC_i is a conjunction of simple periodicity constraints over $\mathbb{Z}$ [TC98, BBFS98, Rev02] and each GC_i is a conjunction of gap-order constraints over $\mathbb{Z}$. A simple periodicity constraint of the form $t \equiv_k c$ where $k \in \mathbb{N}$ and $c \in \{0, ..., k-1\}$ (see Table A.1) denotes the set of integers of the form $c + nk$ with n ranging in $\mathbb{Z}$. For example, the valid period *every Sundays in 2005* could be represented, according to this temporal theory, by the following expression: $t \equiv_7 2 \wedge t_{31/12/04} < t \wedge t < t_{01/01/06}$, where $t_{31/12/04}$ and $t_{01/01/06}$ are the instants of time corresponding to December,31 2004 and to January,1 2006, respectively, and knowing that January,2 2005 was Sunday.

We notice that, by taking into account their expressive power, we may device a partial order between the temporal theories just introduced. Indeed, it is easy to show that all formulas in T^I can be represented inside T^U. Moreover, all the formulas in T^U can be represented in T^P. Thus, based on the notation introduced in Chapter 4, $\mathsf{T}^I \propto \mathsf{T}^U \propto \mathsf{T}^P$.

[1] Using this theory, it is also possible to specify left-open or right-open validity periods and time intervals containing exactly a single instant of time.

Theory	Complexity
Intervals over $\mathbb{Q}$ *or* $\mathbb{Z}$ (conjunction of upper bound and lower bound constraints)	LOGSPACE [KKR90]
Gap order over $\mathbb{Q}$ *or* $\mathbb{Z}$	PTIME [TC98, Kou94, Kou93, Kou95]
Periodicity constraints over $\mathbb{Z}$	PTIME [TC98]
Gap order over $\mathbb{Z}$ + *Periodicity constraints over* $\mathbb{Z}$	PTIME [TC98]

Table 6.2: Satisfiability complexity class for different types of constraint sets

6.2.3 Formula

According to what stated in Section 6.2.1, evaluating the complexity of query predicates based on pattern formulas can be performed by taking into account four distinct problems concerning the reference theory: (i) decision problem; (ii) satisfiability problem; (iii) formula evaluation problem; (iv) query evaluation, for queries based on the given formulas. Since the satisfiability problem and the formula evaluation problem can always be reduced to a decision problem, in the following we focus on the complexity of the decision problem and the data complexity of a query language based on a given constraint theory. The considered constraint theories are the following (see Appendix A for additional information about these constraint classes): (i) intervals over $\mathbb{Q}$ or $\mathbb{Z}$; (ii) gap order over $\mathbb{Q}$ or $\mathbb{Z}$; (iii) equality/ disequality over $\mathbb{Z}$; (iv) linear constraints over $\mathbb{R}$; (iv) polynomial constraints over $\mathbb{R}$.

Complexity results for FOL are summarized in Table 6.3. In such a table, each row corresponds to a constraint theory; the second and third columns report complexity results for the corresponding decision problems and data complexity.

From the results summarized in Table 6.3 emerges that using real linear (or polynomial) constraints for pattern formula leads, in general, to non tractable computational complexity for the decision problem. Therefore, in order be able to deal with real linear constraint in polynomial time several restriction to the logical theory have been considered [Kou01, Rev95a, LM92]. In particular, several constraint classes for which checking the consistency/satisfiability is in PTIME have been identified. In the following, we briefly recall some of them:

- *Generalized linear constraints over* $\mathbb{R}$ [LM92]: this class includes linear inequalities (e.g., $2X + Y \leq 6$) and disjunction of linear disequalities over $\mathbb{R}$ (e.g., $X + Y \neq 2 \vee X + 3Y + 2Z \neq 7$).
- *Horn constraints* [Kou01]: are formulas of the form $d_1 \vee ... \vee d_n$ where each d_i, $i = 1, ..., n$ is a week linear inequality or a linear disequality and the number of inequalities

Constraint Logical Theory	**Decision Problem F.O.L.**	**Data Complexity F.O.L.**
Intervals over $\mathbb{Q}$ *or* $\mathbb{Z}$	LOGSPACE [KKR90]	LOGSPACE-complete [Rev02]
Gap order over $\mathbb{Q}$ *or* $\mathbb{Z}$	PSPACE-complete [Kou94]	LOGSPACE-complete [Rev02]
Equality/ disequality over $\mathbb{Z}$	LOGSPACE [KKR90]	LOGSPACE [Kup90]
Linear constraints over $\mathbb{R}$	DSPACE(2^{cn}) [Rev95a, Kou01]	NC [Rev02, Kup90]
Polynomial constraints over $\mathbb{R}$	NEXP [KKR90, Tar51]	NC [Rev02]

Table 6.3: Decision problem complexity class for first order and object logical theories combined with constraint classes

among $d_1, ..., d_n$ does not exceed one (e.g., $X+Y \neq 2 \vee X+3Y+2Z \neq 7 \vee 2X+Y \leq 6$ is a Horn constraint, while $X + Z \neq 2 \vee 2X + Y \leq 6 \vee 2Z + 3W \leq 12$ is not).

When considering data complexity, many combinations of relational database query languages and constraint logical theories have PTIME (or lower) data complexity [Rev02, ACGK94, Kup90]. For example, the relational calculus extended with linear or polynomial contraints over $\mathbb{R}$ is in NC, whereas the same language extended with intervals, gap-order, or equality/disequality constraints over $\mathbb{Z}$ is in LOGSPACE.

When we consider first-order logic for complex-objects (denoted by COL in the following) [AB95, AHV95, LS97], we must consider that in general query languages for complex objects extended with logical theories (even without constraints) have in general at least EXPTIME decision problem and data complexity [BR95, Rev95b, HS91].

Starting from this result, and based on the observation that the high complexity is mainly due to the fact that variables may range over sets, several restricted languages for complex objects with complexity in PTIME have been proposed [AHV95, GV91, BK95]. In most cases, the basic idea is to impose some conditions over the maximum allowed level of nesting inside the database or inside queries [GV91]. Under such hypothesis, the query language can be proved to be tractable, i.e., in PTIME data complexity or even less. When considering constraints, the complexity class becomes the largest between the data complexity of FOL with the considered class of constraints and that for COL without constraints.

6.2.4 Data source

In our framework, pattern data sources are intensionally described, i.e., they are queries, expressed using a certain pattern query language $\mathcal{Q}$ with respect to which the $\mathcal{EPM}$ model is parametric (Section 4.2). Predicates over data source components may require checking for query containment and equivalence, under an intensional or extensional approach. In the first case, predicates over data source components reduce to query containment and equivalence problems. In the second case, they reduce to query evaluation problems.

Concerning the query evaluation problem, since we assume that $\mathcal{Q}$ is either a complex-value or complex-object query language without constraints, based on the considerations presented in Section 6.2.3 we know that it is possible to find languages with PTIME data complexity.

Concerning query containment and equivalence (defined in Def. 34 and Def. 35, respectively), it has been recognized that such problems are strictly related to the constraint satisfaction problem.

It is well-known that the query containment problem, is undecidable for arbitrary First-Order queries. Nevertheless, when restricting our attention to conjunctive queries (i.e., if we refer to an algebraic querying approach, queries using only select, project, join and Cartesian product operators), the containment problem becomes decidable even if it is hard. Chandra and Merlin have demonstrated that the conjunctive query containment problem is NP-complete [CM77]. Levy and Suciu [LS97] extended this complexity result also to the complex object context. Further, Klug [Klu88] has shown that containment for conjunctive queries with comparison predicates is in Π_2^p, and it is proven to be Π_2^p-hard by Van der Meyden in [vdM92].

Starting from those negative results, several special cases of conjunctive queries for which the containment problem becomes tractable have been identified. This is the case of Boolean conjunctive queries and conjunctive queries with bounded numbers of distinct variables [KV98] as well as acyclic queries or, more in general, conjunctive queries with bounded querywidth [KV98, CR00, Yan81, Qia96]. Complexity results concerning query containment are summarized in Table 6.4.

6.2.5 $\mathcal{E}$PQL Complexity

Starting from the results discussed in the previous sections, we are now able to present some preliminary results concerning the data complexity of the pattern languages proposed in Chapter 5.

As a starting point, we observe that the data complexity of evaluating a query over patterns

Type of queries	Complexity class
Relational or complex-object conjunctive queries	NP-complete [CM77, LS97]
Conjunctive queries with comparison predicates	Π_2^p-hard [Klu88, vdM92]
Boolean (binarized) conjunctive queries	PTIME [KV98]
Conjunctive queries corresponding to Datalog programs with bounded number of distinct variables	PTIME [KV98, LC00]
Conjunctive queries with bounded *querywidth* (Aciclic queries special case of conjunctive queries with querywidth = 1)	PTIME [KV98, CR00, Yan81, Qia96]

Table 6.4: Query containment complexity

is the result of the combination of the complexity of the used query language with the complexity of the predicate evaluation (which may itself requires a query evaluation/execution over patterns or data, as discussed in the previous sections and pointed out in Table 6.1). We can therefore prove the following therem (for the sake of simplicity in the following we focus on the algebraic language, however, due to the equivalence result presented in Section 6.1, a similar result holds for the calculus).

Theorem 2 *Let $\mathcal{E}APQL^{-h}$ be defined as $\mathcal{E}APQL$ without considering operators dealing with hierarchies. Let C be the largest complexity class selection predicate evaluation belongs to, when $\mathcal{E}APQL$ is evaluated over a $\mathcal{EPM}(\mathcal{Q},\mathcal{F},\mathcal{V})$ instance. The following conditions hold:*

- *If $C \subseteq PTIME$, then the data complexity for $\mathcal{E}PQL$ is in $PTIME$.*
- *If $C \supseteq PTIME$, then the data complexity for $\mathcal{E}PQL$ is in C.*

Proof Sketch. The basic idea to prove the theorem is to first note that the problem of evaluating a $\mathcal{E}$APQL^{-h} query over a $\mathcal{EPM}(\mathcal{Q},\mathcal{F},\mathcal{V})$ instance can be reduced to the problem of evaluating a query expressed in ALG^{cv-} (algebra for complex-values presented in [AHV95, AB95]) by assuming that: (i) references in the input database are replaced by the corresponding values; (ii) the formula, the data source, and the validity period components are instances of three special domains for which the evaluation of selection predicates has not constant complexity.

It is easy to show that the first step has PTIME complexity in the database size. Moreover, we notice that, under the hypothesis presented in [AHV95, AB95], ALG^{cv-} has PTIME data complexity. However, in our context, this may not be true. In particular, when the complexity C of the selection predicates is higher than PTIME, the cost of selection may dominate data complexity. We also notice that in general the database size does not necessarily dominate the input size for the selection predicate, due to the presence of

presence over data sources. The theorem result follows from the previous considerations. □

As a corollary, we get the following result.

Corollary 1 *Let $\mathcal{E}APQL^{-ph}$ be the algebra defined as $\mathcal{E}APQL^{-h}$ but without considering predicates over data source and formula components. The data complexity of $\mathcal{E}APQL^{-ph}$ when evaluated over a $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ instance (being $\mathcal{V}$ one of T^I, T^U, T^P) is in PTIME.*

Proof Sketch. Under the proposed assumptions, we can prove that $\mathcal{EPM}$, after replacing pattern identifiers with corresponding values, reduces to a complex-value database as defined in [AB95, AHV95]. Similarly, under the hypothesis, $\mathcal{E}APQL^{-ph}$ coincides with ALG^{cv-} which has been demonstrated to be in PTIME [AHV95]. By finally observing that, by using one of the proposed options for the $\mathcal{V}$ theory, all predicates involving pattern validity periods have complexity in PTIME (see Section 6.2.2, Table 6.2), we get the thesis. □

Chapter 7

Design of $\mathcal{PSYCHO}$, a Pattern Based Management System based on $\mathcal{EPM}$

In this chapter we describe the design of a PBMS prototype, named *PSYCHO (Pattern based management System of arCHitecture prOtotype)*, which is based on the model and languages presented in Chapters 4 and 5. Our design is primarily based on object-relational technology. This choice is motivated by the need to store individually both raw data (that can be stored as tables of complex values in a object-relational DBMS) and patterns (that are complex objects which can be easily implemented within an object-relational DBMS).

The remainder of the chapter is organized as follows. We first we point out which are the assumptions we have taken into account in implementing the system (Section 7.1) and, then, we present the reference architecture of the system (Section 7.2). After that, we focus on the physical layer (Section 7.3 and Section 7.4) implementation and the middle layer (Section 7.5) implementation, which constitutes the core of the entire system. The external layer implementation, mainly consisting in the user interfaces, will be described in more details in the next Chapter 8, when presenting the *PSYCHO* language primitives.

7.1 Assumptions

In designing *PSYCHO* (Pattern based management System of arCHitecture prOtotype), we introduce some restrictions over the model for patterns described in Chapter 4 - and, therefore, over the pattern language features proposed in Chapter 5. In particular, we make the following simplifying hypotheses.

1. Pattern hierarchies are supported. However, query operators allowing the navigation through them (i.e., Decomposition, Drill-Down, and Roll-Up) are not implemented.

2. Source data is assumed to be stored in an Object-Relational DBMS.

3. We require that, whenever the user wants to create a new pattern type, its data source already exists in the system. Then, before creating a new pattern type, the user has to ensure that there exists a relation or a view containing the data he wants to use as data source for the instances of the new pattern type, or, if it is the case, create it.

4. The logical theory exploited for pattern formulas is the linear constraint theory over $\mathbb{R}$.

5. The temporal theory exploited for pattern validity is the theory of intervals over $\mathbb{Z}$. The granularity of the intervals can be chosen by the user among the data type for the date and time provided by Oracle.

6. The pattern structure component is always of type TUPLE.

7. Not all the algebraic query operators proposed in Section 5.2.2 are implemented. In particular, the following $\mathcal{E}$APQL operators are not available: (i) measure projection; (ii) reconstruction; (iii) union join; (iv) intersection join.

8. Concerning predicates for patterns, due to the fact that pattern hierarchies are not completely supported, not all the predicates proposed in Section 5.2.1 are implemented. In particular, for pattern equality predicates only shallow equality is provided. The same holds for temporal validity and semantic validity predicates.

9. Finally, concerning manipulation operations, all operators proposed in Section 5.3 are implemented, except for the deletion for which only the restricted version is provided.

Therefore, the model for patterns implemented within $PSYCHO$ is an instance of $\mathcal{EPM}(\mathcal{Q}, \mathcal{F}, \mathcal{V})$ where $\mathcal{Q}$ is the SQL implementation provided by Oracle; $\mathcal{F}$ is the linear constraint theory over $\mathbb{R}$; and $\mathcal{V}$ is the temporal theory of intervals over $\mathbb{Z}$.

7.2 The Reference Architecture

The architecture we implement to realize $PSYCHO$ (Pattern based management System of arCHitecture prOtotype) is based on three different layers, each corresponding to a specific level of abstraction [CMM05]. The whole system architecture is shown in Fig. 7.1, where the software components populating the three layers of the architecture are depicted along with their interactions.

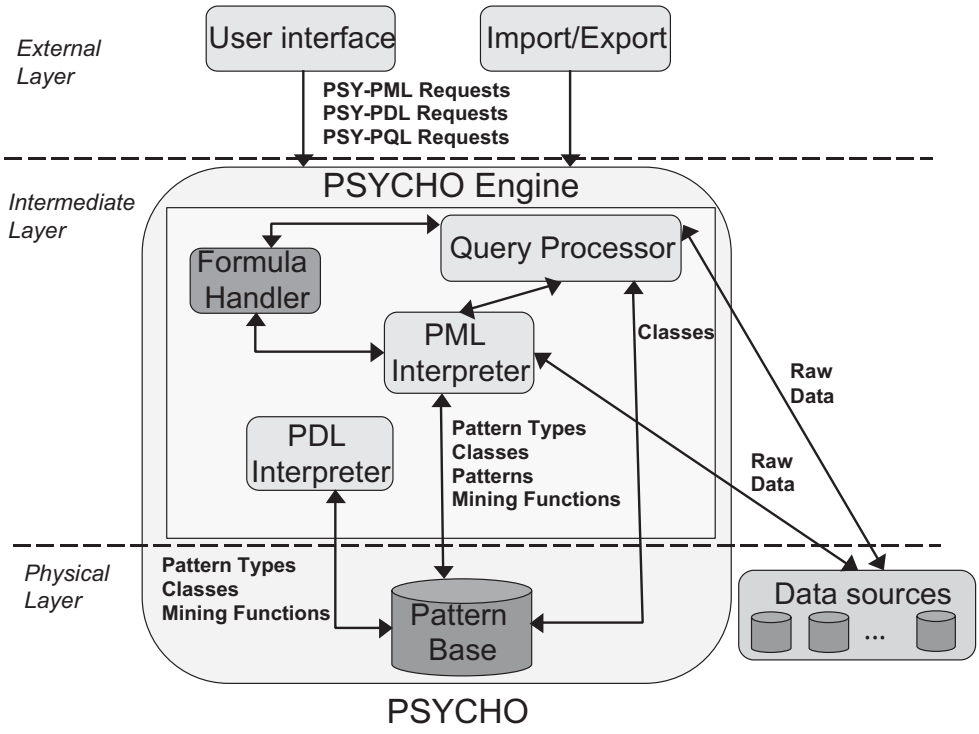


Figure 7.1: The 3 Layers implemented architecture

The *PSYCHO* architecture relies on Oracle [Ora] and Java technologies and can exploit Oracle Data Mining (ODM) server functionalities when dealing with standard data mining patterns and the SICStus Prolog constraint solving engine [sic] for the intensional management of such formulas.

The physical layer contains both the *Pattern Base* and the *Data Source*. The Pattern Base stores pattern types, patterns, and classes; the Data Source stores all raw data from which patterns have been extracted. Therefore, the considered architecture is separated. The middle layer, that we call *PBMS Engine*, coincides with the kernel of the system, and it supports all functionalities for pattern manipulation and retrieval. The PBMS Engine and the Pattern Base represent the core of the *PSYCHO* prototype.

In the following, the physical and middle *PSYCHO* layers are briefly presented, by introducing the basic functionalities of their software components. The external layer corresponds to a set of user interfaces from which the user can send requests to the engine written in specific SQL-like languages, i.e., *PSY* Pattern Manipulation Language (*PSY*-PML), *PSY* Pattern Definition Language (*PSY*-PDL), or in *PSY* Pattern Query Language (*PSY*-PQL), which will be introduced in Chapter 8.

7.2.1 The Physical Layer

The physical layer contains both the *Pattern Base* and the *Data Source*.

Pattern Base. The *Pattern Base* component contains pattern type, pattern, and class definitions and its role is twofold. From one side, it implements the model for patterns, according to the $\mathcal{EPM}$ Model introduced in Chapter 4, and the algebraic operations dealing with patterns, according to the $\mathcal{E}$APQL proposal (see Section5.2.2). From the other side, it hides all the implementation details and choices, giving to the outside only the visibility of the $PSYCHO$ features, but not of the exploited object-relational structures.

We use a database Oracle to store all the data, then, the pattern base component is written in the form of Oracle stored functions and procedures that can be called from outside to ensure the right usage of tables and types used within $PSYCHO$ to internally represent data and patterns. However, all tables, types, and other structures used within the pattern base, cannot be directly accessed by the user, that can interact with the system using only the provided interface functions and procedures.

Data Source. The *Data Source*, in general, can be a distributed database containing raw data from which patterns have been extracted. Various storage technology can be used to store the source datasets: relational or object-relational DBMSs, XML dataset, streams, etc. Within $PSYCHO$, since the main aim is the experimentation of pattern concepts, we assume source data are stored in an Oracle DBMS (see Section 7.1).

7.2.2 The Middle Layer

The middle (or intermediate) layer, represents the engine of the system, which interacts with both the Pattern Base and the Data Source components in the physical layer, and provides processing capabilities .

The *PSYCHO Engine* component is created to raise the level of abstraction of the environment, providing the independence from the data and managing the connections to the Pattern Base. It is written in Java [jav] and its role is to take requests in input - specified using PSY language primitives - and generate calls to the Pattern Base or to the Data Source component, if necessary, preparing the right parameters and calling the right function. In details, after receiving a request coming from the outside higher level, it chooses the right operations to be executed against the Pattern Base and/or against the Data Source and it invokes them. Moreover, it receives the answers and sends them back to the outside layer.

It is logically divided in four different components. Three of them are dedicated to parse, interpret, and execute a different language for pattern management. One is dedicated

to the *PSYCHO-Pattern Definition Language* (PSY-PDL), used to give the definitions of all the entities involved in the pattern management, such as pattern types, patterns, and classes. A second one is dedicated to the *PSYCHO-Pattern Manipulation Language* (PSY-PML) that is used to perform operations such as insertions, deletions, and updates of the patterns. The last of these three components is the query processor, dedicated to the *PSYCHO-Pattern Query Language* (*PSY*-PQL) used to write queries. Finally, within the *PSYCHO* there is a component dedicated to manage the intensional aspects related to pattern formulas. It is called *Formula Handler* and it exploits a Prolog constraint solver engine. More precisely:

PDL Interpreter. This component takes in input the requests written in *PSY*-PDL coming from the higher layer. It translates these requests into calls to the right functions and procedures defined in the Pattern Base dedicated to the entities definition.

PML Interpreter. The input to this component are *PSY*-PML requests. When one of such request is received, the role of the PML Interpreter is to translate it into calls to Pattern Base functions. However, since a *PSY*-PML operation may require the result of a query operation, this module communicates with the Query Processor.

Query Processor. This component takes in input the requests for queries written in *PSY*-PQL, parses them, and defines an access plan for answering the query. The chosen operations are then executed against the Pattern Base and, after having received the results, they are sent back to the external level. As stated above, some query requests can be generated by the PML Interpreter component; in this case, the answer is given directly to that component.

Formula Handler. This component is implemented as a Java module, using the Jasper package [jas] to interact with SICStus Prolog environment [sic]. It is exploited by the Query Processor and the PML Interpreter modules whenever the execution of intensional predicates is required.

7.2.3 Communication Between Layers

An important aspect about the given architecture is how the different components communicate between them. We start our description bottom-up, from the lower layer to the higher.

The Pattern Base, as stated before, is integrated within the DBMS. The *PSYCHO* Engine is placed immediately above the Pattern Base. It creates and manages the connection with the DBMS and the calls to the functions defined in the Pattern Base. The communication is, therefore, the classical communication between a piece of Java code and the DBMS, through the JDBC driver.

On the other hand, the communication between the *PSYCHO* Engine and the external layer is established using sockets. In this way the system is more flexible and a completely distributed architecture can be realized. In details, the *PSYCHO* Engine opens a socket on a fixed port and waits for connections from the outside. When the external module needs to communicate with the server, it makes a connection, creates a serializable object that encapsulate the request, and sends it to the *PSYCHO* Engine. In this way, using serialization, sockets and JDBC driver, the different components corresponding to patterns and data sources, the engine, and external modules can be placed on different hosts.

7.3 Physical Layer: the Pattern Base

In the following we analyze the *PSYCHO* design of the pattern base, describing how pattern types, patterns, and classes are represented inside an object-relational schema.

7.3.1 Pattern Type

We start our analysis by presenting the PSYCHO implementation of the pattern type. In the object-relational environment, the first thing to do when creating objects is to define an Abstract Data Type that represents those objects. Within *PSYCHO*, the objects are the patterns, and the ADTs representing them are the pattern types. Therefore, we define a pattern type as an Abstract Data Type, called *PatternType*, that contains all the components required by the model (the name, the structure schema, the data source schema, the measure schema, the formula schema and the validity schema). This ADT will be the *ancestor* of all the pattern types that can be inserted in the system. The definition of the PatternType is presented in Fig. 7.2. Whenever a new pattern type is to be defined, then, a new ADT will be created inheriting from *PatternType*. As a consequence, a pattern within PSYCHO is an object instance of a certain pattern type.

In details, the ADT *PatternType*, has four fields representing respectively the structure, the data source, the measure, and the validity period (the name component corresponds to the name of the ADT specializing the root PatternType). In the following, these fields will be presented in more details, discussing how they are implemented within PSYCHO.

Structure. The structure is modeled as a component of type *PatternStructure*. We notice that, the structure is a fully context-dependent field that, according to the pattern model, will be provided at pattern type creation time. Thus, in the ADT *PatternType*, we just add a field to represent an empty structure and a method *Equals* that returns a *boolean* and is used to perform an equality test between *PatternStructure* objects. The default implementation returns always *FALSE*; thus, the user have to redefine it. The definition

of the *PatternStructure* ADT is presented in Fig. 7.2 (lines 1-10).

Data Source. One of the pre-requirement of the design is that when a new pattern type is created, its data source is already present in the system. In details, we require that there exists a relation or a view in the system that can be used as data source for the pattern type under construction. Then, in the ADT *PatternType* the data source is simply stored as a field of type string, that will contain the name of the relation or view used as source dataset (Fig. 7.2).

Measure. The measure is modeled as a component of type *Pattern Measure*. Like in the case of the structure, also the pattern measure is defined when a specific pattern type is created, since it can vary from a pattern type to another. Similarly to what we made for the structure, no attributes are provided. However, we insert a method (*Theta*) comparing two measures. The definition of the *Pattern Measure* ADT is presented in Fig. 7.2.

Formula. The formula binds the pattern to the data source, specifying the portion of raw data the pattern represents. It can be used to test which elements in the data source are effectively represented by the pattern. In our system, we translate this test into a boolean method, named *formulaE*, defined for the ADT *PatternType*. The body of such method is left empty, and the user has to define it when creating new pattern types by extending *PatternType*. This method can access both the data source and the structure of the generic pattern type (i.e., fields *d* and *s* in *PatternType*) and it has to be written directly using an Oracle PL/SQL syntax. Moreover, within *PSYCHO* a support for an intensional, declarative representation of pattern formula is provided. Under this approach the pattern formula is viewed an intensional Prolog predicate completely managed by the Formula Handler (see Section 7.5.4 for details).

```
1:  CREATE TYPE PatternStructure AS OBJECT (
2:  MEMBER FUNCTION Equals (other PatternStructure) RETURN BOOLEAN)
3:    NOT INSTANTIABLE NOT FINAL;
4:
5:  CREATE TYPE PatternMeasure AS OBJECT (
6:    MEMBER FUNCTION Theta (other PatternMeasure) RETURN BOOLEAN)
7:  NOT INSTANTIABLE NOT FINAL;
8:
9:  CREATE TYPE ValPeriod AS OBJECT (min TIMESTAMP, max TIMESTAMP);
10:
11: CREATE TYPE PatternType AS OBJECT (s PatternStructure, d VARCHAR2(20),
12:   m PatternMeasure, v ValPeriod,
13:   NOT INSTANTIABLE MEMBER FUNCTION formulaE RETURN BOOLEAN)
14: NOT INSTANTIABLE NOT FINAL;
```

Figure 7.2: Generic *PatternType* ADT and support ADTs

Validity Period. The validity period is stored as a pair of *TIMESTAMP* values; this is

obtained by defining a new ADT, called *ValPeriod*. The definition of the *ValPeriod* ADT is presented in Fig. 7.2.

Starting from type *PatternType* any user-defined pattern type can be defined as a subtype of PatternType. To this purpose, the steps to do are the following:

- the types *PatternStructure* and *PatternMeasure* have to be refined to deal with the actual pattern component types; functions Equal and Theta have to be refined accordingly;
- the function representing the extensional pattern formula has to be overridden.

Fig. 7.2 presents an example of creation of the pattern type *AssociationRule* inside the Physical Layer. The ADT that inherits from *PatternType* which models the proper pattern type is called *AssociationRule*, and those inheriting from *PatternStructure* and *PatternMeasure*, are called *AssociationRuleStructure* and *AssociationRuleMeasure*, respectively.

```
1:  CREATE TYPE AssociationRuleStructure
2:    UNDER PatternStructure(head TABLE OF VARCHAR2(20), body TABLE OF VARCHAR2(20));
3:
4:  CREATE TYPE AssociationRuleMeasure
5:    UNDER PatternMeasure(support real,confidence real,
6:    OVERRIDING MEMBER FUNCTION Theta (other AssociationRuleMeasure)
7:    RETURN BOOLEAN );
8:
9:  CREATE TYPE BODY AssociationRuleMeasure AS
10:   OVERRIDING MEMBER FUNCTION Theta (other AssociationRuleMeasure)
11:   RETURN BOOLEAN IS
12:   BEGIN
13:     IF SELF.support >= other.support AND SELF.confidence >= other.confidence
14:     THEN RETURN TRUE;
15:     END IF;
16:     RETURN FALSE;
17:   END Theta;
18: END;
19:
31: CREATE TYPE AssociationRule UNDER PatternType (
32:   OVERRIDING MEMBER FUNCTION formula RETURN BOOLEAN );
33:
34: CREATE TYPE BODY AssociationRule AS
35:   OVERRIDING MEMBER FUNCTION formula RETURN BOOLEAN IS
36:     CURSOR c1 IS SELECT transaction FROM AssRuleDataSource
37:   BEGIN
38:     FOR ds_rec IN c1 LOOP
39:       FOR i IN s.head.FIRST ..  s.head.LAST LOOP
40:         IF NOT memberOf( s.head(i), ds_rec.transaction )
41:         GOTO next_transaction;
42:         END IF;
43:       END LOOP;
44:       FOR i IN s.body.FIRST ..  s.body.LAST LOOP
45:         IF NOT memberOf( s.body(i), ds_rec.transaction )
46:         GOTO next_transaction;
47:         END IF;
48:       END LOOP;
49:       RETURN TRUE;
50:     <<next_transaction>>
51:     END LOOP;
52:   RETURN FALSE;
53:   END formula;
54:   END;
```

Figure 7.3: Creation of the *Association Rule* type and support types and functions

7.3.2 Patterns

According to what stated in Chapter 4, when presenting $\mathcal{EPM}$, patterns are instances of a certain pattern type. Thus, within *PSYCHO* system, they correspond to objects that are instances of the ADT implementing the corresponding pattern type.

In order to represent the extension of a given pattern type inside the Pattern Base, for each created pattern type, a typed table over such ADT is created. When a pattern is created, it is automatically inserted in the typed table associated with its type. For instance, when the *AssociationRule* type is created in the system, also the corresponding typed table, named *AssociationRuleTAB*, is created. In this way, there is not the need of explicitly inserting a PID field, since the *rowid* that is automatically associated each row of the typed table can be used for this purpose (see Fig. 7.1).

7.3.3 Classes

According to the $\mathcal{EPM}$ model presented in Chapter 4, a class is a collection of patterns that are instances of the same pattern type for which the class has been defined for. Recall that, a pattern can belong to zero, one, or more classes.

In *PSYCHO*, a class can be defined as a table containing just the references to the patterns belonging to it. In this way there is no duplication of information.

7.3.4 System Catalog

According to what stated in Chapter 4, two different patterns, even if they are instances of the same pattern type, may be generated by two different mining functions. The same holds for measure functions, used to calculate the value of the measure component of a pattern instance of a specific pattern type. Since information concerning mining and measure functions are useful from an operational point of view, but they are not queried by final users, within *PSYCHO* such information is stored in some system catalogs. Actually, two different system catalogs are created: a first one for the pattern type layer and a second one for the pattern layer.

The *PSYCHO* pattern type system catalog contains, for each pattern type, information concerning mining and measure functions defined for it. We assume there exist some libraries of mining and measure functions for each pattern type. However, *PSYCHO* provides the user the capability to define specific mining and measure functions (as we will see in Chapter 8), stored in the system as stored procedures. Therefore, once a new function is created for a pattern type in the system, its information are recorded in the

pattern type catalog.

On the other hand, the *PSYCHO* catalog for the pattern layer contains for each pattern, the following information: (i) a reference to the table in the pattern layer containing the pattern; (ii) the name of the specific mining function used to generate it; (iii) the name of the measure function used to evaluate its measures; (iv) the pattern transaction time, i.e., the timestamp at which the pattern has been generated and inserted in the system.

The usage of the information stored in the system pattern catalog will become clearer in the Section 7.4, when presenting the implementation of pattern manipulation and querying operations at the Pattern Base layer.

7.4 Physical Layer: Pattern Languages within the Pattern Base

At the physical layer, operators for the languages introduced in Chapter 5 are implemented as PL/SQL procedures, which are invoked by the upper layer during the computation. On the other hand, query predicates, presented in Section 5.2.1 are modeled as PL/SQL functions. In the following, we describe how the manipulation operations (insertion, update, deletion, and operations for classes) are implemented within *PSYCHO* at the physical layer (Section 7.4.1) and, then, we discuss query operators and predicates implementation (Section 7.4.2).

7.4.1 $\mathcal{E}PML$ Primitives within *PSYCHO*

According to what stated in Section 5.3, manipulation operations need to use the mining functions and the measure functions defined for the proper pattern type. Indeed, several manipulation operations take those functions as input parameter. For example, the pattern extraction operation take in input, besides the target pattern type, a proper mining function to be used to generate new patterns. As we have already said in Section 7.3.4, within *PSYCHO* the information concerning mining and measure function is stored in the system catalog. Therefore, we assume they already exist in order to be used in the context of a manipulation operation.

Besides mining and measure function, also filtering conditions can be passed as an input parameter to a manipulation operation. Therefore, there is the need to create those conditions and to pass them as operators' parameters. To this end, they are implemented in *PSYCHO* as *stored functions* taking in input a pattern (i.e., an object of a type that is heir of *PatternType*) and returning a *BOOLEAN* value expressing whether the given

pattern respect the condition or not. A name is assigned to a condition in order to be able to further re-use it. Thus, when a condition has to be passed as a parameter to some operation, the name of the stored function implementing it can be used.

All the $\mathcal{E}PML$ insertion operations are implemented within the $PSYCHO$ pattern base by using PL/SQL stored procedures. The signature of the procedures corresponds to the ones presented in Chapter 5 (see Fig. 7.4).

```
1:  CREATE PROCEDURE Extraction (Name VARCHAR2, d VARCHAR2, mu VARCHAR2,
2:      cond VARCHAR2, v ValPeriod)
3:  CREATE PROCEDURE Insertion (d VARCHAR2, structure PatternStructure,
4:    v ValPeriod)
5:  CREATE PROCEDURE Recomputation (cond VARCHAR2, d VARCHAR2, mu_m VARCHAR2,
6:    v ValPeriod)
7:  CREATE PROCEDURE ClassInsert (p REF(PatternType), className VARCHAR2)
8:  CREATE PROCEDURE ClassDelete (p REF(PatternType), className VARCHAR2)
9:  CREATE PROCEDURE Deletion (Name VARCHAR2, cond VARCHAR2)
10: CREATE PROCEDURE NewPeriod (Name VARCHAR2, cond VARCHAR2, v ValPeriod)
11: CREATE PROCEDURE Synchronize (Name VARCHAR2, cond VARCHAR2, mu_m VARCHAR2)
12: CREATE PROCEDURE Validate (cond VARCHAR2, mu_m VARCHAR2)
```

Figure 7.4: Procedure implementing $\mathcal{E}PML$ operations within the Pattern Base

7.4.2 $\mathcal{E}PQL$ Primitives within $PSYCHO$

In this section we analyze how the query operators and the predicates that they can use (both over pattern components and over entire patterns) are implemented within the $PSYCHO$ pattern base, at the physical layer.

According to what stated in Section 5.2, querying operations need to use predicates over patterns and their components. The semantics of the query predicates implemented within $PSYCHO$ is the one presented in Section 5.2.1. As we have already said, at the $PSYCHO$ physical layer query predicates over patterns are modeled as PL/SQL functions taking as input elements of pattern type and returning boolean value (see Fig. 7.6). The same holds for query predicates acting over pattern components, in this case the input parameters are objects of the proper type (for example in Fig. 7.5 predicates over pattern data sources are presented). On the other side, predicates dealing with validity period pattern component are modeled as member function of the *ValPeriod* ADT (see Fig. 7.7).

```
1:  CREATE FUNCTION iEquiv (ds1 VARCHAR2, ds2 VARCHAR2) RETURN BOOLEAN;
2:
3:  CREATE FUNCTION eEquiv (ds1 VARCHAR2, ds2 VARCHAR2) RETURN BOOLEAN;
4:
5:  CREATE FUNCTION eContain (ds1 VARCHAR2, ds2 VARCHAR2) RETURN BOOLEAN;
```

Figure 7.5: Functions implementing predicates over pattern data sources

```
1:  CREATE FUNCTION Identity (p1 PatternType, p2 PatternType) RETURN BOOLEAN;
2:  CREATE FUNCTION SEquals (p1 PatternType, p2 PatternType) RETURN BOOLEAN;
3:  CREATE FUNCTION isTValid (p PatternType, t TIMESTAMP) RETURN BOOLEAN;
4:  CREATE OR REPLACE FUNCTION wSubsumes (p1 PatternType, p2 PatternType)
5:    RETURN BOOLEAN;
6:  CREATE FUNCTION isBetterOf (p1 PatternType, p2 PatternType)
7:  CREATE FUNCTION isSValid (p PatternType, thresholds PatternMeasure)
8:    RETURN BOOLEAN;
```

Figure 7.6: Functions implementing predicates over patterns

```
1:  CREATE OR REPLACE TYPE ValPeriod AS OBJECT (min TIMESTAMP, max TIMESTAMP,
2:    MEMBER FUNCTION equals (other ValPeriod) RETURN BOOLEAN,
3:    MEMBER FUNCTION equalsC (const TIMESTAMP) RETURN BOOLEAN,
4:    MEMBER FUNCTION before (other ValPeriod) RETURN BOOLEAN,
5:    MEMBER FUNCTION beforeC (const TIMESTAMP) RETURN BOOLEAN,
6:    MEMBER FUNCTION meets (other ValPeriod) RETURN BOOLEAN,
7:    MEMBER FUNCTION overlaps (other ValPeriod) RETURN BOOLEAN,
8:    MEMBER FUNCTION during (other ValPeriod) RETURN BOOLEAN,
9:    MEMBER FUNCTION duringC (const TIMESTAMP) RETURN BOOLEAN,
10:   MEMBER FUNCTION starts (other ValPeriod) RETURN BOOLEAN,
11:   MEMBER FUNCTION finishes (other ValPeriod) RETURN BOOLEAN );
```

Figure 7.7: Functions implementing predicates over pattern validity periods

According to what stated in Chapter 5, queries over the pattern base apply over one or more classes. Recall that, in some case, query results may requires the definition of a new pattern type and a new class containing objects of the new type. Both the new type and the new class created are temporary entities, since they exist only for the duration of the session in which they are defined. However, it seems to be useful that the user can decide to make persistent (one of) these entities, this new feature is supported in *PSYCHO* by allowing the user to specify a name for the pattern type or class resulting by the execution of a query.

Within the *PSYCHO* Pattern Base, all the operations supporting the query execution are implemented as *stored functions*, taking in input class names, creating a new class containing the result, and returning its name. The results of different operations (that are different classes) can be combined together or used in other queries.

In the following, we briefly review technical aspects concerning the *PSYCHO* implementation of some query operators, whose implementation is slightly different from the operations proposed in Section 5.2.2.

Projection and Selection. Within *PSYCHO* Pattern Base, standard operations such as *projection*, and *selection* are obtained directly by the query language of the object-relational dbms. The stored functions for these operations are called, respectively, *opProject* and *opSelect*. Concerning projection, we wish to outline that it does not explicitly corresponds to an $\mathcal{E}APQL$ operations, rather it is an extension of the traditional projection over object components. More precisely:

- *opProject*: it takes in input a class name and an array of *VARCHAR2* containing the names of the fields over which project. The result, after a check of the correctness of the names with respect to the pattern type of the class, is a new class (whose name is returned) containing only the needed fields. The signature of the *opProject* function is shown in Fig. 7.8 (lines 1-2).
- *opSelect*: it takes in input a class and a condition and returns a new class, with the same structure of the original, containing only the patterns that respect the condition. The name of this class can be specified with *tabName*, otherwise, with a null value, its name is chosen by the system. The signature of the *opSelect* function is shown in Fig. 7.8 (lines 3-4).

Join. In the current *PSYCHO* version the join operation implemented is a special case of the general join operator presented in Section 5.2.2. Indeed, in order to apply the join over two classes $class_1$ and $class_2$, defined for pattern types pt_1 and pt_2, respectively, it is required they have the same data source. If this mandatory condition is verified, the join operation creates a new pattern type, let us call it $pt_1 J pt_2$, with the following fields:

- Structure: a new ADT, named $pt_1 J pt_2 Structure$, is created to represent the structure of the joined types. In details, $pt_1 J pt_2 Structure$ contains two fields, one of type $pt_1 Structure$ and the other of type $pt_2 Structure$. In this way there is the full access to both the structures of the starting types.
- Data Source: since both pt_1 and pt_2 have the same data source, also this new pattern type have the same one.

- Measures: similarly to the case of the structure component, a new ADT $pt_1Jpt_2Measure$, containing both the starting measures, is created.
- Formula: since the formulas are stored as text, the new formula representing the joined one can be obtained combining the codes of the formulas of the input pattern type. In details, all the local declared variables are put together in the common declaration and the two different codes are conjuncted with an *and* operator. Moreover a renaming of the fields referring the structure is needed, but it becomes simply an adding of a prefix (exploiting the dot notation).
- Validity Period: validity period values of the resulting patterns will be the intersection between the validity periods of the involved patterns.

Within the Pattern Base, the function performing the join is called *opJoin*; it takes in input the names of two classes and a join condition. The join condition, that can involve patterns of both classes, is given with the name of a stored procedure (that have to be already defined) that takes two pattern in input. In this way it can refer the structures of both the patterns. To contain the result, a new class for the type pt_1Jpt_2 is created. For each pair of patterns p_1 and p_2 such that $p_1 \in class_1$ and $p_2 \in class_2$, if the given condition $F(p_1, p_2)$ is verified (i.e., the given function called over p_1 and p_2 returns $TRUE$) the pattern is inserted into the new class. The return value is the name of the new class created. The signature of the *opJoin* function is shown in Fig. 7.8 (lines 5-6).

```
1:  CREATE FUNCTION opProject(className VARCHAR2, selNames CHARARRAY)
2:  RETURN VARCHAR2;
3:  CREATE FUNCTION opSelect(className VARCHAR2,cond VARCHAR2, tabName VARCHAR2)
4:  RETURN VARCHAR2;
5:  CREATE FUNCTION opJoin(class1 VARCHAR2, class2 VARCHAR2, cond VARCHAR2)
6:  RETURN VARCHAR2;
7:  CREATE FUNCTION opDrillThrough(className VARCHAR2,cond VARCHAR2)
8:  RETURN VARCHAR2;
9:  CREATE FUNCTION opDataCovering(d VARCHAR2, p PatternType)
10: RETURN VARCHAR2;
```

Figure 7.8: Functions implementing querying operators

7.5 Middle Layer: the $PSYCHO$ Engine

The $PSYCHO$ middle (or intermediate) layer consists of the *PSYCHO Engine* component, whose aim is to execute requests (concerning both queries or manipulation operations) delivered by the external layer, using the Pattern Base and Data Sources when required. As already said, the $PSYCHO$ Engine has been implemented in Java and is logically divided

into four main sub-modules: the *PDL Interpreter*, the *PML Interpreter*, the *Query Processor*, and the *Formula Handler*. The first three modules are dedicate to parse, interpret, and execute *PSY*-PDL, *PSY*-PML, and *PSY*-PQL requests, respectively. The Formula Handler module deals with the declarative representation of pattern formulas solving intensional predicates involving them.

From a functional point of view, the behavior of the *PSYCHO* Engine can be summarized as follows:

1. it creates the connections to the Pattern Base and to the Data Source;
2. it opens a socket waiting a connection from the external layer;
3. it enters in a main loop in which waits for incoming requests;
4. when a request is received, it is analyzed to establish which module is involved, and then, depending on its nature, passed to the PDL Interpreter, the PML Interpreter, or the Query Processor; those modules, in turn, process the request and generate the call for the physical layer structure or for the Formula Handler, when required;
5. if an answer is generated, it sent back to the external layer via socket.

In the remainder of this section, we present in more details the PDL Interpreter (Section 7.5.1), the PML Interpreter (Section 7.5.2), the Query Processor (Section 7.5.3), and the Formula Handler (Section 7.5.4).

7.5.1 PDL Interpreter

This component takes in input the requests written in *PSY*-PDL coming from the external layer and translates them into calls to the right functions and procedures defined at the physical layer - i.e., in the Pattern Base - dedicated to the entities definition. It is implemented as a Java class providing a method for each *PSY*-PDL command, besides several other methods dedicated to the initialization of the communication involving the module.

In particular, within the PDL Interpreted we may device two main groups of methods: the one dealing with *PSY*-PDL creation statements and the ones dealing with *PSY*-PDL dropping statements.

The *PSY*-PDL creation statements allows the user to define different kinds of entities, i.e., pattern types, classes, mining functions, and measure functions. Therefore, PDL Interpreter methods dealing with this kind of commands, when receiving a *PSY*-PDL creation statement extract all the specified parameters and prepare and execute the corresponding

calls to data definition language commands within the Pattern Base. The effective translation from of each *PSY*-PDL command into calls to data manipulation operation executed over the Pattern Base will be presented and discussed in Chapter 8, when presenting *PSY*-PDL primitives in detail.

A special treatment is deserved to mining function and measure function creation. Indeed, each new mining or measure function is implemented within the physical layer as stored functions. Thus, the PDL Interpreter translates this statement into the creation of a stored function whose name is the one specified by the user and with body the PL/SQL code specified by the user. The return value is a VARCHAR object representing the name of a table in the Pattern Base where patterns generated by the execution of the mining function are stored. Similarly, the PDL Interpreter translates each *PSY*-PDL statement for creating new measure function into a command for the definition of a new stored function within the Pattern Base, having the name and the body specified by the user. The return value is a *PatternMeasure* object instantiated over the specific chosen pattern type.

On the other side, the *PSY*-PDL dropping statements have an easy syntax, since the only parameter needed is the name of the entity to drop (see Chapter 8). Therefore, they can be easily translated by corresponding methods of the PDL Interpreter into commands over the Pattern Base structures. Some of them are call to the corresponding Pattern Base procedure (such as, in the case of the *PSY*-PDL comand requiring the dropping of a pattern class, which corresponds to the physical layer to a call to the procedure 'DeleteClass' implemented within the Pattern Base); some other are symply call to data manipulation Oracle statement (such as, in the case of the *PSY*-PDL command dropping a certain pattern type, which correspond in the physical layer to an SQL drop type command).

7.5.2 PML Interpreter

The PML Interpreter receives requests written in *PSY*-PML, by the external layer, for inserting, deleting or manipulating patterns, inserting patterns in classes, and removing patterns from them. Once received, it translates these requests to the right call(s) to the Pattern Base communicating, if necessary, with the *Query Processor* to execute a query that can be part of a *PSY*-PML statement (see below). In the following, we discuss PML Interpreter behaviour over *PSY*-PML statements grouping them into three classes: (i) insertion statements, (ii) update statements and deletion statements; (iii) statements for classes. The syntax of the *PSY*-PML commands will be described in more details in Chapter 8.

7.5.2.1 Insertion Statements

The insertion operations provided by the *PSYCHO* Engine is the one proposed in Section 5.3.

As we have already said, within the *PSYCHO* Pattern Base, the extraction is implemented with a stored function, named *Extraction*; thus, the PML Interpreter simply translates the `EXTRACT PATTERNS` *PSY*-PML command into a call to this function.

Similarly to the case of the extraction operation, ther insertion operation are implemented within the *PSYCHO* the Pattern Base with stored functions, named *Insertion* and *Recomputation*, respectively, that the PML Interpreter call in order to execute the corresponding *PSY*-PML commands.

7.5.2.2 Update Statements

According to the $\mathcal{E}PML$ proposal presented in Chapter 5, the update operations implemented within *PSYCHO* are *New Period*, *Synchronize*, and *Validate*, whereas the *deletion* operation implemented corresponds to the restricted version of the delete operators.

All the update operations are implemented by a *PSY*-PML statement starting with the keywords `UPDATE PATTERNS` (see Chapter 8 for details concerning the proper syntax). All of these statements require may involve the specification of an optional parameter specifying a filtering condition, indicating which patterns have to be updated. Such condition can be defined within the *WHERE* clause of the `UPDATE PATTERNS` *PSY*-PML statement. However, in the following, we do not presents examples having a filtering condition, for more details we refer to the next Chapter 8.

In order to execute an update or a deletion *PSY*-PML command the PML Interpreter extract parameters from the input statement ans it translates it into a call to the proper stored procedure defined in the Pattern Base.

7.5.2.3 Class Statements

The manipulation operations involving classes, supported by *PSY*-PML, are insertion and deletion of patterns into or from an existing class defined for the proper pattern type. All these statements may have an optional filtering condition indicating which patterns have to be updated. In this case the *WHERE* clause is used (see Chapter 8 for details).

The PML Interpreter, when receives one of this command, translates it into a call to the proper stored procedure defined in the Pattern Base, that performs the insertion into class (*ClassInsert*) or the removal from a class named *ClassDelete*.

7.5.3 Query Processor

This component takes in input the requests for queries written in *PSY*-PQL. Its role is to receive the requests, parse them, and decide how the query has to be executed. In details it establish whether the *PSY*-PQL requests, in order to be solved, require an intensional processing (see Section 7.5.4) or an extensional processing and, in this case, whether the Pattern Base is enough to solve them or also the data source has to be used. After, that it translates the *PSY*-PQL request in a query session to be posed to the Pattern Base and/or to the DBMS handling data sources. Then, it waits for the answer and, after having received and collected the results, it send them back to the higher level from which the request has been generated. We notice that, whenever an intensional processing is required in order to solve the *PSY*-PQL request, the Formulas Handler module is invoked. Thus, the Query Processor collects the outputs from it and it integrates these results with the query outputs obtained from the Pattern Base or the Database, in order to generate the answer for the external layer.

As stated above (Section 7.5.2), some requests to the Query Processor can be generated, within the *PSYCHO* Engine, by the PML Interpreter component; in this case, the answer is send back directly to that component, without going to the outside. This happens whenever a filtering condition is specified within a manipulation operation, by means of the *WHERE* clause (for more details concerning the syntax specification of all *PSY*-PML operations we refer to Chapter 8).

In the remainder of this section, we first we analyze how the Query Processor supports conditions (Section 7.5.3.1) and, then, we discuss the behavior of the Query Processor when dealing with the execution of a query (section 7.5.3.2).

7.5.3.1 Conditions

According to what stated in Section 7.4, handling filtering conditions is a crucial aspect within *PSYCHO*. As we have already stated, they are used not only in query operations, but also in manipulation operations. The conditions we manage are a sort of functions with a name and a code, the user can specify, that can be stored in the system and used more than one time. The user can define its own conditions by using the *PSY*-PDL command `CREATE CONDITION` and writing their body using PL/SQL. The only parameter required when creating a new condition consists in a pattern instance of a certain pattern type. The user can access the fields of the pattern in order to write the condition. It is mandatory to specify when the condition on the given pattern instance is true (returning an integer value 1) or false (returning 0).

The declaration of a condition through the `CREATE CONDITION` *PSY*-PDL command is translated at the Pattern Base layer into the definition of a new stored function with the

same name specified by the user for the condition and with just a parameter, representing a pattern instance of the pattern type for which the condition is defined.

Whenever a user needs to use a condition, in order to choose to which patterns apply a manipulation operation or a query operation, she simply uses the name of the condition in the *WHERE* clause of the *PSY* command. Then the PML Interpreter or the Query Processor, depending on the case, analyze the *PSY* statement and invokes the function implementing the filtering condition with the right parameters.

A simple example of the definition of a condition and its usage within a manipulation operations is presented in Fig. 7.9. The created condition is named *testCond* and it is defined for patterns of type *ExampleType*; *p* is the variable representing the generic pattern to be used in the body of the condition. Such a condition is, then, used to customize the extraction statement concerning patterns of type *ExampleType* (lines 9-13).

```
1:  CREATE CONDITION testCond FOR ExampleType p AS ret INTEGER;
2:  BEGIN
3:    IF p.field1 > 2 AND p.field2 > 5 THEN ret := 1;
4:    ELSE ret := 0;
5:    END IF;
6:    RETURN ret;
7:  END;
8:
9:  EXTRACT PATTERNS OF ExampleType
10: FROM datasource
11: USING ExampleFunction
12: WHERE testCond
13: VALID FROM 03-09-2005 TO 04-21-2005;
```

Figure 7.9: Example of the creation and usage of a condition

7.5.3.2 Selection Statement

The parsing, interpretation, and execution of the *PSY*-PQL selection statements are the main purposes of the Query Processor module. For more details about supported query features we refer to Section 8.1.3. In the following, we summarize the behavior of this *PSYCHO* component.

Whenever the Query Processor receives a *PSY*-PQL request, it parses it in order to understand which type of request is and which are the conditions involved. Within *PSYCHO* four different types of queries operations have been implemented: (i) selection; (ii) drill through; (iii) data covering; (iv) pattern covering. Therefore, first of all, the Query Processor establishes whether a *PSY*-PQL request is a cross-over query or not, i.e., if it requires to access just the pattern base structure, in order to be solved, or if it needs to access the

data source structures. Then, it analyzed the query condition in order to establish whether it involves or not the evaluation of an intensional predicate.

If no intensional predicate execution is required, the Query Processor translates the *PSY*-PQL request into calls to the physical layer structure (i.e., to the Pattern Base or to the Database), by setting the right parameters, and it waits for the result. Once the query have been executed at the physical layer, the Query Processor catches the results and can sends them back to the external layer.

Whenever also an intensional predicate execution is required, the Query Processor separates the *PSY*-PQL request into two logical parts: one containing only extensional aspects and the other concerning the intensional aspects. The extensional part of the request is translated into calls to the physical layer structures; whereas the intensional part is delivered to the Formula Handler which manages it (see Section 7.5.4). Once both the physical layer and the Formula Handler module returns their outputs to the Query Processor, it combine the results and it prepares and sends the response to the external layer.

We notice that, the final response is delivered to the external layer, by creating a *Resultset* Java object containing the selected patterns that will be send (using the socket connection established between the *PSYCHO* Engine and the external layer during the initialization phase) to it.

7.5.4 Formula Handler

This component is a Java module managing the declarative representation of pattern formulas. As we have already pointed out in Section 7.1, within the *PSYCHO* implementation pattern formulas are represented by using linear constraints over Reals. The constraint solver exploited to manage this kind of constraint is the one provided by SICSTus Prolog [sic]. Therefore, essentially, the Formula Handler constitutes a software interface toward the constraint solver environment, which otherwise cannot be integrated within the object-relational environment used to manage data and patterns (e.g., in our case Oracle10g). The software tool enabling the communication between the Java environment and the SICStus environment is the Jasper package [jas].

This software module is invoked by both the Query Processor and the PML Interpreter modules whenever the execution of intensional predicates is required involving pattern formulas is required. In details, a call to the Formula Handler module is required whenever a *PSY* operation involves a condition applying an intensional predicate over pattern formulas. We recall that, an intensional predicate is checked at a declarative layer without accessing source data. Actually, within *PSYCHO* two distinct intensional predicates are provided: a first one, called *IEQUIV*, checking for equivalence between intensional formulas and a second one, called *ICONTAIN* checking whether a formula contains another

formula, i.e., whatever is the considered data source, all the solutions of the first formula are contained in the set of solutions of the second one.

For more details concerning intensional predicates and how intensional condition can be specified within *PSYCHO* queries we refer to Section 8.1.3.

7.6 External Layer

Within *PSYCHO*, an external layer module enabling the communication between the user and the *PSYCHO* Engine is provided. In details, this module is a textual interface in which the user can insert statements expressed using one of the provided languages (i.e., *PSY*-PDL, *PSY*-PML, and *PSY*-PQL). The module, when activated, makes connection request to the *PSYCHO* Engine using sockets. When the connection is established, it is ready for sending requests to the *PSYCHO* Engine and waiting for the answers.

When a *PSY*-PDL or a *PSY*-PML operation is specified, the answer returned by the *PSYCHO* Engine will be a message containing the response of the statement execution, that is displayed just to give a feedback to the user. On the other side, if the executed statement is a *PSY*-PQL query, the answer returned to the user is a *Resultset* Java object (see Section 7.5.3.2). In this case the data contained in the *Resultset* are read and displayed in the form of a table.

As already said, we will present in more details the user interface in Chapter 8, where the primitives of all three *PSYCHO* languages will described.

Chapter 8

$\mathcal{PSYCHO}$ User Interface

The user interacts with *PSYCHO* by using a textual interface which is SQL-enabled. According to the *PSYCHO* reference architecture presented in Section, such interface constitutes the main module of the *PSYCHO* external layer and it accepts and processes *PSYCHO* language commands as well as SQL and PL/SQL commands written in an SQL-like syntax. We outline that, the languages for pattern manipulation and querying implemented within *PSYCHO* adhere to the specification of the languages proposed in Chapter 5. Therefore, the implementation of *PSYCHO* languages results system independent. However, since *PSYCHO* implementation is based on Oracle technology, we instantiate the language primitives with respect to Oracle.

In the following we first introduce language primitives provided within *PSYCHO*, by discussin their syntax and explaing their behaviour (Section 8.1), then, based on them, several examples covering *PSYCHO* usage are presented (Section 8.2).

8.1 $\mathcal{PSYCHO}$ Languages

In this section, *PSYCHO* languages, used to define and manipulate basic *PSYCHO* elements, and to query patterns, are presented. In particular, Section 8.1.1 concerns the Pattern Definition Language ($\mathcal{PSY}$-PDL), Section 8.1.2 deals with the Pattern Manipulation Language ($\mathcal{PSY}$-PML) and Section 8.1.3 presents the Pattern Query Language ($\mathcal{PSY}$-PQL).

8.1.1 $\mathcal{PSY}$-PDL: $\mathcal{PSYCHO}$ Pattern Definition Language

The *PSYCHO* Pattern Definition Language ($\mathcal{PSY}$-PDL) allows one to define new pattern types, new mining functions, new measure functions, new classes, and to remove them from the PBMS using an SQL-like syntax. Moreover, it allows the user to define specific functions to be used in queries, whenever complex conditions have to be specified in $\mathcal{PSY}$-PQL queries. In the following, each operation is described in details.

8.1.1.1 Operations for Pattern Types

```
1:  <Create_Pattern_Type> ::=
2:  CREATE PATTERN TYPE <Pattern_Type_Name>
3:  STRUCTURE <Field_Declaration>
4:    [ DEFINE EQUALS ON <ParamName> USING <Field_Declaration> CODE <Code>]
5:  MEASURE <Field_Declaration>
6:    [ DEFINE THETA ON <ParamName> USING <Field_Declaration> CODE <Code>]
7:    [ FORMULA EXTENSIONAL ON <DsName> USING <Field_Declaration> CODE <Code>]
8:    [ , FORMULA INTENSIONAL <Name>]
9:
10: <Field_Declaration> ::= <Field_Name> <SQL_Type> [ ,<Field_Declaration>]
11:
12: <Delete_Pattern_Type> ::= DROP PATTERN TYPE <Pattern_Type_Name>
13:
14: <Code> ::= procedural code
15: <SQL_Type> ::= any valid SQL type
16:
17: <Pattern_Type_Name> ::= any valid identifier
18: <DsName> ::= any valid identifier
19: <Name> ::= any valid identifier
20: <Field_Name> ::= any valid identifier
21: <ParamName> ::= any valid identifier
```

Figure 8.1: $\mathcal{PSY}$-PDL commands for pattern types

In order to support the creation of new pattern types, a $\mathcal{PSY}$-PDL, named `CREATE PATTERN TYPE`, is provided. The `CREATE PATTERN TYPE` command is presented in Fig. 8.1 (lines 1-8).

This command creates a new pattern type with name `<Pattern_Type_Name>`, according to the following clauses:

- `STRUCTURE` clause. Specify the structure schema of patterns instances of `<Pattern_Type_Name>`. The structure schema is a tuple of SQL types, specified in `<Field_Declaration>`.

  - DEFINE EQUALS clause. Specify a method to be used to check equality between the structure of an instance of Pattern_Type_Name and the structure object identified by <ParamName>. <Code> is a procedural piece of code, using variables declared in the local <Field_Declaration>.

- MEASURE clause. Specify the measure schema of patterns instances of <Pattern_Type_Name>. The measure schema must be a tuple of numeric values.
  - DEFINE THETA clause. Specify a method to check a 'better than or equal to' relationship between the measure of an instance of <Pattern_Type_Name> and a measure object identified by <ParamName>. <Code> is a procedural piece of code, using variables declared in the local <Field_Declaration>.

- FORMULA EXTENSIONAL clause. Specify a method to be used to select the subset of a source dataset (approximatively) represented by a pattern instance of <Pattern_Type_Name>. <DsName> is the name of the source dataset, to be used as input for the subset computation. The generated subset is then stored in a table called <Pattern_Type_Name>FormulaE. <Code> is a procedural piece of code, using variables declared in <Field_Declaration>. No return clause is required. If FORMULA EXTENSIONAL clause is missing, a default object method returning the entire dataset associated with the pattern is created.

- FORMULA INTENSIONAL clause. Specify the name of a Prolog predicate (which is also the name of a Prolog file (.pl)). Such predicate must (approximatively) represent the subset of the source dataset represented by the pattern, using a conjunction of linear constraints. The Prolog predicate is parametric with respect to structure information, that is available in the instance.

Within the code definition (<Code>) of each clause, only variables defined in <Field_Declaration> of the same clause, the object SELF, and <DsName> (for FORMULA EXTENSIONAL) can be used.

The CREATE PATTERN TYPE statement creates some new user-defined types in the system. As a result, instances of pattern type <Pattern_Type_Name> have the following components:

- s: pattern structure, with type <Pattern_Type_Name>Structure, having the components declared in the CREATE PATTERN TYPE statement.

- d: pattern data source, a string representing an existing table/view.

- m: pattern measure, with type <Pattern_Type_Name>Measure, having the components declared in the CREATE PATTERN TYPE statement.

- `formulaI`: pattern formula, a string representing the Prolog predicate specified in the pattern type `<Pattern_Type_Name>` creation, instantiated with information taken from the pattern structure.
- `v`: pattern validity period, with type `ValPeriod(min: Date, max: Date)`.

Besides the command for creating a new pattern type, a $\mathcal{PSY}$-PDL command for dropping an existing pattern type is provided. It is called `DROP PATTERN TYPE` and it drops an existing pattern type `<Pattern_Type_Name>` from the system. Its syntax is presented in Fig. 8.1(lines 12).

8.1.1.2 Mining and Measure Functions Support

According to that previously discussed in Chapter 4 a mining function implements an extraction algorithm, whereas a measure function implements a procedure to evaluate the quality measures of a pattern with respect to a certain data source. $PSYCHO$ supports both mining and measure functions creation and deletion.

Concerning mining functions, within $PSYCHO$ they can be created in two ways:

- by directly creating a system function, which must take as input the name of a table/view to be used as data source and must store the extracted patterns in table `<Pattern_Type_Name>Tab`;
- by using the `CREATE MINING FUNCTION` statement, which creates a mining function with name `<Function_Name>`, to extract patterns of type `<Pattern_Type_Name>`, according to the following clauses:
  - `USING` clause. Specify the string `<Data_Source_Variable_Name>` to be used as input parameter of the mining function. It represents the name of the table or view from which patterns have to be extracted.
  - `WITH` clause. Specify the formal parameters for the function under definition.
  - `AS` clause. Specify declarations for local variables to be used in `Mining_Code`.

The syntax of the `CREATE MINING FUNCTION` statement is presented in Fig. 8.2 (lines 1-9). However, we point out that the implementation of this function will be available in the next $PSYCHO$ version.

```
 1:  <Create_Mining_Function> ::=
 2:    CREATE MINING FUNCTION <Function_Name>
 3:    FOR <Pattern_Type_Name>
 4:    USING <Data_Source_Variable_Name>
 5:    [WITH <Field_Declaration>]
 6:    AS <Field_Declaration>
 7:    BEGIN
 8:      <Mining_Code>
 9:    END;
10:
11: <Create_Measure_Function> ::=
12:   CREATE MEASURE FUNCTION <Function_Name>
13:   FOR <Pattern_Type_Name> <Pattern_Name>
14:   USING <Data_Source_Variable_Name>
15:   AS <Field_Declaration>
16:   BEGIN
17:     <Code>
18:   RETURN MEASURE( <List_of_Arguments> )
19:   END;
20:
21: <Drop_Mining_Function> ::= DROP MINING FUNCTION <Function_Name>
22:
23: <Drop_Measure_Function> ::= DROP MEASURE FUNCTION <Function_Name>
24:
25: <Field_Declaration> ::= <Field_Name> <SQL_Type> [ ,<Field_Declaration>]
26: <Mining_Code> ::= { <Code> | <Create_Pattern>; } [ <Mining_Code> ]
27: <Code> ::= procedural code
28: <SQL_Type> ::= any valid SQL type
29: <Function_Name> ::= any valid identifier
30: <Pattern_Type_Name> ::= any valid identifier
31: <List_of_Arguments> ::= <Argument> [ , <List_of_Values> ]
32: <Argument> ::= any valid expression of the right type
33: <Data_Source_Variable_Name> ::= any valid identifier
34: <Name> ::= any valid identifier
```

Figure 8.2: $\mathcal{PSY}$-PDL commands for mining and measure functions

On the other side, the `DROP MINING FUNCTION` command drops the mining function named `<Function_Name>` from the system. Its syntax is presented in Fig. 8.2 (line 21).

As already stated, *PSYCHO* supports also measure functions creation and deletion. In particular, measure functions can be created in two ways:

- by directly creating an Oracle function. Such function must take as input the name of a table/view to be used as data source and must return a list of measures, instances of the measure types specified during the creation of `<Pattern_Type_Name>Measure` type. Such function should compute measures starting from the input dataset and

the structure of the input pattern;

- by using the `CREATE MEASURE FUNCTION` statement, as described below.

The syntax of the `CREATE MEASURE FUNCTION` command is presented in Fig. 8.2 (lines 11-19), it creates a measure function with name `<Function_Name>`, to compute measures associated with patterns of type `<Pattern_Type_Name>`, according to the following clauses:

- `<Pattern_Name>` is the formal parameter representing the pattern of type `<Pattern_Type_Name>`, to be used as input parameter by the measure function.
- `USING` clause. Specify the variable name `<Data_Source_Variable_Name>`, used as input parameter by the measures function. It represents the name of the table or view from which pattern measures have to be computed.
- `AS` clause. Specify declarations for local variables to be used in `<Code>`.
- `RETURN` clause. Specify the list of measures to be returned. Each measure can be any Oracle expression of type compatible with the corresponding `<Pattern_Type_Name> Measure` component type.

Finally, the `DROP MEASURE FUNCTION` command drops the measure function named `<Function_Name>` from the system. Its syntax is presented in Fig. 8.2 (line 23).

8.1.1.3 Class Operations

According to that previously discussed in Chapter 4, patterns are collected into pattern classes. Patterns not belonging to any class are not querieable. Therefore, $\mathcal{PSY}$-PDL provides support for pattern class creation and deletion.

```
1:  <Create_Class> ::= CREATE CLASS <Class_Name> OF <Pattern_Type_Name>
2:
3:  <Delete_Class> ::= DROP CLASS <Class_Name>
4:
5:  <Class_Name> ::= any valid identifier
6:  <Pattern_Type_Name> ::= any valid identifier
```

Figure 8.3: $\mathcal{PSY}$-PDL commands for classes

The `CREATE CLASS` command creates a class with name `<Class_Name>` for patterns of type `<Pattern_Type_Name>`. Its syntax is presented in Fig. 8.3 (line 1).

The DROP CLASS command removes the class named <Class_Name> from the system. Its syntax is presented in Fig. 8.3 (line 3).

```
1:  <Create_Condition> ::=
2:  CREATE CONDITION <Condition_Name>
3:  FOR <Pattern_Type_Name> <Pattern_Name> AS <Field_Declaration>
4:    BEGIN
5:      <Code>
6:      RETURN <Return_value>;
7:    END
8:
9:  <Create_Join_Condition> ::=
10: CREATE JOIN CONDITION <Condition_Name>
11: FOR <Pattern_Type_Name> <Pattern_Name> JOIN
12:   <Pattern_Type_Name> <Pattern_Name> AS <Field_Declaration>
13: BEGIN
14:   <Code>
15:   RETURN <Return_value>;
16: END
17:
18: <Create_Composition_Function> ::=
19: CREATE COMPOSITION FUNCTION <Composition_Function_Name> FOR
20:   { EXISTING PATTERN TYPE <Pattern_Type_Name> <Pattern_Name> |
21:   NEW PATTERN TYPE <Pattern_Type_Name> <Pattern_Name> (
22:   STRUCTURE ( <Field_Declaration>
23:   [ DEFINE EQUALS ON <ParamName> USING <Field_Declaration> CODE <Code> ] ),
24:   MEASURE ( <Field_Declaration>
25:   [ DEFINE THETA ON <ParamName> USING <Field_Declaration> CODE <Code> ] ),
26:   FORMULA ( <Field_Declaration> CODE <Code> RETURN <Name> ) ) }
27:   OF <Pattern_Type_Name> <Pattern_Name>
28:   AND <Pattern_Type_Name> <Pattern_Name> AS <Field_Declaration>
29:   BEGIN
30:     <Code>
31:   END
32:
33: <Field_Declaration> ::= <Field_Name> <SQL_Type> [ ,<Field_Declaration> ]
34: <Code> ::= procedural code
35: <Return_value> ::=  1 | 0
36: <Condition_Name> ::= any valid identifier
37: <Pattern_Type_Name> ::= any valid identifier
38: <Pattern_Name> ::= any valid identifier
```

Figure 8.4: $\mathcal{PSY}$-PDL commands supporting querying conditions and join composition function

8.1.1.4 CREATE CONDITION

The `CREATE CONDITION` command creates an Oracle function `<Condition_Name>`, having a single parameter representing an instance of type `<Pattern_Type_Name>` and checking a given condition over such instance. It can be specified according to the following clauses:

- `<Pattern_Name>` is the name of the variable containing the pattern (of type `<Pattern_Type_Name>`), representing the function formal parameter.
- `AS` clause. Specify declarations for local variables to be used in `Code`.
- `RETURN` clause. The function must return either 1 (for true) or 0 (for false).

The syntax of the `CREATE CONDITION` statement is presented in Fig 8.4 (line 1-7).

8.1.1.5 CREATE JOIN CONDITION

The `CREATE JOIN CONDITION` command creates an Oracle function `<Condition_Name>`, having two parameters, the first representing a pattern instance of the first specified pattern type, the second representing an instance of the second pattern type, and checking a given condition over such instances. It can be specified according to the following clauses:

- `<Pattern_Name>` is the name of the variable containing the pattern (of type `<Pattern_Type_Name>`), representing one formal parameter.
- `AS` clause. Specify declarations for local variables to be used in `Code`.
- `RETURN` clause. The function must return either 1 (for true) or 0 (for false).

The syntax of the `CREATE JOIN CONDITION` command is presented in Fig 8.4 (lines 9-16).

8.1.1.6 CREATE COMPOSITION FUNCTION

The `CREATE COMPOSITION FUNCTION` command creates an Oracle function `<Composition_Function_Name>`, having two parameters, the first representing a pattern instance of the first specified pattern type, the second representing an instance of the second pattern type. Such function returns a new pattern obtained by combining the input ones. The returned pattern must be an instance of either an existing pattern type or of a new pattern type, whose structure can be declared inside the `CREATE COMPOSITION FUNCTION` statement. A composition function can be specified according to the following clauses:

- `EXISTING PATTERN TYPE` clause. Specify the name `<Pattern_Type_Name>` of an existing pattern type, to be used as type of the result. `<Pattern_Name>` is the name of the variable to be used for the output pattern.
- `NEW PATTERN TYPE` clause. Specify a new pattern type, according to the syntax of the `CREATE PATTERN TYPE` statement.
- `OF` clause. Specify the types of the input patterns and the name of the variables to be used for the input patterns.
- `AS` clause. Specify declarations for local variables to be used in `<Code>`.
- `RETURN` clause. The function must return either 1 (for true) or 0 (for false).

The syntax of the `CREATE COMPOSITION FUNCTION` command is presented in Fig 8.4 (lines 18-31).

8.1.2 $\mathcal{PSY}$-PML: $\mathcal{PSYCHO}$ Pattern Manipulation Language

The *PSYCHO* Pattern Manipulation Language ($\mathcal{PSY}$-PML) implements the manipulation language, $\mathcal{E}$PML, introduced in Section 5.3. It allows one to extract or directly insert, delete, recompute over new data sources, and update patterns belonging to the pattern layer. The patterns to be deleted, recomputed, or updated are always selected with a condition to be evaluated over the pattern layer. In the following, each $\mathcal{PSY}$-PML operation is described in details.

```
<Extraction> ::=
  EXTRACT PATTERNS OF <Pattern_Type_Name> <Alias>
  FROM <Data_Source_Name>
  USING <Mining_Function_Name>([<List_of_Values>])
  WITH (<List_of_Values>)
  [ VALID FROM <ts> TO <ts> ]
  [ INTO CLASS <Class_name> ]

<Direct_Insertion> ::=
  DIRECT INSERT PATTERN OF <Pattern_Type_Name>
  FROM <Data_Source_Name>
  STRUCTURE (<List_of_Values>)
  [ MEASURE (<List_of_Values>) ]
  [ VALID FROM <ts> TO <ts> ]
  [ INTO CLASS <class_name> ]

<Recomputation> ::=
  RECOMPUTE PATTERNS OF <Pattern_Type_Name> <Alias>
  ON <Data_Source_Name>
  USING <Measure_Function_Name>
  [ WHERE <Condition> ]
  [ VALID FROM <ts> TO <ts> ]
  [ INTO CLASS <Class_name> ]

<Update> ::= UPDATE PATTERNS OF <Pattern_Type_Name> <Alias>
  { <Set_Validity> | <Synchronize> | <Validate> }

<Set_Validity> ::= SET VALIDITY FROM <ts> TO <ts>;

<Synchronize> ::= SYNCHRONIZE USING <Measure_Function_Name>
  [ WHERE <Condition> ] ;

<Validate> ::= VALIDATE USING <Measure_Function_Name>
  [ WHERE <Condition> ]
  [ INTO CLASS <Class_name> ];

<Deletion> ::= DELETE PATTERNS OF <Pattern_Type_Name> <Alias>
  [ WHERE <Condition> ];

<Pattern_Type_Name> ::= any valid identifier
<Data_Source_Name> ::= any valid identifier
<Mining_Function_Name> ::= any valid identifier
<Measure_Function_Name> ::= any valid identifier
<Class_name> ::= any valid identifier
<Alias> ::= any valid identifier
<List_of_Values> ::= <Value> [ , <List_of_Values> ]
<Value> ::= a value
<ts> ::= a date value
```

Figure 8.5: $\mathcal{PSY}$-PML commands supporting pattern insertion and update

8.1.2.1 EXTRACTION

The execution of the EXTRACTION command extracts patterns of type <Pattern_Type_Name> from the existing table/view named <Data_Source_ Name>, using the existing mining function <Mining_Function_Name>. Such patterns are inserted in the pattern layer. Extraction can be specified according to the following clauses:

- USING clause. Specify an existing mining function call (thus, actual parameters have also to be specified).
- WITH clause. Specify a value for each measure of patterns of type <Pattern_Type_Name>. Each value must be an instance of the type declared in the pattern type. Only patterns having measures that are better than or equal to those specified are extracted. For the comparison, method THETA defined during pattern type creation is used.
- VALID FROM clause. Specify the validity period to be assigned to the extracted patterns.
- INTO CLASS clause. Specify the name of an existing class, in which the extracted patterns have to be inserted.

The syntax of the EXTRACTION command is presented in Fig. 8.5 (lines 1-7).

8.1.2.2 DIRECT INSERTION

The DIRECTION INSERTION command directly inserts a pattern of type <Pattern_Type_Name> into the pattern layer, according to the following clauses:

- FROM clause. Specify the table/view name, to be used as pattern source dataset.
- STRUCTURE clause. It specifies a value for each pattern structure component. Each value must be an instance of the type declared in the pattern type.
- MEASURE clause. Specify a value for each measure of patterns of type <Pattern_Type_Name>. Each value must be an instance of the type declared in the pattern type.
- VALID FROM clause. Specify the validity period to be assigned to the inserted pattern.
- INTO CLASS clause. Specify the name of an existing class, in which the pattern has to be inserted.

The syntax of the DIRECTION INSERTION command is presented in Fig. 8.5 (lines 9-15).

8.1.2.3 RECOMPUTATION

The `RECOMPUTATION` command recomputes the measures of a subset of patterns instances of `<Pattern_Type_Name>` and generates new patterns with the same structure, a new dataset and a new validity period, taken as input. The measures of the new patterns are recomputed over the new dataset. The recomputed patterns can be inserted into a class specified in input. Recomputation can be specified according to the following clauses:

- `ON` clause. Specify the table/view name, to be used as the source dataset for the recomputed patterns.
- `USING` clause. Specify the name of the existing measure function to be used for the computation of the new measures over the specified source dataset.
- `WHERE` clause. Specify the condition selecting patterns, instances of `<Pattern_Type_Name>`, whose measures have to be recomputed. For `<Condition>`, see Section 8.1.3.
- `VALID FROM` clause. Specify the validity period to be assigned to the recomputed patterns.
- `INTO CLASS` clause. Specify the name of an existing class, in which the recomputed patterns have to be inserted.

The syntax of the `RECOMPUTATION` command is presented in Fig. 8.5 (lines 17-23).

8.1.2.4 UPDATE PATTERNS

The `UPDATE PATTERNS` command updates a subset of patterns instances of `<Pattern_Type_Name>` in three distinct ways:

- by updating their validity period;
- by synchronizing their measures with the actual data source;
- by validating the pattern, possibly recomputing new patterns when the quality of the representation achieved by the pattern decreases (thus, the recomputed measures are worst than the existing ones).

Update can be specified according to the following clauses:

- `SET VALIDITY` clause. Specify the new validity period to be assigned to the updated patterns.
- `SYNCHRONIZE` clause. Specify the name of the existing measure function to be used for the update of the measures of the selected patterns. The current data source, associated with the selected patterns, is used for the computation.
- `VALIDATE` clause. Specify the name of the existing measure function to be used for the validation of the updated patterns. It first recomputes the measure values associated with selected temporally valid patterns (which are patterns whose validity period has not expired yet) using `<Measure_Function_Name>`. If such measure values are better than or equal to the ones associated with patterns before validation (this can be checked by using method `THETA`), patterns are semantically valid. Thus, similarly to synchronization, measures are modified, and the validity period is left unchanged. On the other hand, if measures are worst than before, the validity period of the pattern is changed, setting the end time to `Current_time` and a new pattern is created, with the same structure and dataset than the previous one, but with the new measures. Its validity period is set to `[Current_time,` $+\infty$`)`.
- `WHERE` clause. Specify the condition that identifies patterns, instances of `<Pattern_Type_Name>`, that must be updated For `<Condition>`, see Section 8.1.3.
- `INTO CLASS` clause. Specify the name of an existing class, in which validated patterns have to be inserted. Such clause cannot be specified when updating the validity period or when synchronizing patterns, since no new patterns are generated in those cases.

The syntax of the `UPDATE PATTERNS` command is presented in Fig. 8.5 (lines 25-35).

8.1.2.5 DELETE PATTERNS

The `DELETE PATTERNS` command removes from the pattern layer all the patterns, instances of
`<Pattern_Type_Name>`, satisfying condition `<Condition>`, that are contained in no class. If the `WHERE` clause is not specified, all patterns instances of `<Pattern_Type_Name>` are removed from the pattern layer. For `<Condition>`, see Section 8.1.3.

The syntax of the `DELETE PATTERNS` command is presented in Fig. 8.5 (lines 37-38).

8.1.2.6 Manipulation Operations for Classes

```
1:  <Insert_Into_Class> ::= INSERT INTO CLASS <Class_Name> <Alias>
2:      [WHERE <Condition>];
3:
4:  <Delete_From_Class> ::= DELETE FROM CLASS <Class_Name> <Alias>
5:      [WHERE <Condition>];
6:
7:  <Class_name> ::= any valid identifier
8:  <Alias> ::= any valid identifier
```

Figure 8.6: $\mathcal{PSY}$-PML commands supporting pattern class management

8.1.2.7 INSERT INTO CLASS

This command inserts into the existing class `<Class_Name>` the already existing patterns, instances of the pattern type over which the class has been defined, satisfying condition `<Condition>`. If the `WHERE` clause is not specified, all patterns instances of the pattern type over which the class has been defined are inserted into the class. For `<Condition>`, see Section 8.1.3.

The syntax of the `INSERT INTO CLASS` command is presented in Fig. 8.6 (lines 1-2)

8.1.2.8 DELETE FROM CLASS

This command deletes from the existing class `<Class_Name>` patterns that are instances of the pattern type over which the class has been defined, satisfying condition `<Condition>`. Patterns are not removed from the pattern layer. If the `WHERE` clause is not specified, all patterns instances of the pattern type over which the class has been defined are removed from the class. For `<Condition>`, see Section 8.1.3.

The syntax of the `DELETE FROM CLASS` command is presented in Fig. 8.6 (lines 4-5)

8.1.3 $\mathcal{PSY}$-PQL: $\mathcal{PSYCHO}$ Pattern Query Language

The *PSYCHO* Pattern Query Language ($\mathcal{PSY}$-PQL) implements the query language, $\mathcal{E}PQL$, introduced in Section 5.2. It allows one to retrieve patterns according to specified conditions. Moreover, it allows one to combine patterns together through join operation and to combine patterns with raw data (cross-over queries). We remark that, all query operations are executed over pattern classes. In the following, each operation is described in details.

8.1.3.1 SELECT

The SELECT command retrieves patterns according to some specified condition. All the pattern information are returned to the user (thus, no projection is provided). Its syntax is shown in Fig. 8.7.

```
<Select> ::=
    SELECT *
    FROM { <Single_class> | <Join> }
    [ WHERE { <Condition> |
    <Condition> { I_AND|I_OR } <Intensional_condition> } ] ;

<Single_class> ::= {<ClassName> | ( <Select> )} <Alias>

<Join> ::= { <Single_class> INTERSECT JOIN <Class_Name> <Alias> |
    <Single_class> CJOIN <Class_Name> <Alias>
    WITH <Composition_Function_Name> }

<Condition> ::= [ NOT ] <Bool_Expression> [ { AND | OR } <Condition> ]

<Bool_Expression> ::= { <Argument> <Operator> <Argument> |
    <Condition_Name>(<Alias>) = 1 |
    <Predicate> ( <Argument> , <Argument> ) = 1 }

<Predicate> ::= { ISTEQUALS | ISTEQUALSD | PRECEDES | PRECEDESD |
    FOLLOWS | FOLLOWSD | MEETS | OVERLAP | DURING | STARTS | FINISHES |
    EEQUIV | ECONTAIN | IDENTITY | SEQUALS | ISTVALID |
    WSUBSUMES | ISBETTEROF | ISSVALID | INCLASS }

<Intensional_condition> ::= <Intensional_predicate> ( <Argument> ,
{ <Argument> | <Value> })
<Intensional_predicate> ::= { IEQUIV | ICONTAIN }
<Argument> ::= any expression of the right type
<Operator> ::= any predicate of the right type
<Condition_Name> ::= any valid identifier
<Class_name> ::= any valid identifier
<Alias> ::= any valid identifier
<Value> ::= any value of the right type
```

Figure 8.7: $\mathcal{PSY}$-PQL commands supporting pattern querying

Two types of queries are supported: simple queries and joins. Selection can be specified according to the following clauses:

- FROM clause. Specify the set of patterns to which selection has to be applied. There are two different modalities:

- `<Single_Class>` clause. Specify the name of a class or a select query from which patterns have to be selected (thus nesting is supported).
- `<Join>` clause. Specify a join operation. Two join operations are supported:
  - `INTERSECTION JOIN`. It takes two classes (only the first can be nested) and, for each pair of patterns `p1` and `p2`, the first belonging to the first class, the second belonging to the second one, generates a new pattern `p` defined as follows: `p.s = (p1.s,p2.s)`, `p.d = p1.d INTERSECT p2.d`, `p.m = (p1.m,p2.m)`, `p.v = (max(p1.v.min,p2.v.min)`, `min(p1.v.max, p2.v.max)`, `p.formulaI = p1.formulaI AND p2.formulaI`. The further selection is applied to the resulting patterns.
  - `CJOIN`. It takes two classes (only the first can be nested) and, for each pair of patterns, the first belonging to the first class, the second belonging to the second one, it applies the composition function `<Composition_Function_Name>`. The further selection is applied to the resulting patterns.

- `WHERE` clause. Specify the selection condition that returned patterns must satisfy. More precisely:
  - `<Condition>`: logical combination of boolean expressions. Each boolean expression may take three different forms:
    - `<Argument> <Operator> <Argument>`: Oracle boolean expression (possibly using the alias);
    - `<Condition_Name>(<Name>) = 1`: calls to $PSYCHO$ user-defined conditions.
      `<Condition_Name>` must be the name of an existing $PSYCHO$ condition, created with the command `CREATE CONDITION`;
    - `<Predicate> ( <Argument> , <Argument> ) = 1`: calls to the following predefined $PSYCHO$ binary boolean predicates:
      - `ISTEQUAL`: temporal predicate, checking whether two validity periods are equal.
      - `ISTEQUALD`: temporal predicate, checking whether a validity period (first argument) is equal to a given date (second argument).
      - `PRECEDES`: temporal predicate, checking whether the first validity period ends before the second one.
      - `PRECEDESD`: temporal predicate, checking whether a validity period (first argument) precedes a given date (second argument).
      - `FOLLOWS`: temporal predicate, checking whether the first validity period starts after the second one.

- FOLLOWSD: temporal predicate, checking whether a validity period (first argument) starts after a given date (second argument).
- MEETS: temporal predicate, checking whether the first validity period ends when the second validity period starts.
- OVERLAP: temporal predicate, checking whether two validity periods intersects.
- DURING: temporal predicate, checking whether the first validity period is contained in the second one.
- STARTS: temporal predicate, checking whether two validity periods start at the same date and the first is contained in the second.
- FINISHES: temporal predicate, checking whether two validity periods ends at the same date and the first is contained in the second.
- EEQUIV: predicate over data sources, checking whether two data sources coincide.
- ECONTAIN: predicate over data sources, checking whether the first data source contains the second one.
- IDENTITY: predicate over patterns, checking whether two patterns have the same PID.
- SEQUALS: predicate over patterns, checking whether two patterns are shallow equal, i.e., all components except PID and validity period are equal.
- WSUBSUMES: predicate over patterns, checking whether the first pattern represents a larger set of data than the second. Patterns may have different types.
- ISTVALID: it takes a pattern and a date and determines whether the patter is temporally valid at that date (i.e., the input date is contained in the pattern validity period).
- ISSVALID: it takes a pattern, a data source name, and a measure function name. It determines whether the pattern is semantically valid with respect to the input data source, i.e., whether the measures computed over the input data source using the input measure function are better than or equal to the current existing measures.
- INCLASS: it takes a pattern and a class and determines whether that pattern is contained in that class.

– {I_AND|I_OR} <Intensional_condition>: <Condition> can be put in AND (IAND) or in OR (IOR) with an intensional condition <Intensional_condition>. Such condition applies an intensional predicate <Intensional_predicate> to the pattern formula, thus it exploits the declarative semantics of patterns (through the Prolog-defined intensional formula) to check pattern properties. For each

WHERE clause, only one intensional condition is allowed. Two distinct intensional predicates are provided (data sources are not accessed for their evaluation):

* IEQUIV: predicates over two intensional formulas, checking whether the two formulas are equivalent.
* ICONTAIN: predicates over two intensional formulas, checking whether the first formula contains the second one, i.e., whatever is the considered data source, all the solutions of the first formula are contained in the set of solutions of the second one.

8.1.3.2 DRILL THROUGH

The DRILL THROUGH command is a cross-over operation that allows one to navigate from patterns to their source data. In particular, it returns the union of the data sources of the patterns selected. Its syntax is presented in Fig. 8.8 (lines 1-2).

```
1:  <Drill_Through> ::= DRILL THROUGH <ClassName> | ( <Select> )  <Alias>
2:      [ WHERE <Condition> ] ;
3:
4:  <Data_Covering> ::= DATA COVERING <ClassName> | ( <Select> )  <Alias>
5:      FOR <Data_Source_Name>
6:      [ WHERE <Condition> ] ;
7:
8:  <Pattern_Covering> ::= PATTERN COVERING <Data_Source_Name>
9:      FOR <ClassName> | ( <Select> )  <Alias>
10:     [ WHERE <Condition> ] ;
11:
12: <Class_name> ::= any valid identifier
13: <Alias> ::= any valid identifier
```

Figure 8.8: $\mathcal{PSY}$-PML commands supporting cross-over querying

8.1.3.3 DATA COVERING

The DATA COVERING command is a cross-over operation that allows one to navigate from patterns to data, approximatively represented by the patterns. To this purpose: (i) it first selects a set of patterns; (ii) it applies their extensional formula to an input table/view data source with name <Data_Source_Name>; (iii) it finally returns the union of the results. The syntax of the DATA COVERING command is presented in Fig. 8.8 (lines 4-6).

8.1.3.4 PATTERN COVERING

The `PATTERN COVERING` command is a cross-over operation that allows one to navigate from data to patterns, approximatively representing those data. To this purpose: (i) it first selects a set of patterns; (ii) it applies the extensional formula to each of the selected patterns, using the table/view with name `<Data_Source_Name>` as input data source; (iii) it finally returns the patterns such that the dataset generated by the extensional formula contains the input one. The syntax of the `PATTERN COVERING` command is presented in Fig. 8.8 (lines 8-10).

8.2 $\mathcal{PSYCHO}$ Application Scenarios

In this section we present several scenarios describing different applications of $PSYCHO$. For each scenario, we point out the main goal and several operations, written in $PSYCHO$ languages, aimed at showing the potentials of the system.

8.2.1 Scenario 1

The aim of the first scenario is to show representation, generation, manipulation and querying for patterns of the same type in $PSYCHO$. To this purpose, we consider two well-known data mining pattern types: association rules and clusters. Such pattern types are managed by any commercial systems dealing with data mining. However, $PSYCHO$ allows to perform several operations that are not directly supported by other existing tools.

8.2.1.1 Association rules

The reference domain for this scenario is Market Basket Analysis (see Chapter 2), thus we deal with patterns of type `AssociationRules`.

Set-up. In order to represent and manage a given pattern type in $PSYCHO$, the first step consists in declaring all schema objects required to represent and manage instances of that pattern type. In this phase, we define the pattern type, the mining function required to extract association rules from source data, and the measure function, required to recompute measures upon a given data source.

For association rules, we may assume such objects are already existing in the system, since the association rule type is a standard data mining pattern. In the following, we however show how these objects can be defined.

Data Source. We assume source data is stored in a table with schema ($DSid$, $Item_1, ..., Item_n$), where each tuple represents a sale transaction identified by $DSid$, $Item_i$ is either 1 or 0, and $Item_i = 1$ means that the corresponding transaction contains $Item_i$.

Pattern Type. Since association rules are standard data mining patterns, we assume a pattern type, named `AssociationRule`, is already available in the system, using the standard representation of an association rule, presented above.

Besides head, body, support, and confidence, in $PSYCHO$ each pattern type is associated with three additional components: the extensional formula, the intensional formula, and the validity period. The extensional formula is just a PL/SQL function that takes a source dataset and returns the subset of such dataset (possibly approximatively) represented by the pattern. The intensional formula has the same meaning, but it is intensionally represented through a Prolog predicate, defined by a set of linear constraints. The validity period is just a temporal interval inside which we assume the information represented by the pattern is reliable.

In order to define the pattern type for Association Rule, a basic type to model an array of strings and a function to test whether two such arrays are equal are required. For that, the user can exploit the object-relational Oracle features (see Fig. 8.9).

```
1:  CREATE TYPE CharArray AS varray[100] OF varchar2(25);
2:
3:  CREATE OR REPLACE FUNCTION CHARARRAY_EQUAL ( a CharArray, b CharArray)
4:  RETURN INTEGER AUTHID CURRENT_USER IS
5:      i int; j int; cond int;
6:  BEGIN
7:      cond := 0; i := a.first;
8:      if a.count <> b.count then return 0; end if;
9:      while i <= a.last loop
10:     cond := 0;
11:     j := b.first;
12:     while j <= b.last AND cond=0 loop
13:     if a(i) <> b(j) then j:= j+1; else cond:= 1; end if;
14:     end loop;
15:     if cond = 0 then return 0; end if;
16:     i:= i+1;
17:     end loop;
18:     return cond;
19:     END chararray_equal;
```

Figure 8.9: Scenario 1: *AssociationRule*'s support items creation

The pattern type can be created by using the $\mathcal{PSY}$-PDL statement shown in Fig. 8.10.

```
1:  CREATE PATTERN TYPE AssociationRule
2:  STRUCTURE head CharArray, body CharArray
3:    DEFINE EQUALS ON p1 USING ret int CODE
4:      if CHARARRAY_EQUAL(self.s.head, p1.s.head)=1 AND
5:      CHARARRAY_EQUAL(self.s.body,p1.s.body)=1
6:      then ret:= 1;
7:      else ret:=0;
8:      end if;
9:      return ret;
10: MEASURE support REAL, confidence REAL
11:   DEFINE THETA ON p2 USING ret int CODE
12:     if (self.m.support >= p2.m.support AND
13:       self.m.confidence>=p2.m.confidence)
14:     then ret:= 1;
15:     else ret:=0;
16:     end if;
17:     return ret;
18: FORMULA EXTENSIONAL ON varDS
19: USING condB varchar2(100), condH varchar2(100) CODE
20:     condH:= ‘’; condB:= ‘’;
21:     if self.s IS NOT NULL
22:     then
23:     for i IN self.s.head.first ..  self.s.head.last-1 loop
24:         condH:=condH || self.s.head(i) || ‘=1 AND ’;
25:     End loop;
26:     condH:=condH|| self.s.head(self.s.head.last) || ‘=1 ’;
27:     for j IN self.s.body.first ..  self.s.body.last-1 loop
28:         condB:=condB || self.s.body(j) || ‘=1 AND ’;
29:     end loop;
30:     condB:=condB|| self.s.body(self.s.body.last) || ‘=1 ’;
31:     Execute Immediate ‘INSERT INTO AssociationRuleFormulaE
32:     (SELECT * FROM ’ || varDS || ‘ WHERE ’ || CondH || ‘ AND ’
33:     || CondB || ‘) ’;
34:     end if;
35: FORMULA INTENSIONAL ARFormula_INT;
```

Figure 8.10: Scenario 1: *AssociationRule* patter type creation

Note that:

- as default, an extensional formula returning the whole pattern data source is associated with a pattern of type `AssociationRule`; whenever the user specifies an extensional formula code this overrides the default behavior;
- concerning the intensional formula, the user has to specify the name of an existing library intensional predicate (i.e. a Prolog predicate);

- the structure is associated with a function `EQUALS`, checking the equality between two structure values for the pattern type under definition;
- the measure is associated with a function `THETA`, checking relation 'better of or equal to' between two measure values for the pattern type under definition.

Mining Function. Given a pattern type, several mining functions can be used to extract patterns of that type from a given dataset. For example, for association rules, we may assume to have a Java implementation of the Apriori algorithm [AS94]. Different mining functions may however be available. In the current version of *PSYCHO*, this can be done by defining some Oracle functions.

Measure Function. Various measure functions can also be defined, each recomputing measure values for patterns instances of a given pattern type, upon a given data source. As an example, the statement in Fig. 8.11 creates a measure function for computing confidence and support over a given data source (identified by the variable `varDS`).

```
1:  CREATE MEASURE FUNCTION AR_Measure_func
2:  FOR AssociationRule ar
3:  USING varDS AS CondB varchar2(100), CondH varchar2(100),
4:  Supp real, Conf real, CountAll real
5:  BEGIN CondH:= ‘’; CondB:= ‘’; Supp := 0; Conf := 0;
7:    For i IN ar.s.head.first ..  ar.s.head.last-1 Loop
8:      CondH:=condH || ar.s.head(i) || ‘=1 AND ’;
9:    End loop;
10:   CondH:=condH|| ar.s.head(ar.s.head.last) || ‘=1 ’;
11:   For j IN ar.s.body.first ..  ar.s.body.last-1 Loop
12:     CondB:=condB || ar.s.body(j) || ‘=1 AND ’;
13:   End loop;
14:   CondB:=condB|| ar.s.body(ar.s.body.last) || ‘=1 ’;
15:   Execute Immediate ‘BEGIN SELECT count ( * ) INTO :ret FROM ’ ||
16:       varDS || ‘; END;’
17:       using out CountAll;
18:   Execute Immediate ‘BEGIN SELECT count ( * ) INTO :ret FROM ’|| varDS ||
19:       ‘ WHERE ’|| CondH || ‘ AND ’|| CondB || ‘; END;’
20:       using out supp;
21:   Execute Immediate ‘BEGIN SELECT count ( * ) INTO :ret FROM ’|| varDS ||
22:       ‘ WHERE ’|| CondB || ‘; END;’ using out conf;
23:   Conf := supp/conf;
24:   Supp := supp/CountAll;
25: RETURN MEASURE(supp,conf);
26: END;
```

Figure 8.11: Scenario 1: measure function creation for *AssotionationRule*

Population. In this step we show how *PSYCHO* can be used to: (i) use various mining

functions to extract patterns of a given type; (ii) directly insert patterns, possibly based on some PMML model (in the current *PSYCHO* version this is possible only for association rules).

```
1:  CREATE CLASS AR30_PSYCHO OF AssociationRule;
2:
3:  EXTRACT PATTERNS OF AssociationRule ar
4:  FROM itemsDS_30
5:  USING Apriori(0.4,0.7)
6:  VALID FROM '01-jul-2005' TO '10-aug-2005'
7:  INTO CLASS AR30_PSYCHO;
8:
9:  CREATE CLASS AR30_ODM OF AssociationRule;
10:
11: EXTRACT PATTERNS OF AssociationRule ar
12: FROM ITEMSDS_30
13: USING aprioriODM(0.4,0.7)
14: VALID FROM '10-jun-2005' TO '10-aug-2005'
15: INTO CLASS AR30_ODM;
16:
17: SELECT *
18: FROM AR30_ODM or intersect join AR30_PSYCHO pr
19: WHERE chararray_equal(or.s.head,pr.s.head)=1 AND
20: chararray_equal(or.s.body,pr.s.body)=1;
21:
22: DIRECT INSERT PATTERN OF AssociationRule
23: FROM itemsDS_30
24: STRUCTURE ( chararray('BREAD','MILK'),chararray('JAM','BUTTER','WINE') )
25: MEASURE (0.5,0.7)
26: VALID FROM '01-aug-2005' TO '15-aug-2005'
27: INTO CLASS AR30_PSYCHO;
28:
29: RECOMPUTE PATTERNS of AssociationRule ar
30: ON itemsDS_5
31: USING AR_measure_func;
```

Figure 8.12: Scenario 1: insertion session of patterns of type *AssociationRule*

Pattern Extraction. We first show association rules extraction, using a *PSYCHO* Java mining function `apriori` implementing the Apriori algorithm. Before extracting patterns, it may be useful to create a class (i.e., a collection), where to store the extracted patterns (if no class is used, patterns are inserted in the pattern layer but they cannot be used in queries). A class, called `AR30_PSYCHO`, can be created using the syntax presented in Fig. 8.12 (line 9).

Now rules can be extracted by using the existing function `Apriori` over a data source

ITEMSDS_30 and the result stored in class AR30_PSYCHO. We specify that the support of extracted rules must be higher than 0.4 and the confidence higher than 0.7 (using THETA method specified in the pattern type); the validity period of the extracted rules is set from 01-jul-2005 to 10-aug-2005.

Other association rules can be extracted by using the mining function available in Oracle Data Mining (ODM) [Ora]. Assume to design a mining function Apriori_ODM calling the ODM one and suppose to store the extracted rules in a class AR30_ODM. The following statement extract rules with support higher than 0.4 and confidence higher than 0.7 from a data source ITEMSDS_30, using mining function Apriori_ODM. The validity period of the extracted rules is set from 10-jul-2005 to 10-aug-2005. If now we analyze the context of the two generated classes we discover that the two mining functions extract different association rules. To see this, we can execute the following query.

Direct Insertion. Single association rules can also be directly inserted in *PSYCHO* by specifying each component. For example, the following statement inserts in class AR30_PSYCHO a rule having items BREAD and MILK in its *head* and items JAM, BUTTER, and WINE in its *body*, representing data in itemsDS_5 dataset with support equals to 0.5 and confidence equals to 0.7, and valid from August, 1 to August, 15 2005.

Recomputation. New patterns can also be generated by recomputing measures of existing patterns over a new data source. Here is an example, recomputing all association rules over dataset itemsDS_5.

Class Management. In order to use patterns for queries or other manipulation operations, patterns have to be stored in classes. A class is just a set of semantically related patterns. Such patterns may however be extracted or inserted in different steps. Each class contains instances of a certain pattern type; however, a certain pattern may belong to different classes. In this way, using classes, it is possible to deal with different semantic aspects of patterns.

Suppose we want to define a class containing all association rules having a support greater than 0.5%. Such class, named AR_high_support, can be created with the statement presented in Fig. 8.13 (line 1). Now, association rules can be inserted in the just created class by using the statement in Fig. 8.13 (line 3).

```
1:  CREATE CLASS AR_high_support OF AssociationRule;
2:
3:  INSERT INTO CLASS AR_high_support ar WHERE ar.m.support > = 0.5;
4:
5:  CREATE CLASS AR_about_bread OF AssociationRule;
6:
7:  CREATE CONDITION bread_in_Head FOR AssociationRule ar AS
8:  BEGIN
9:  for i in ar.s.head.first ..  ar.s.head.last loop
10: if (ar.s.head(i) = 'BREAD') then return 1;
11: else return 0;
12: end if;
13: end loop;
14: END;
15:
16: INSERT INTO CLASS AR_about_bread ar WHERE bread_in_Head(ar)=1;
```

Figure 8.13: Scenario 1: *AssociationRule* class management

As an additional example, suppose we want to define a class containing all association rules - already extracted, inserted, or imported (and, therefore, available) in $PSYCHO$ - having the element 'bread' in their head. Such class, named `AR_about_bread`, can be created with the statement presented in Fig. 8.13 (line 5). Now, association rules can be inserted in the just created class by using the statements in Fig. 8.13 (lines 7-14). Note that, after the `WHERE` clause, we inserted the name of a condition `bread_in_Head`. A condition is no more than a function, used to define a `WHERE` condition for the sake of simplicity when SQL cannot be directly used and PL-SQL is required.

Queries. $\mathcal{PSY}$-PQL is very expressive and it provides the capability to exploit all pattern logical model characteristics. In details, it supports the following querying features: (i) simple queries involving predicates dealing with pattern components; (ii) pattern composition; (iii) nested queries; (iv) pattern-data reasoning (cross-over queries).

In the following, we present examples for each class of queries.

Simple Queries. Simple queries allow one to select patterns from a given class, according to a variety of predicates. In the `WHERE` clause of a `SELECT` statements we can directly insert the condition, according to an SQL-like syntax, or use a predefined condition, e.g. a function, possibly written in PL/SQL (see above).

In the following we describe several examples of possible queries (see Fig. 8.14 for the correspondent codes):

Q_1) Retrieve all association rules from `AR30_PSYCHO` with support smaller than or equal to 0.75.

Q_2) Retrieve all association rules from `AR30_PSYCHO` that are valid on July, 20 2005.

Q_3) Retrieve all association rules from `AR30_PSYCHO` that are valid during the period August,1 2005 - August, 10 2005.

Q_4) Retrieve all association rules from `AR30_ODM` having validity period equal to June,10 2005 - August, 10 2005.

Q_5) Retrieve all association rules from `AR30_PSYCHO` valid before the period September,1 2005 - September, 10 2005.

Q_6 Retrieve all association rules from `AR30_PSYCHO` having at least 1 items in their head and 3 items in their body.

Q_7) Retrieve all association rules from `AR30_PSYCHO` having a confidence value greater or equal to 0.75 or which are temporally valid on August,15 2005.

```
Q1: SELECT *
      FROM AR30_PSYCHO ar
      WHERE ar.m.confidence >= 0.75;

Q2: SELECT *
      FROM AR30_PSYCHO ar
      WHERE isTvalid(ar,'20-jul-2005')=1;

Q3: SELECT *
      FROM AR30_PSYCHO ar
      WHERE during(ar.v, valPeriod('01-aug-2005','10-aug-2005') )=1;

Q4: SELECT *
      FROM AR30_ODM ar
      WHERE isTEqual(ar.v,valPeriod('10-jun-2005','10-aug-2005'))=1;

Q5: SELECT *
      FROM AR30_PSYCHO ar
      WHERE precedes(ar.v,valPeriod('1-sep-2005','10-sep-2005'))=1;

Q6: SELECT *
      FROM AR30_PSYCHO ar
      WHERE ar.s.Head.count >= 1 AND ar.s.Body.count >= 3;

Q7: SELECT *
      FROM AR30_PSYCHO ar
      WHERE ar.m.confidence >= 0.75 OR isTvalid(ar,'15-aug-2005')=1;

Q8: SELECT *
      FROM AR30_PSYCHO ar1 CJOIN AR30_PSYCHO ar2 WITH Trans_closure_ar
      WHERE ar2.s.body.count >= 2;

Q9: SELECT *
      FROM AR30_PSYCHO ar1 INTERSECT JOIN AR30_PSYCHO ar2
```

Figure 8.14: Scenario 1: simple and join query examples

Pattern Composition. Within *PSYCHO* different types of join are available. In the actual release, two types of join are provided: a general one (`CJOIN`) and a specific one (`INTERSECTION JOIN`). The `CJOIN` takes two classes and, for each pair of patterns, the first belonging to the first class, the second belonging to the second one, it applies a specified composition function, specifying the structure of the resulting patterns. On the other hand, the `INTERSECTION JOIN` takes two classes and returns new patterns, whose structure is a combination of the input structures and whose intensional formula is the conjunction of input intensional formulas.

In the following we describe several examples of possible queries (see Fig. 8.14 for the

correspondent codes):

Q_8) Determine all association rules $H_1, ..., H_n \leftarrow B_1, ..., B_m$, obtained as the transitive closure of two existing association rules, and having at least two items in the body. In order to represent this query, we can use a composition join, by defining an ad-hoc composition function `Trans_closure_ar`.

Q_9) Determine all pairs of valid association rules, generating their intersection.

Nested Queries. $\mathcal{PSY}$-PQL queries can also be nested. In the current version, nesting is provided in the `FROM` clause. Here are some examples of queries (see the correspondent code in Fig. 8.15).

Q_{10}) Select among rules in the class `AR30_PSYCHO` with at least confidence value equal to 0.7 the ones which are temporally valid in 15-aug-2005.

Q_{11}) Select among rules in the class `AR30_PSYCHO` with at least confidence value equal to 0.7 the ones with validity period that meets the time interval 01-aug-2005 - 15-aug-2005.

Cross-Over Queries. $\mathcal{PSY}$-PQL supports pattern-data reasoning, i.e. it allows the user to specify queries involving both data and patterns. Such kind of queries are quite important in pattern management, since they allow the user to discover interesting (possibly new) correlations between patterns and data.

In the following we describe several examples of possible queries (see Fig. 8.15 for the correspondent codes):

Q_{12}) Determine all data represented by association rules in class `AR30_ODM`. This statement returns the union of the datasets of the input patterns.

Q_{13}) Which data are represented by a certain association rule, identified by PID= 1001348, in class `AR30_ODM`? In this query, there is the need to combine together the `DRILL THROUGH` and the `SELECT` operators.

Q_{14}) Determine whether the association rule with PID=1001348 and confidence at least 0.8 is suitable to represent a certain dataset `itemsDS_30`, possibly different from the one from which the rule has been generated. The two correspondent statements in Fig. 8.15 are equivalent and return all data in `itemsDS_30` represented by the selected pattern.

Q_{15}) Determine whether the association rule with PID=1001348 and confidence at least 0.8 is suitable to represent a certain dataset `itemsDS_30`, possibly different from the one from which the rule has been generated. The correspond statements in Fig. 8.15 are equivalent.

```
Q10:SELECT * FROM (SELECT *
          FROM AR30_PSYCHO ar
          WHERE ar.m.confidence >= 0.7) rule
      WHERE isTvalid(rule,'10-aug-2005') = 1;

Q11:SELECT * FROM (SELECT *
          FROM AR30_PSYCHO ar
          WHERE ar.m.confidence >= 0.7) rule
        WHERE meets(rule.v,valPeriod('10-aug-2005','15-aug-2005'))=1;

Q12:DRILL THROUGH AR30_PSYCHO ar;

Q13:DRILL THROUGH (SELECT *
          FROM AR30_PSYCHO ar
          WHERE ar.PID = 1001348) rule;

Q14:DATA COVERING (SELECT *
          FROM AR30_PSYCHO pr
          WHERE pr.PID = 1001348) ar
        FOR itemsDS_30
        WHERE ar.m.support>=0.4;

Q14(b):DATA COVERING (SELECT *
          FROM AR30_PSYCHO pr
          WHERE pr.PID = 1001348
          AND pr.m.support>=0.4) ar FOR itemsDS_30;

Q15:PATTERN COVERING itemsDS_30 FOR AR30_PSYCHO ar
      WHERE ar.m.confidence >= 0.8;

Q15(b):PATTERN COVERING itemsDS_30 FOR (SELECT *
        FROM AR30_PSYCHO pr
        WHERE pr.m.confidence >= 0.8) ar;
Q16:SELECT *
    FROM AR30_PSYCHO ar
    WHERE isSvalid(ar,ar.d,'AR_MEASURE_FUNC', AssociationRuleMeasure(null,0.4,0.4))=1;
```

Figure 8.15: Scenario 1: complex query examples

Pattern Validity. *PSYCHO* supports two types of validity: temporal validity and semantic validity. A pattern is temporally valid with respect to a certain date (or a date interval) if its

validity period contains the specified date (or the date interval). A pattern is semantically valid with respect to a certain dataset and a set of thresholds if the pattern measures computed over the input datasets are better than those provided as input. As far as we know, no system supports this kind of analysis. In order to check temporal and semantic validity, *PSYCHO* supports specific predicates.

For example, the previous queries Q_9 and Q_{10} rely on a temporal validity check. On the other side, the following query applies a semantic validity check (for the code see Fig. 8.15).

Q_{16}) Retrieve from `AR30_PSYCHO` all semantically valid rules (with respect to their data source), with support and confidence greater than or equal to 0.4.

Other Manipulation Operations. Differently from most existing systems and standards, *PSYCHO* supports various types of update operations. The most important update operation is synchronization that allows one to synchronize pattern measures with the current data source, that may be changed with respect to its status at extraction time. For example, the $\mathcal{PSY}$-PML statement shown in Fig. 8.16 (lines 1-4) synchronizes all association rules contained in class `AR30_PSYCHO`, using measure function `AR_measure_func`.

A second update operation allows one to update the validity period. An example of an application of this manipulation operation over association rules contained in class `AR30_PSYCHO` is shown in Fig. 8.16 (lines 6-8).

After having performed the association rules synchronization, it may be useful to validate them with respect to fixed numerical parameters. Validate a pattern means possibly recomputing new patterns when the quality of the representation achieved by a set of pattern decreases (thus, the recomputed measures are worst than the existing ones). For instance, in Fig 8.16 (lines 10-13) an update statement which validates rules in class `AR30_PSYCHO` and inserts the new created patterns are inside the same class is presented.

```
1:  UPDATE PATTERNS OF AssociationRule ar
2:  SYNCHRONIZE
3:  USING AR_measure_func
4:  WHERE INCLASS(ar,'AR30_PSYCHO')=1;
5:
6:  UPDATE PATTERNS OF AssociationRule ar
7:  SET VALIDITY FROM '10-jun-2005' TO '31-aug-2005'
8:  WHERE INCLASS(ar,'AR30_PSYCHO')=1;
9:
10: UPDATE PATTERNS OF AssociationRule ar
11: VALIDATE USING AR_measure_func
12: WHERE inclass(ar, 'AR30_PSYCHO')=1
13: INTO CLASS AR30_PSYCHO;
```

Figure 8.16: Scenario 1: manipulation examples concerning *AssociationRule*

8.2.1.2 Clusters of 2-D Points

As a further example, we consider clusters of 2-dimensional points. Such clusters can be represented in several ways. We consider two distinct representations:

1. *Extensional clusters*: in this case, we assume a cluster is represented by a set of points belonging to it.
2. *CH-clusters*: in this case, we assume that the cluster is identified by the convex region containing all the points belonging to it. Such region corresponds to the convex hull of such points. The cluster in this case is represented by the vertexes of such region.

The quality of the representation achieved by both types of clusters can be evaluated by the intra distance measure. The definition of such pattern type shows the $PSYCHO$ capabilities in representing and manipulating the image formula.

Set-up. In this phase, we define the pattern type, the mining function required to extract clusters from source data, and the measure function, required to recompute measures upon a given data source.

Data Source. We assume source data is stored in a relational table with schema $(DSID, X, Y)$, with a tuple for each 2D point belonging to the distribution.

Pattern Type. Two distinct pattern types, one for extensional clusters and one for CH-clusters, are defined in the following. We assume that a cluster contains at most 100 points.

```
1:  CREATE TYPE POINT2D AS OBJECT (X real, Y real);
2:  CREATE TYPE POINT2DARRAY AS varray[100] OF POINT2D;
3:
4:  CREATE OR REPLACE FUNCTION Pointarray_Equal(a Point2DArray, b Point2DArray)
5:  RETURN INTEGER AUTHID CURRENT_USER IS
6:  i int; j int; cond int;
7:  BEGIN
8:  cond := 0; i := a.first;
9:  if a.count <> b.count then return 0; end if;
10: while i <= a.last loop
11:   j := b.first;
12:   while j <= b.last and cond=0 loop
13:     if a(i).x = b(j).x and a(i).y = b(j).y
14:       then cond:= 1;
15:       else j:= j+1;
16:     end if;
17:     if cond = 0 then return 0; end if;
18:     i:= i+1;
19:   end loop;
20: end loop;
21: return 1;
22: END pointarray_equal;
```

Figure 8.17: Scenario 1: *Cluster*'s support items definition

```
1:  CREATE PATTERN TYPE EXTCluster
2:  STRUCTURE pointset POINT2DARRAY
3:  DEFINE EQUALS ON p1 USING ret int CODE
4:    if Pointarray_Equal (self.s.pointset, p1.s.pointset)=1
5:      then ret:= 1;
6:      else ret:=0;
7:    end if;
8:  RETURN ret;
9:  MEASURE IntraDist REAL
10: DEFINE THETA ON p2 USING ret int CODE
11:   if (self.m.IntraDist >= p2.m.IntraDist)
12:     then ret:=1;
13:     else ret:= 0;
14:   end if;
15: RETURN ret;
16: FORMULA EXTENSIONAL ON varDS USING dummy varchar2(100) CODE
17: For i IN self.s.pointset.first ..  self.s.pointset.last Loop
18:   Execute Immediate 'INSERT INTO EXTClusterFormulaE (SELECT * FROM ' ||
19:   varDS || ' WHERE X= ' || self.s.pointset(i).x ||
20:   ' AND Y= ' || self.s.pointset(i).y || ') ';
21: End Loop;
22: FORMULA INTENSIONAL convex_hull;
23:
24: CREATE PATTERN TYPE CHCluster
25: STRUCTURE pointset POINT2DARRAY
26: DEFINE EQUALS ON p1 USING ret int CODE
27:   if Pointarray_Equal (self.s.pointset, p1.s.pointset)=1
28:     then ret:= 1;
29:     else ret:=0;
30:   end if;
31: RETURN ret;
32: MEASURE IntraDist REAL
33: DEFINE THETA ON p2 USING ret int CODE
34:   if (self.m.IntraDist >= p2.m.IntraDist)
35:     then ret:= 1;
36:     else ret:= 0;
37:   end if;
38: RETURN ret;
39: FORMULA EXTENSIONAL ON varDS USING dummy varchar2(100) CODE
40: For i IN self.s.pointset.first ..  self.s.pointset.last Loop
41:   Execute Immediate 'INSERT INTO EXTClusterFormulaE (SELECT * FROM ' ||
42:   varDS || ' WHERE X= ' || self.s.pointset(i).x ||
43:   ' AND Y= ' || self.s.pointset(i).y || ') ';
44: End Loop;
45: FORMULA INTENSIONAL convex_hull;
```

Figure 8.18: Scenario 1: pattern type for modeling clusters of 2D points

As we have already pointed out, as default an extensional formula returning the whole pattern data source is associated with a pattern. In this case, we want to specify a different behavior for the extensional formula associated with extensional clusters. In particular, since the structure of extensional cluster contains exactly the points that belong to the cluster, the extensional formula method has to return exactly all points in the pattern structure. In this second case, since the cluster structure approximates the cluster shape, the extensional formula associated with the convex cluster is not very significant. On the other hand, in this case, much more meaning is associated with the intensional formula pattern component, which contains the constraint set describing the convex hull (i.e. the convex hull edges) evaluated by applying the Prolog predicate named `convex_hull`, defined in file `convex_hull.pl`, whose content is shown in Fig. 8.19.

```
1:  convex_hull(Points, Xs) :-
2:    lin_comb(Points, Lambdas, Zero, Xs),
3:    zero(Zero),
4:
5:  polytype(Lambdas).
6:  polytype(Xs) :-
7:    positive_sum(Xs, 1).
8:
9:  positive_sum([], Z) :- Z=0.
10: positive_sum([X|Xs], SumX) :-
11:    X >= 0, SumX = X+Sum ,
12:   positive_sum(Xs, Sum).
13:
14: zero([]).
15: zero([Z|Zs]) :- Z=0, zero(Zs).
16:
17: lin_comb([], [], S1, S1).
18: lin_comb([Ps|Rest], [K|Ks], S1, S3) :-
19:   lin_comb_r(Ps, K, S1, S2),
20:   lin_comb(Rest, Ks, S2, S3).
21:
22: lin_comb_r([], _, [], []).
23: lin_comb_r([P|Ps], K, [S|Ss], [Kps|Ss1]) :-
24:    Kps = K*P+S ,
25: lin_comb_r(Ps, K, Ss, Ss1).
```

Figure 8.19: Scenario 1: *convex_hull* Prolog predicate

Mining Function. We consider two mining functions for each cluster type. The implementation of both these mining functions is based on the K-means algorithm provided by ODM and is parametric with respect to the value of parameter K. We call such functions: `KMeansODM` and `KMeansODM_convex`.

Measure Function. We consider one measure function for each cluster type, names `EC_Measure_func` and `CH_Measure_func`, the first for extensional clusters and the second for CH-clusters. Such measure functions compute the intra-distance value with respect to the points belonging to the pattern structure. The $\mathcal{PSY}$-PDL statement used to define `EC_Measure_func` is shown in Fig. 8.20.

```
1:  CREATE MEASURE FUNCTION EC_Measure_func
2:  FOR EXTCluster ec
3:  USING varDS AS tot real, numP real
4:  BEGIN
5:    tot := 0; NumP := ec.s.pointset.count;
6:    For i IN ec.s.pointset.first ..  ec.s.pointset.last Loop
7:      For j IN ec.s.pointset.first ..  ec.s.pointset.last Loop
8:        if i<>j
9:        then tot := tot + sqrt((power((ec.s.pointset(i).x
10:       - ec.s.pointset(j).x),2))
11:       + (power((ec.s.pointset(i).y
12:       - ec.s.pointset(j).y),2)));
13:       end if;
14:     End loop;
15:   End loop;
16:   tot:= tot/numP;
17:   RETURN Measure(tot);
18:   END;
```

Figure 8.20: Scenario 1: measure function definition for $EXTCluster$

Population. In this step we show how $PSYCHO$ can be used to: (i) use various mining functions to extract patterns of type `EXTCluster` and `CHCluster`; (ii) directly insert patterns.

Extraction. Suppose we want to extract patterns of type `EXTCluster` from the dataset `Points30` by using the `KmeansODM` mining function. First of all, we create two classes for storing the extracted extensional clusters ($K = 5$). We call it `ClassK5_EC` and `ClassK5_CH` (see Fig. 8.21, lines 1-2).

Now, we extract clusters with intra distance greater than 0.3 from dataset `Points30` and we set the validity period of the extracted clusters from 10-jun-2005 to 10-jul-2005 (see Fig. 8.21, lines 11-18).

```
1:  CREATE CLASS ClassK5_EC OF EXTCluster;
2:  CREATE CLASS ClassK5_CH OF CHCluster;
3:
4:  EXTRACT PATTERNS OF EXTCluster ec
5:  FROM Points30
6:  USING KmeansODM(5)
7:  WITH (0.3)
8:  VALID FROM '10-jun-2005' TO '10-aug-2005'
9:  INTO CLASS ClassK5_EC;
10:
11: EXTRACT PATTERNS OF CHCluster cc
12: FROM Points30
13: USING KmeansODM_Convex(5)
14: WITH (0.3)
15: VALID FROM '10-jun-2005' TO '10-jul-2005'
16: INTO CLASS ClassK5_CH; 17:
18: CREATE CLASS ClassK6_EC OF EXTCluster;
19:
20: EXTRACT PATTERNS OF EXTCluster ec
21: FROM points60
22: USING KmeansODM(6)
23: WITH (0.3)
24: VALID FROM CURRENT_DATE TO CURRENT_DATE+60
25: INTO CLASS ClassK6_EC;
26:
27: DIRECT INSERT PATTERN OF EXTCluster
28: FROM Points60
29: STRUCTURE(POINT2DARRAY(POINT2D(1,1), POINT2D(2,1), POINT2D(3,1),
30: POINT2D(2,2), POINT2D(3,2), POINT2D(2,3)) )
31: VALID FROM CURRENT_DATE TO CURRENT_DATE+30
32: INTO CLASS ClassK5_EC;
33:
34: DIRECT INSERT PATTERN OF CHCluster
35: FROM Points60
36: STRUCTURE(POINT2DARRAY(POINT2D(1,1), POINT2D(3,1),
37: POINT2D(3,2), POINT2D(2,3)) )
38: MEASURE (0.3)
39: VALID FROM '01-aug-2005' TO '15-aug-2005'
40: INTO CLASS ClassK5_CH;
```

Figure 8.21: Scenario 1: manipulation examples concerning *EXTCluster* and *CHCluster*

Note that, the user may customize the previous extraction operations by specifying how many clusters she wants to obtain as output of the clustering algorithm (the value of the parameter K). The $\mathcal{PSY}$-PML statements performing the extraction of 6 extensional clusters from `Points60` dataset and their insertion into the new class `ClassK6_EC` is shown in Fig. 8.21 (lines 20-25).

Direct Insertion. $PSYCHO$ supports the direct insertion of patterns. For example, we can insert a new pattern in the previously created classes `ClassK5_EC` and `ClassK5_CH` (see Fig. 8.21, lines 27-40).

Queries. *Analysis of Different Pattern Representation.* The aim of the following queries is to point out the various pattern representations available in $PSYCHO$.

In the following we describe several examples of possible queries (see Fig. 8.22 for the correspondent codes):

Q_1) Determine data from which cluster with `PID=1001615` has been extracted. The correspondent statement retrieves the overall set of data from which the cluster has been extracted.

Q_2) Determine data approximatively represented by the cluster with `PID=1001615`, contained in class `ClassK5_EC`. For this query, it is sufficient to select the pattern. The intensional formula obtained as result corresponds to such approximate representation.

Q_3) Determine which data items contained in the source dataset `Points30` are represented by the cluster with `PID=1001615`, contained in class `ClassK5_EC`. For this query, we can use the `DATA COVERING` operation, relying on the extensional formula. The obtained data satisfied the intensional formula associated with the selected pattern, since data covering relies on the application of the extensional formula.

```
Q1: DRILL THROUGH ClassK5_EC cl
        WHERE cl.PID = 1001615;

Q2: SELECT *
        FROM ClassK5_EC cl
        WHERE cl.PID = 1001615;

Q3: DATA COVERING (SELECT *
          FROM ClassK5_EC cl
          WHERE cl.PID = 1001615) c
        FOR Points30
        WHERE c.PID = 1001615;

Q4: SELECT *
      FROM ClassK5_EC cl
      WHERE cl.d = 'Points30'
      I_AND icontain(cl.formulaI,[X>=2,X=<7,Y>=3,Y=<10]);

Q5: SELECT *
      FROM ClassK5_EC cl
      WHERE cl.d = 'Points30'
      I_AND iintersect(cl.formulaI,[X>=2,X=<7,Y>=3,Y=<10]);
```

Figure 8.22: Scenario 1: queries over a class defined for $EXTCluster$

Queries Involving the Intensional Formula. The aim of the following queries is to show the usage of the intensional formula, (see Figure 8.22 for the correspondent codes).

Q_4) Determine clusters in class `ClassK5_EC`, extracted from data source `Points30`, that contain a certain region of space, characterized by the following formula `[X>=2,X=<7,Y>=3,Y=<10]`. Notice that the obtained result is an approximation of the real one since actual points belonging to the cluster are not taken into account. Predicate `icontain` is checked at the intensional level, by using the Prolog definition of the intensional formula (no access to data sources is required).

Q_5) Determine clusters in class `ClassK5_EC`, extracted from data source `Points30`, that intersect a certain region of space, characterized by the following formula `[X>=2,X=<7,Y>=3,Y=<10]`.

8.2.2 Scenario 2

The aim of the second scenario is to show representation, manipulation, and querying for hierarchies of patterns. To this purpose, we consider clusters of association rules. Indeed,

in some data mining applications, it may be useful to group association rules describing correlations among sold products based on some grouping criteria. In this scenario, we group association rules based on their semantic validity with respect to a certain data source. In this way, we are able to show how $PSYCHO$ supports hierarchies of patterns.

We outline that, in this scenario, we apply a very simple clustering criteria. However, this does not impact the issue we wish to demonstrate. Indeed, it is even possible to extend $PSYCHO$ library with more sophisticated data mining or machine learning algorithms in order to deal with more complex clustering purposes.

Set-up.

Data Source. The considered data source is classes of association rules, created in Section 8.2.1.

Pattern Type. Clusters of association rules can be modeled by using a pattern type `EXTClusterOfRules`, defined in $\mathcal{PSY}$-PDL as shown in Fig. 8.23 (lines 22-43). Note that, as default an extensional formula returning the whole pattern data source is associated with a pattern of type `EXTClusterOfRule`, but no intensional formula is associated with patterns of this type.

```
1:  CREATE TYPE ARSET AS varray(100) OF REF AssociationRule;
2:
3:  CREATE OR REPLACE FUNCTION ARSET_EQUAL (a ARSET, b ARSET)
4:  RETURN INTEGER AUTHID CURRENT_USER
5:  IS i int; j int; cond int; p1 PatternType; p2 PatternType;
6:  BEGIN
7:    cond := 0; i := a.first;
8:    if a.count <> b.count then return 0; end if;
9:    while i <= a.last loop
10:     cond := 0; j := b.first;
11:     while j <= b.last AND cond=0 loop
12:       SELECT DEREF(pattern) INTO p1 FROM PBMScat WHERE pattern = a(i);
13:       SELECT DEREF(pattern) INTO p2 FROM PBMScat WHERE pattern = b(j);
14:       if identity(p1,p2)=1 then j:=j+1; else cond:=1; end if;
15:     end loop;
16:   if cond = 0 then return 0; end if;
17:   i:=i+1;
18: end loop;
19: return cond;
20: END ARSET_EQUAL;
21:
22: CREATE PATTERN TYPE EXTClusterOfRules
22: STRUCTURE ruleset ARSET
23: DEFINE EQUALS ON r1 USING ret int CODE
24: if ARSET_EQUAL(self, p1)=1 then ret:=1; else ret:=0; end if;
25: return ret;
26: MEASURE Svalidity REAL
27: DEFINE THETA ON r2 USING ret int CODE
28: if self.m.Svalidity >= r2.m.Svalidity then ret:=1; else ret:= 0; end if;
29: return ret;
30: FORMULA EXTENSIONAL ON varDS USING p AssociationRule CODE
31:   For i in self.s.ruleset.first ..  self.s.ruletset.last Loop
32:     Execute Immediate 'INSERT INTO EXTClusterOfRulesFormulaE (
33:     SELECT * FROM ' || varDS || ' WHERE pattern = ' || ruleset(i) || ') ';
34:   End Loop;
35: FORMULA INTENSIONAL dummy_predicate; 36:
37: CREATE MEASURE FUNCTION CRules_measure_func
38: FOR EXTClusterOfRules cr USING varDS
39: AS val real BEGIN
40:   val := isSvalid(cr, 'varDS', 'CRules_own_measure',
41:   EXTClusterOfRulesMeasure(null,0));
42:   RETURN MEASURE(val);
43: END;
```

Figure 8.23: Scenario 2: set-up commands

Mining Function. Since, for the sake of simplicity, we want to define clusters of association rules based on validity, the mining function is very simple and just divide a set of

association rules into two clusters. The mining function we use in the following is named `EXTClusterOfRulesMF`.

Measure Function. We assume the mining function just returns 1 when the pattern is semantically valid with respect to the specified data source. Otherwise, it returns 0. Thus, measure function `CRules_measure_func` can be easily defined as shown in Fig. 8.23 (lines 37-43).

```
1:  CREATE CLASS ClassCR_1 OF EXTClusterOfRules;
2:
3:  EXTRACT PATTERNS OF EXTClusterOfRules ec
4:  FROM AR30_PSYCHO
5:  USING EXTClusterOfRuleMF('itemsDS_30')
6:  WITH (0)
7:  VALID FROM '01-aug-2005' TO '30-sep-2005'
8:  INTO CLASS ClassCR_1;
```

Figure 8.24: Scenario 2: population phase

8.2.2.1 Population

We consider only extraction.

Extraction. We first create a class for storing the extracted clusters, named `ClassCR_1` (see Fig. 8.24 (line 1)).

After that, patterns of type `EXTClusterOfRules` can be extracted from the *PSYCHO* class `AR30_PSYCHO` by using the `EXTClusterOfRuleMF` mining function introduced above, setting the validity period of the extracted clusters starting from August, 1 2005 and ending in September, 30 2005 (see Fig. 8.24 (lines 3-8)). Note that the data source in this case is class `AR30_PSYCHO` and the mining function has a parameter which corresponds to a source dataset for association rules, used to check semantic validity.

Queries. In the following several queries involving clusters of rules are presented (see Fig. 8.25 for the correspondent codes).

Q_1) Retrieve all patterns from class `ClassCR_1` containing at least 4 rules.

Q_2) Retrieve all patterns from class `ClassCR_1` containing at least N rules with support greater than 0.45. Notice that for this query a condition is required since arrays cannot be manipulated inside SQL but only through PL/SQL.

Q_3) Determine the association rule dataset from which the cluster with a certain PID, belonging to class `ClassCR_1`, has been generated.

```
Q1: SELECT *
    FROM ClassCR_1 cr
    WHERE cr.s.ruleset.count >= 4;

Q2: CREATE CONDITION cl_supp_length FOR EXTClusterOfRules cr AS
      ccount int, myarset arset, myar AssociationRule
      BEGIN
      ccount:= 0;
      SELECT
        TREAT(mycr.s as EXTClusterOfRulesstructure).ruleset INTO myarset
      FROM EXTClusterOfRulestab mycr
      WHERE mycr.PID = cr.PID;
      for i in myarset.FIRST ..  myarset.LAST loop
        SELECT VALUE(ar) INTO myar
        FROM AssociationRuleTab ar
        WHERE REF(ar) = myarset(i);
      if TREAT(myar.m AS AssociationRuleMeasure).support > 0.45
      then ccount := ccount + 1;
      end if;
      end loop;
      if ccount >= 1 then return 1; end if;
      return 0;
    END;

    SELECT *
    FROM ClassCR_1 cr
    WHERE cl_supp_length(cr)=1;

Q3: DRILL THROUGH
    ( SELECT * FROM ClassCR_1 c WHERE c.PID = 1001454) a;
```

Figure 8.25: Scenario 2: query examples

8.2.3 Scenario 3

The aim of the third scenario is to show representation, manipulation, and querying for user-defined patterns. To this purpose, we consider 2-dimensional intervals, that may correspond to portions of moving object trajectories.

Set-up.

Data Source. Any set of 2-dimensional points can be considered as data source.

Pattern Type. Since each interval is defined by an equation such as $aX+bY = c\,AND\,X \leq d\,ANDX \geq e$, each interval can be represented as a tuple (a, b, c, d, e). The intensional formula and the extensional formula can be easily defined starting from such equation. The

measure can be defined as the variance of the distances of moving object positions with respect to the corresponding interval. The statement for creating the *Trajectory* pattern type is presented in Fig. 8.26.

```
1:  CREATE PATTERN TYPE Trajectory
2:  STRUCTURE a real, b real, c real, d real, e real
3:  DEFINE EQUALS ON t1 USING ret int CODE
4:    if (self.s.a = t1.s.a and self.s.b = t1.s.b and
5:      self.s.c = t1.s.c and self.s.d = t1.s.d and
6:      self.s.e = t1.s.e )
7:    then ret:= 1; else ret:=0;
8:    end if;
9:  RETURN ret;
10: MEASURE MeanVariance REAL
11: DEFINE THETA ON t2 USING ret int CODE
12: if self.m.MeanVariance >= t2.m.MeanVariance
13:   then ret:= 1; else ret:= 0;
14: end if;
15: RETURN ret;
16: FORMULA EXTENSIONAL ON varDS
17: USING cur Types.cursor_type, x real, y real, dsid int CODE
18: OPEN cur FOR 'SELECT * FROM ' || varDS; LOOP
19:   FETCH cur INTO dsID, x, y;
20:   EXIT WHEN cur%NOTFOUND;
21:   if ((self.s.a*x)+(self.s.b*y) = self.s.c)
22:     and x>= self.s.d and x<=self.s.e
23:   then EXECUTE IMMEDIATE
24:     'INSERT INTO TrajectoryFormulaE values (:dsID, :x, :y)'
25:     USING dsID, x, y;
26:   end if;
27: END LOOP;
28: FORMULA INTENSIONAL segments;
```

Figure 8.26: Scenario 3: creation of the *Trajectory* pattern type

Population. For this example, we just consider an example of direct insertion (see Fig. 8.27).

```
1:  CREATE CLASS ClassTrj1 OF Trajectory;
2:
3:  DIRECT INSERT PATTERN OF Trajectory
4:  FROM Points30
5:  STRUCTURE (2,3,6,7,9)
6:  MEASURE (0.45)
7:  VALID FROM '01-aug-2005' TO '10-aug-2005'
8:  INTO CLASS classTrj1;
```

Figure 8.27: Scenario 3: population phase

Queries. Patterns of the new user-defined pattern type can be queried similarly to all the other "standard" patterns. In the following we describe several examples of possible queries (see Fig. 8.28 for the correspondent codes).

Q_1) Retrieve all $Trajectory$ patterns from $ClassTrj1$.

Q_2) Retrieve all $Trajectory$ patterns from $ClassTrj1$ which are temporally valid on August, 1 2005 and with mean variance less than 10.

Q_3) Retrieve all $Trajectory$ patterns from $ClassTrj1$ which are temporally valid on August, 1 2005 and intersecting a given area.

Q_4) Determine all trajectories temporally valid in date '07-aug-2005' belonging to class `ClassTrj1`, covering a given area.

Q_5) We want to determine which data items contained in the source dataset `Points60` are covered by trajectories valid on 10-Aug-2005 with a variance value lower than 10 contained in class `ClassTrj1`.

```
Q1: SELECT * FROM ClassTrj1 t;

Q2: SELECT * FROM ClassTrj1 t
    WHERE ISTVALID(t,'10-aug-2005')=1 AND t.m.meanVariance <10;

Q3: SELECT * FROM ClassTrj1 t
    WHERE ISTVALID(t,'07-aug-2005')=1 I_AND
    IINTERSECT(t.formulaI,[X>=7,X=<8.5,-Y=<10,Y=<4]);

Q4: SELECT * FROM ClassTrj1 t
    WHERE testalwaystrue(t)=1 I_AND
    IEQUIV(t.formulaI,[X>=7,X=<8.5,Y>=1,Y=<4]);

Q5: DATA COVERING ClassTrj1 t FOR Points60
    WHERE ISTVALID(t,'10-aug-2005')=1 AND t.m.meanVariance < 10;
```

Figure 8.28: Scenario 3: Query examples

Chapter 9

Conclusions

In this thesis, we have presented a framework for the management of heterogeneous patterns under a unified approach. The proposed framework is general enough to handle many different kinds of patterns deriving from different application contexts, such as basic data mining patterns (e.g., association rules and clusters) or complex data mining patterns (e.g., clusters of association rules). Moreover, it supports user defined patterns, do not necessary resulting from a mining process (e.g., moving objects trajectory equations).

We wish to outline that the research work underlying this PhD thesis has started in the context of the PANDA European Project. Thus, some of the initial activities was partially founded by the European Commission through the PANDA IST Thematic Network [PAN02].

We conclude this thesis by briefly reviewing the main contributions of the PhD research work (Section 9.1) and by discussing several open issues for further research activities (Section 9.2).

9.1 Summary of the Contributions

The major contributions of this thesis can be summarized as follows.

- A logical model for patterns supporting advanced modeling features, such as, user defined patterns, exact or approximated representation of raw data effectively represented by a pattern, validity aspects related to patterns, and pattern hierarchies, has been formalized.
- A pattern manipulation language supporting the management of both a-priori and

a-posteriori patterns, along with pattern-data synchronization and validity issues has been proposed.

- A pattern query algebra and a pattern query calculus providing features for the retrieval, the analysis, and the combination of patterns together and with raw data have been proposed. Moreover, theoretical aspects concerning the expressive power and complexity of the proposed query languages have been addressed.

- A prototype system based on the proposed logical model and languages has been designed and developed. It is called PSYCHO (Pattern based management SYstem arChitecture prOtotype) and it has been implemented by exploiting object-relational database technologies (provided by the Oracle platform [Ora]) integrated with logical constraint solving features (provided by the Sicstus Prolog engine [sic]) through the usage of object-oriented programming (Java platform [jav] and the Jasper interface [jas] versus Prolog).

Some of the results achieved in the context of this PhD thesis have already been presented - or will be presented in the next future - at the international scientific community.

The foundations of the defined logical model for patterns have been proposed in the context of the PANDA consortium [BCG+03]. At the beginning of the work, we had a positive feedback on the PhD research program proposal [Mad04]. A comparative analysis of pattern management solutions has been presented in [CM], where several parameters for the evaluation of pattern management proposals have been identified.

Our proposal for the logical model for patterns has been presented in [TVS+04] and its temporal extension can be found in [CMM+04]. Concerning query and manipulation languages, we have proposed a sketch of the operators for querying patterns in [BCM04] with a possible application scenario in the area of Web analysis [CM04]. Then, temporal manipulation and querying operations have been presented in [CMM+04].

Concerning *PSYCHO*, we have recently presented and demonstrated the system [CMM05, CM06]. Technical documents concerning the prototype are available at [PSY] where also a set-up package to install and configure the entire prototype system can be downloaded.

9.2 Topics for Further Research

In order to make pattern management a practical technology, besides the topics covered by this thesis, additional issues have to be taken into account when developing a PBMS. In particular, two different directions for future work can be devised: the first one concerns the extension of the proposed prototype pattern based management system proposed; the

second one concerns additional issues aimed at consolidating the proposed framework. Both directions are briefly discussed in the following.

9.2.1 PSYCHO Extensions

In order to become an effective usable system for pattern management, PSYCHO functionalities have to be consolidated and expanded. Directions for future work concerning PSYCHO implementation includes the following issues:

1. The development of a graphical user interface to interact with the system and to visualize in a user-friendly way discovered knowledge.
2. The integration with existing standards for pattern representation, especially with respect to PMML standard [PMM].
3. The design and development of an *open-source* version of the system do not requiring a licensed constraint solving engine, like Sicstus Prolog, in order to deal with intensional aspects concerning pattern management. This version of the system would constitute the first step toward the development of a free pattern based management system relying on open-source technologies.
4. The experimentation of PSYCHO over real application domains, such as: retail analysis, clickstream analysis concerning Web navigations, or biomedical applications.

9.2.2 Other Open Issues

In the following, further topics for future research work on pattern management are briefly discussed. Such topics have only been partially taken into account by existing proposals.

Pattern reasoning. The capability to apply a reasoning process over patterns is supported only in few theoretical proposals, in the form of similarity check [BCN+04] or pattern combination [JLN00, CMM+04]. However, an overall approach for reasoning about possibly heterogeneous patterns needs more sophisticated techniques, describing the semantics of pattern characteristics. As an example, consider measures. In general, various approaches exist for measure computation (general probabilities, Dempster-Schafer, Bayesian Networks - see for example the proposal of Silberschatz and Tuzhilin [ST96]). It is not clear how patterns, possibly having the same type but characterized by different measures, can be compared and managed together. Probably, measure ontologies can be used to support such kind of quantitative pattern reasoning.

Design of a physical model for patterns. Since patterns are assumed to be stored in a repository, specific physical design techniques must be developed. Unfortunately, it is not clear up to now what should be a reasonable physical layer for patterns. Most commercial DBMSs store patterns as BLOB that are then manipulated using specific methods. However in order to provide a more efficient access, specific physical representations, clustering, partitioning, caching, and indexing techniques should be developed. Concerning theoretical proposals, as we have already seen, in the context of the 3W model, patterns are represented as regions, thus techniques developed for spatial databases can be used for their physical management In the context of PSYCHO, methods for object-relational model have to be customized in order to ope with PSYCHO peculiarities.

Query optimization. Query optimization for pattern queries is an interesting topic for further research work. Some preliminary work, concerning query rewriting, has been proposed in the context of the 3W framework. However, an overall query optimization approach, taking into account choices concerning the physical design, has not been defined yet. Assuming to deal with a separated architecture, the main issue is how to perform data and patterns computations in an efficient way. An important issue here is how it is possible to use patterns to reduce data access in data and cross-over queries and how data and pattern query processors can be combined. On the other side, under an integrated architecture, where extraction is a kind of query, the main issue is the optimization of pattern generation. Some work in this direction has been done in the context of inductive databases, where approaches to optimize pattern extraction, based on constraints on pattern properties [NHL98] or to refine the set of generated patterns [BP99], have been proposed. Techniques for reducing the size of the generated pattern sets, by representing them using condensed representations, have also been proposed for itemsets and association rules [CIN01].

Access control. Patterns represent high-sensitive information. Their access has, therefore, to be adequately controlled. The problem is quite similar to access control in presence of inference [FJ02]. In general, assuming a user has access to some non-sensitive data, the inference problem arises when, through inference, sensitive data can be discovered from non-sensitive one. In terms of patterns, this means that users may have the right to access some patterns, for example some association rules, and starting from them they may infer additional knowledge over data, over which they may not have the access right.

Techniques already proposed in the inference context should be adapted and extended to cope with the more general pattern management framework. Some of these approaches rely on pre-processing techniques, checking through mining techniques whether it is possible to infer sensitive data; some others can be applied at runtime, i.e. during the knowledge discovery phase, releasing patterns only when they

do not represent sensitive information. Finally, modifications over original data, such as perturbation and sample size restrictions, do not disturbing data mining results can be also applied in order to encrypt the original data and to prevent unauthorized user data access.

Bibliography

[AB95] Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *VLDB J.*, 4(4):727–794, 1995.

[ACGK94] Foto N. Afrati, Stavros S. Cosmadakis, Stéphane Grumbach, and Gabriel M. Kuper. Linear vs polynomial constraints in database query languages. In *PPCP '94: Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, pages 181–192, London, UK, 1994. Springer-Verlag.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AR00] Juan M. Ale and Gustavo H. Rossi. An approach to discovering temporal association rules. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 294–300, New York, NY, USA, 2000. ACM Press.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[BBFS98] Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, 1998.

[BCC+02] Daniele Braga, Alessandro Campi, Stefano Ceri, Mika Klemettinen, and Pier Luca Lanzi. A tool for extracting xml association rules. In *ICTAI*, pages 57–, 2002.

[BCG+03] Elisa Bertino, Barbara Catania, Matteo Golfarelli, Maria Halkidi, Anna Maddalena, Spriros Skiadopoulos, Stefano Rizzi, Manolis Terrovitis, Panos Vassiliadis, Michalis Vazirgiannis, and Euripides Vrachnos. The logical model for patterns. Technical Report PANDA Technical report TR-2003-02, 2003.

[BCH00] Andi Baritchi, Diane J. Cook, and Lawrence B. Holder. Discovering structural patterns in telecommunications data. In *FLAIRS Conference*, pages 82–85. AAAI Press, 2000.

[BCM04] Elisa Bertino, Barbara Catania, and Anna Maddalena. Towards a language for pattern manipulation and querying. In *PaRMa*, 2004.

[BCN+04] Ilaria Bartolini, Paolo Ciaccia, Irene Ntoutsi, Marco Patella, and Yannis Theodoridis. A unified and flexible framework for comparing simple and complex patterns. In *PKDD*, pages 496–499, 2004.

[BEJ+00] Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. *The object data standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[BFGM03] Elisa Bertino, Elena Ferrari, Giovanna Guerrini, and Isabella Merlo. T-odmg: an odmg compliant temporal object model supporting multiple granularity management. *Inf. Syst.*, 28(8):885–927, 2003.

[BJW00] Claudio Bettini, Sushil G. Jajodia, and Sean X. Wang. *Time Granularities in Databases, Data Mining and Temporal Reasoning*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[BK95] Alexander Brodsky and Yoram Kornatzky. The lyric language: Querying constraint objects. In *SIGMOD Conference*, pages 35–46, 1995.

[BL04] Michael J. A. Berry and Gordon S. Linoff. *Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management. 2nd Edition*. Wiley, 2004.

[BP99] Elena Baralis and Giuseppe Psaila. Incremental refinement of mining queries. In *DaWaK*, pages 173–182, 1999.

[BR95] Jo-Hag Byon and Peter Z. Revesz. Disco: A constraint database system with sets. In *Workshop on Constraint Databases and Applications (CDB)*, Lecture Notes in Computer Science, pages 68–83. Springer, 1995.

[BWJL98] Claudio Bettini, Xiaoyang Sean Wang, Sushil Jajodia, and Jia-Ling Lin. Discovering frequent event patterns with multiple granularities in time sequences. *IEEE Trans. Knowl. Data Eng.*, 10(2):222–237, 1998.

[BYRN99] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[CIN01] Cinq project. http://www.cinq-project.org, 2001.

[CM] Barbara Catania and Anna Maddalena. Pattern management: Practice and challenges. In *Processing and Managing Complex Data for Decision Support*. Idea Group, Inc.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Symposium on Theory of Computing*, pages 77–90, 1977.

[CM04] Barbara Catania and Anna Maddalena. A framework for cluster management. In *EDBT Workshops*, pages 568–577, 2004.

[CM06] Barbara Catania and Anna Maddalena. Flexible pattern management within psycho. In *PaRMa [To appear]*, 2006.

[CMM+04] Barbara Catania, Anna Maddalena, Maurizio Mazza, Elisa Bertino, and Stefano Rizzi. A framework for data mining pattern management. In *PKDD*, pages 87–98, 2004.

[CMM05] Barbara Catania, Anna Maddalena, and Maurizio Mazza. Psycho: A prototype system for pattern management. In *VLDB*, pages 1346–1349, 2005.

[CN94] Peter Clifford and Geoff Nicholls. *A Metropolis Sampler for Polygonal Image Reconstruction*. Dept. of Statistics, Oxford Univ., 1994.

[Con02] Darrell Conklin. Representation and discovery of vertical patterns in music. In *ICMAI '02: Proceedings of the Second International Conference on Music and Artificial Intelligence*, pages 32–42, London, UK, 2002. Springer-Verlag.

[CP00] Xiaodong Chen and Ilias Petrounias. Discovering temporal association rules: Algorithms, language and system. In *ICDE*, page 306, 2000.

[CR00] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211–229, 2000.

[CSD98] Soumen Chakrabarti, Sunita Sarawagi, and Byron Dom. Mining surprising patterns using temporal description length. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 606–617, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[CWM05] Common warehouse metamodel. http://www.omg.org/technology/cwm/, 2005.

[DB2] Db2 intelligent miner. http://www-306.ibm.com/software/data/iminer/.

[DCM] Dublin core metadata initiative. http://dublincore.org/.

[DDL+90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[DH02] Roger B. Dannenberg and Ning Hu. Discovering musical structure in audio recordings. In *ICMAI*, pages 43–57, 2002.

[DLLL97] S.T. Dumais, T.A. Letsche, M.L. Littman, and T.K. Landauer. Automatic cross-language retrieval using latent semantic indexing, 1997.

[ESF01] Mohamed G. Elfeky, Amani A. Saad, and Souheir A. Fouad. Odmql: Object data mining query language. In *Proceedings of the International Symposium on Objects and Databases*, pages 128–140, London, UK, 2001. Springer-Verlag.

[FJ02] Csilla Farkas and Sushil Jajodia. The inference problem: A survey. *SIGKDD Explorations*, 4(2):6–11, 2002.

[FPSS96] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery: An overview. In *Advances in Knowledge Discovery and Data Mining*, pages 1–34. 1996.

[GGR99] Venkatesh Ganti, Johannes Gehrke, and Raghu Ramakrishnan. A framework for measuring changes in data characteristics. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 126–137. ACM Press, 1999.

[GV91] Stéphane Grumbach and Victor Vianu. Tractable query languages for complex object databases. pages 315–327, 1991.

[HD04] Sherri K. Harms and Jitender S. Deogun. Sequential association rule mining with time lags. *J. Intell. Inf. Syst.*, 22(1):7–22, 2004.

[HFW+96] Jiawei Han, Yongjian Fu, Wei Wang, Krzysztof Koperski, and Osmar Zaiane. DMQL: A data mining query language for relational databases. In *SIGMOD'96 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'96)*, Montreal, Canada, 1996.

[HGN00] Jochen Hipp, Ulrich Guntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining a general survey and comparison. *SIGKDD Explor. Newsl.*, 2(1):58–64, 2000.

[HK01] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Academic Press, 2001.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[HS91] Richard Hull and Jianwen Su. On the expressive power of database queries with intermediate types. *J. Comput. Syst. Sci.*, 43(1):219–267, 1991.

[HYG02] Perfecto Herrera, Alexandre Yeterian, and Fabien Gouyon. Automatic classification of drum sounds: A comparison of feature selection methods and classification techniques. In *ICMAI*, pages 69–80, 2002.

[IM96] Tomasz Imielinski and Heikki Mannila. A database perspective on knowledge discovery. *Commun. Of The ACM*, 39(11):58–64, 1996.

[IV99] Tomasz Imielinski and Aashu Virmani. MSQL: A query language for database mining. *Data Min. Knowl. Discov.*, 3(4):373–408, 1999.

[jas] Jasper java interface. http://www.sics.se/sicstus/docs/latest/ html/sicstus/Jasper.html.

[jav] Products & technologies java technology. http://java.sun.com.

[JDM04] Java data mining api specification. http://www.jcp.org/en/jsr/detail?id=73, 2004.

[JLN00] Theodore Johnson, Laks V. S. Lakshmanan, and Raymond T. Ng. The 3w model and algebra for unified data mining. In *The VLDB Journal*, pages 21–32, 2000.

[JMF99] Anil K. Jain, M. Narasimha Murty, and Patric J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[KKR90] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. In *Symposium on Principles of Database Systems*, pages 299–313, 1990.

[Klu88] Anthony Klug. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160, 1988.

[Kou93] Manolis Koubarakis. *Foundations of Temporal Constraint Databases*. PhD thesis, 1993.

[Kou94] Manolis Koubarakis. Complexity results for first-order theories of temporal constraints. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *KR'94: Principles of Knowledge Representation and Reasoning*, pages 379–390. Morgan Kaufmann, San Francisco, California, 1994.

[Kou95] Manolis Koubarakis. Databases and temporal constraints: Semantics and complexity. In *Temporal Databases*, pages 93–109, 1995.

[Kou01] Manolis Koubarakis. Tractable disjunctions of linear constraints: basic results and applications to temporal reasoning. *Theoretical Computer Science*, 266(1–2):311–339, 2001.

[KSE] Knowledge sharing effort. http://www.cs.umbc.edu/kse/.

[Kup90] Gabriel M. Kuper. On the expressive power of the relational calculus with arithmetic constraints. In *ICDT*, pages 202–211, 1990.

[KV98] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. In *Symposium on Principles of Database Systems*, pages 205–213, 1998.

[LC00] Chen Li and Edward Y. Chang. Query planning with limited source capabilities. In *ICDE*, pages 401–412, 2000.

[LM92] Jean-Louis Lassez and Ken McAloon. A canonical form for generalized linear constraints. *J. Symb. Comput.*, 13(1):1–24, 1992.

[LNWJ03] Yingjiu Li, Peng Ning, X. Sean Wang, and Sushil Jajodia. Discovering calendar-based temporal association rules. *Data Knowl. Eng.*, 44(2):193–218, 2003.

[LS97] Alon Y. Levy and Dan Suciu. Deciding containment for queries with complex objects (extended abstract). In *PODS '97: Proceedings of the 16th ACM SIGMOD Symposium on Principles of Database Systems*, pages 20–31, 1997.

[Mad04] Anna Maddalena. Pattern based management: Data models and architectural aspects. In *EDBT Workshops*, pages 54–65, 2004.

[Min04] Minerule mining system (demo version). http://kdd.di.unito.it/minerule2/demo.html, 2004.

[Mit99] Tom M. Mitchell. Machine learning and data mining. *Commun. ACM*, 42(11):30–36, 1999.

[Mla98] Dunja Mladenic. Turning yahoo to automatic web-page classifier. In *European Conference on Artificial Intelligence*, pages 473–474, 1998.

[MLK04] Rosa Meo, Pier Luca Lanzi, and Mika Klemettinen, editors. *Database Support for Data Mining Applications*, volume 2682 of *Lecture Notes in Computer Science*. Springer-Verlag, Inc., 2004.

[MOF] Meta-object facility (mof) specification, vers. 1.4. http://www.omg.org/technology/documents/formal/mof.htm.

[Mol04] Molfea: the molecular feature miner based on the lvs algorithm (demo version). http://taranto.informatik.uni-freiburg.de/cgi-bin/molfea/molfea.cgi, 2004.

[MPC98] Rosa Meo, Giuseppe Psaila, and Stefano Ceri. An extension to SQL for mining association rules. *Data Min. Knowl. Discov.*, 2(2):195–224, 1998.

[NHL98] Raymond T. Ng, Jiawei Han, and Laks V. S. Lakshmanan. Editorial. *Data Min. Knowl. Discov.*, 2(3):231–232, 1998.

[NPP] Chikahito Nakajima, Massimiliano Pontil, and Tomaso Poggio. People recognition and pose estimation in image sequences. In *IEEE-INNS-ENNS International Joint Conference on Neural Networks*.

[OLE] Ole db for datamining specification. http://www.microsoft.com/data/oledb.

[Ora] Oracle10g database. http://www.oracle.com/technology/products/database/oracle10g/.

[OWL] Web ontology language. http://www.w3.org/2001/sw/WebOnt.

[PAN02] Panda project. http://dke.cti.gr/panda, 2002.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PMM] Predictive model markup language. http://sourceforge.net/projects/pmml.

[PRC00] François Pachet, Pierre Roy, and Daniel Cazaly. A combinatorial approach to content-based music selection. *IEEE MultiMedia*, 7(1):44–51, – 2000.

[PSY] Psycho web site. http://www.disi.unige.it/people/CataniaB/psycho/.

[PTK02] Aggelos Pikrakis, Sergios Theodoridis, and Dimitris Kamarotos. Recognition of isolated musical patterns using hidden markov models. In *ICMAI*, pages 133–143, 2002.

[Qia96] Xiaolei Qian. Query folding. In Stanley Y. Su, editor, *12th Int. Conference on Data Engineering*, pages 48–55, New Orleans, Louisiana, 1996.

[Rae02] Luc De Raedt. A perspective on inductive databases. *SIGKDD Explor. Newsl.*, 4(2):69–77, 2002.

[RBC+03] Stefano Rizzi, Elisa Bertino, Barbara Catania, Matteo Golfarelli, Maria Halkidi, Manolis Terrovitis, Panos Vassiliadis, Michalis Vazirgiannis, and Euripides Vrachnos. Towards a logical model for patterns. In *Proc. of the 22nd International Conference on Conceptual Modeling (ER 2003)*, pages 77–90, 2003.

[RDF] Resource description framework. http://www.w3.org/RDF/.

[Rev90] Peter Z. Revesz. A closed form for datalog queries with integer order. In *ICDT*, pages 187–201, 1990.

[Rev93] Peter Z. Revesz. A closed-form evaluation for Datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116(1):117–149, 1993.

[Rev95a] Peter Z. Revesz. Constraint databases: A survey. In *Semantics in Databases*, pages 209–246, 1995.

[Rev95b] Peter Z. Revesz. Datalog queries of set constraint databases. In *ICDT '95: Proceedings of the 5th International Conference on Database Theory*, pages 425–438, London, UK, 1995. Springer-Verlag.

[Rev02] Peter Revesz. *Introduction to Constraint Databases*. Springer, 2002.

[RJLM02] Luc De Raedt, Manfred Jaeger, Sau D. Lee, and Heikki Mannila. A theory of inductive query answering. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 123, Washington, DC, USA, 2002. IEEE Computer Society.

[RS02] John F. Roddick and Myra Spiliopoulou. A survey of temporal knowledge discovery paradigms and methods. *IEEE Trans. Knowl. Data Eng.*, 14(4):750–767, 2002.

[SB99] Stephan Schulz and Felix Brandt. Using term space maps to capture search control knowledge in equational theorem proving. In *FLAIRS Conference*, pages 244–248, 1999.

[sic] Sicstus prolog (v.3). http://www.sics.se/isl/sicstuswww/site/index.html.

[Sno95] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[SQLa] Iso sql/mm part 6. http://jtc1sc32.org/doc/N0601-0650/32N0647T.pdf.

[SQLb] Microsoft sql server analysis server. http://www.microsoft.com/sql/evaluation/bi/bianalysis.asp.

[ST96] Abraham Silberschatz and Alexander Tuzhilin. What makes patterns interesting in knowledge discovery systems. *IEEE Trans. Knowl. Data Eng.*, 8(6):970–974, 1996.

[Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, California, 1951.

[TC98] David Toman and Jan Chomicki. Datalog with integer periodicity constraints. *J. Log. Program.*, 35(3):263–290, 1998.

[TCG+93] Abdullah U. Tansel, James Clifford, Shashi K. Gadia, Arie Segev, and Richard T. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[TK03] S. Theodoridis and K. Koutroumbas. *Pattern Recognition. 2nd Edition.* Academic Press, 2003.

[TVS+04] Manolis Terrovitis, Panos Vassiliadis, Spiros Skiadopoulos, Elisa Bertino, Barbara Catania, and Anna Maddalena. Modeling and language support for the management of pattern-bases. In *SSDBM*, pages 265–274, 2004.

[UML] Uml specification, vers. 1.5. http://www.omg.org/technology/documents/formal/uml.htm.

[vdM92] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *PODS '92: Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 331–345, New York, NY, USA, 1992. ACM Press.

[VV05] Keshri Verma and O. P. Vyas. Efficient calendar based temporal association rule. *SIGMOD Rec.*, 34(3):63–70, 2005.

[XMI] Xml metadata interchange specifications vers.2.0. http://www.omg.org/technology/documents/formal/xmi.htm.

[yah] Yahoo! Main Page available at http://www.yahoo.com.

[Yan81] Mihalis Yannakakis. Node-deletion problems on bipartite graphs. *SIAM J. Comput.*, 10(2):310–327, 1981.

Appendix A

First-order theories with constraints

In the following, we briefly present the main concepts concerning first-order theories with constraints, used to model formula and validity period pattern components inside $\mathcal{EPM}$.

A *first-order predicate calculus* is defined as follows. The *alphabet* of the language consists of the following:

- A countably infinite set of variables.
- A countably infinite set δ of constants. The domain of each variable is δ.
- A set $\mathcal{R}$ of relation symbols with a fixed arity, that is, number of arguments.
- Connectives: $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation).
- Quantifiers: $\exists$ and $\forall$.

The *predicate calculus formulas* are inductively defined as follows:

- If R is an n-ary relation symbol and $x_1, ..., x_n$ are variables or constants, $R(x_1, ..., x_n)$ is a formula.
- If ϕ is a formula, then $\neg\phi$ is a formula.
- If ϕ_1 and ϕ_2 are formulas, then $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ are formulas.
- If ϕ is a formula and x is a variable, then $\exists x(\phi)$ and $\forall x(\phi)$ are formulas.

In the last line of the above definition, we call each occurrence of x in ϕ a *bound* variable. Variables that are not bound are called *free* variables in a formula.

Constaint name	**Constraint form**[a]
Lower bound	$u\theta b$
Upper bound	$-u\theta b$
Equality	$u = v$
Disequality	$u \neq v$
Order	$u\theta v$
Gap-Order	$u - v\theta b$ where $b \geq 0$
Linear	$c_1x_1 + ... + c_nx_n\theta b$
Polinomial	$p(x_1, ..., x_n)\theta b$
Simple Periodicity	$u \equiv_k c$, $k \in K$
Periodicity	$u \equiv_k (v + c)$, $k \in K$ and $c \in \{0, ..., k-1\}$

Legenda:
$\theta \in \{\geq, \geq\}$
[a] b and ci are variables belonging to the considered domain
u and v are either variables or constants
p is a polynomial function over variables $x_1, ..., x_n$
K is a finite set of natural numbers

Table A.1: Constraint normal forms

By first-order theories with constraints we mean various subsets of first-order predicate calculus in which the only relations are constraints. Different constraint classes can be considered. Examples of basic constraints are presented in Table A.1. Such constraints can be defined over any of the domains $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$, and $\mathbb{R}$. We notice that the constraints reported in Table A.1 share some relationships each other. For example, each order constraint is also a gap-order constraint, which in turn is a linear constraint. At the same manner, a linear constraint is also polynomial constraint. Concerning periodicity constaints, it is quite obvious that a simple periodicity is also a peridicity constraint.

A first-order theory with constraints for complex-objects is a strongly typed extension of the first-order theories with constraints defined above, defined over a multi-type domain representing complex-objects, as defined in Chapter 4. Besides constraints over atomic domains, a first-order theory with constraints for complex-objects provides additional predicates to cope with complex types (sets and tuples), such as inclusion or containment.

Given a first-order theory with constraints, several problems can be defined:

- *Decision problem*: problem of identifying whether a first-order formula without free variables is true or false.
- *Satisfiability*: problem of identifying whether there exists at least one assignment of constant values to variables which make the first-order formula true in the considered domain. We notice that the satisfiability problem for a formula ϕ with free variables

$x_1, ..., x_n$ can always be seen as a special case of the decision problem for the formula $\exists x_1, ...x_n(\phi)$.

- *Formula evaluation*: problem of identifying whether a formula without variables is true or false. We notice that this problem can be seen as a special case of the decision problem, since the input formula does not contain free variables.

- *Query evaluation*: given a formula ϕ with free variables $x_1, ..., x_n$ and a set $d \subseteq \delta$, problem of computing $\{(x_1, ..., x_n) | x_1 \in d \wedge ... \wedge x_n \in d \wedge \phi(x_1, ..., x_n)\}$.

Appendix B

Proofs of the Results Presented in Chapter 6

Lemma 1 $\mathcal{E}APQL \subseteq \mathcal{E}CPQL^{SS}$.

Proof: We assume that a preprocessing step is preliminarly applied, replacing each pattern identifier with the corresponding value. In the following, we present the translation from algebra to calculus only for the algebraic operators introduced explicitly for dealing with patterns and for those operators having a different semantics with respect to the ones proposed for complex objects [AB95]. We, therefore, do not provide an explicit translation for standard set-based operators and for the operators used within the reconstruction operation, which are similar to the ones proposed by Abiteboul and Beeri. For those operators, we refer the reader to [AB95].

Selection The selection algebraic operation $\sigma_{pr}(c_1)$ can be translated in calculus as follows:

$\{p/\exists p_1\, c_1(p_1) \wedge pr(p_1) \wedge p = p_1\}$

Measure projection The measure projection algebraic operation $\pi_{lm}(c_1)$ where $lm = m_1, ..., m_k$ and $lm \subseteq PT_1.ms$ s.t $c_1 : PT_1$ can be translated in calculus as follows:

$\{p/\exists p_1\, c_1(p_1) \wedge p.pid = new() \wedge p.s = p_1.s \wedge p.d = p_1.d \wedge p.m =< p_1.m_1, ..., p_1.m_k > \wedge p.f = p_1.f \wedge p.v = p_1.v\}$

Reconstruction The reconstruction algebraic operation $\rho_Q(c_1)$, where Q is a query over the pattern structure using complex value algebraic operators similar to the ones introduced in [AB95] (e.g., projection, P-projection, tuple_create, set_create, tuple_destroy, set_destroy, nest, and unnest), can be translated in calculus as follows:

$\{p/\exists p_1 \in c_1 \wedge p.pid = p_1.pid \wedge p.s = Q(p_1.s) \wedge p.d = p_1.d \wedge p.m = p_1.m \wedge p.f = pt_1.f[p.s/pt.s] \wedge p.v = p_1.v\}$

Join The join algebraic operation $c_1 \bowtie_{F,j} c_2$, where F is the join predicate constructed according to predicates previously introduced in Section 5.2.1 and $j = (newpid(), j_s, j_d, j_m, j_f, j_v)$ is the joining function which determines the pattern type of the resulting patterns, can be translated in calculus as follows:

$\{p/\exists p_1 \exists p_2\,(c_1(p_1) \wedge c_2(p_2) \wedge F(p_1, p_2) \wedge p = j(p_1, p_2))\} =$
$\{p/\exists p_1 \exists p_2\,(c_1(p_1) \wedge c_2(p_2) \wedge F(p_1, p_2) \wedge p.pid = newpid() \wedge p.s = j_s(p_1.s, p2.s)$
$\wedge p.d = j_d(p_1.d, p_2.d) \wedge p.m = j_m(p_1.m, p_2.m) \wedge p.f = j_f(p_1.f, p_2.f) \wedge p.v = j_v(p_1.v, p_2.v))\}$

Decomposition Let pt_1 and pt_2 be two pattern types such that $pt_1 \hookrightarrow pt_2$ and c_1 be a class defined for pt_1. The decomposition operator $\mathcal{C}_{pt_2}(c_1)$ can be translated in calculus as follows:

$\{p|\exists p_1\,(c_1(p_1) \wedge \exists p_2(p_1 \hookrightarrow p_2 \wedge p_2 \in ext(pt_2)) \wedge p = p_2)\}$

Drill-down Let pt_1 and pt_2 be two pattern types such that $pt_1 \Rightarrow pt_2$ and c_1 be a class defined for pt_1. The drill-down operator $\delta_{pt_2}(c_1)$ can be translated in calculus as follows:

$\{p|\exists p_1\,(c_1(p_1) \wedge \exists p_2(p_1 \Rightarrow p_2 \wedge p_2 \in ext(pt_2)) \wedge p = p_2)\}$

Roll-up Let pt_1 and pt_2 be two pattern types such that $pt_1 \Rightarrow pt_2$ and c_2 be a class defined for pt_2. The roll-up operator $\rho_{pt_1}(c_2)$ can be translated in calculus as follows:

$\{p|\exists p_2\,(c_2(p_2) \wedge \exists p_1(p_1 \Rightarrow p_2 \wedge p_1 \in ext(pt_1)) \wedge p = p_1)\}$

Drill-through The drill-through algebraic cross-over operation $\gamma(c_1)$ can also be translated in calculus. Indeed, since $\mathcal{EPM}$ is parametric with respect to the query language $\mathcal{Q}$ (see Section 4.2), a pattern data source $(p.d)$ is intensionally described as a query expressed in $\mathcal{Q}$ over a certain database instance db belonging to the Universal Data Base, if we do not consider the refinement relationship, or over a certain pattern base instance pb belonging to the PBMS, if there exists a refinement relationship. Thus, assuming the query describing $p.d$ has a correspondent calculus query, the drill-through algebraic operation can be translated using one of the following calculus query:

$\{t/\exists p_1\, c_1(p_1) \wedge p_1.d(t)\}$

Data-covering The data covering algebraic cross-over operation $\theta^D(c_1, D)$ can be translated in calculus as follows:

$\{t/\exists t_1 \in D \wedge \exists p_1\, c_1(p_1) \wedge p_1.f(t_1) \wedge t = t_1\}$

Pattern-covering The pattern covering algebraic cross-over operation $\theta^P(c_1, D)$ can be translated in calculus as follows:

$\{p/\forall t_1 \in D \wedge \exists p_1\, c_1(p_1) \wedge p_1.f(t_1) \wedge p = p_1\}$ □

Lemma 2 $\mathcal{E}CPQL^{SS} \subseteq \mathcal{E}APQL$.

Proof: In the following, we provide a translation of each calculus expression into an equivalent $\mathcal{E}APQL$ query.

The proof consists in constructing for each domain-independent $\mathcal{E}CPQL^{SS}$ query ($q = \{x|\varphi\}$) over a pattern base an equivalent $\mathcal{E}APQL$ query and the proof is an extension of the proof of the same result in the context of complex-values databases provided in [AB95]. Indeed, since our pattern theory is an extension of the complex value theory presented in [AB95], it is sufficient to demonstrate that all $\mathcal{E}CPQL^{SS}$ P-queries can be expressed by using a $\mathcal{E}APQL$ expression. The equivalence between D-queries and $\mathcal{E}APQL$ queries is inherited from the analogous result already demonstrated in [AB95].

In order to state the equivalence between the two querying approaches we exploit the notions of active domains given in Section 6.1.1.2.

The key idea of the proof is to construct, for each subformula ψ of ϕ, an algebraic formula E_ψ with the property that for each input $\mathbf{I}$:

$$E_\psi = \{y|\exists x_1, ..., x_n(y =< A_{x_1} : x_1, ..., A_{x_n} : x_n > \wedge\psi(x_1, ..., x_n))\}_{adom}(\mathbf{I})$$

where $x_1, ..., x_n$ are free variables in ψ.

Therefore, the proof is completed in three steps: the *active domain computation*, the *complex values construction*, and the *construction of the algebraic query*, which are presented in more detail in the following.

Active domain computation. The goal of this step is to construct an algebra query E_{adom} having type **dom** such that, on input instance $\mathbf{I}$, $E_{adom}(\mathbf{I}) = adom(q, \mathbf{I})$ for each calculus query q. To this end, we prove by induction that for each type $\tau \in \mathcal{T}_e \cup \mathcal{PT}$, there exists an algebra operation F_τ that maps a set I of values of type τ to $adom(I)$.

Base case: When τ is a type for atomic constants, for pattern identifiers or for type names (i.e., $\tau \in \mathcal{AT} \cup \mathcal{OID} \cup \mathcal{L}$), it suffices to use for F_τ an identity algebraic operation (e.g. $tup\_create_A \circ ttup\_destroy$).

Inductive case: The following cases have to be considered:

1. τ is a tuple type with one attribute, i.e. τ is $< A_1 : \tau_1 >$. Then F_τ is

$$F_{\tau_1}(tup\_destroy).$$

2. τ is a tuple type with at least two attribute components, i.e. τ is $< A_1 : \tau_1, ..., A_n : \tau_n >$ with $n \geq 2$. Then F_τ is

$$F_{<A_1:\tau_1>}(\pi_{A_1}) \cup ... \cup F_{<A_n:\tau_n>}(\pi_{A_n}).$$

Note that each $pt \in PT$ is a particular case of tuple type with exactly six components. Thus, every pattern type can be treated at the same manner of a tuple type.

3. τ is a set type, i.e. τ is $\{\tau_1\}$. Then F_τ is

$$F_{\tau_1}(set\_destroy).$$

4. τ is a pattern type, i.e. $\tau \in \mathcal{PT}$ is $pt_{name} :< s : \tau_s, d : \tau_d, m : \tau_m, f : \tau_f, v : \tau_v >$, with $\tau_v \equiv \mathcal{V}$. Then F_τ is

$$F_{pt_{name}} \cup F_{<s:\tau_s>}(\pi_s) \cup F_{<d:\tau_d>}(\pi_d) \cup F_{<m:\tau_m>}(\pi_m) \cup F_{<f:\tau_f>}(\pi_f) \cup F_{<v:\tau_v>}(\pi_v).$$

Now consider the pattern system with a pattern base component PB with schema $\widehat{PB}$ and a data base component with DB with schema $\widehat{DB}$

For each schema relation $\widehat{R} \in \widehat{DB}$, $F_{type(\widehat{R})}$ maps a relation instance I over $\widehat{R}$ to $adom(I)$. Thus $adom(q, \mathbf{I})$ can be computed with the query:

$$E_{adom} = F_{type(\widehat{R_1})}(R_1) \cup ... \cup F_{type(\widehat{R_m})}(R_m) \cup \{a_1\} \cup ... \cup \{a_p\}$$

where $R_1, ..., R_m$ are relations in DB and $a_1, ..., a_p$ is the list of elements occurring in q.

Similarly, for each pattern class schema $\widehat{C} \in \widehat{PB}$, $F_{type(\widehat{C})}$ maps a pattern class instance I over $\widehat{C}$ to $adom(I)$. Thus $adom(q, \mathbf{I})$ can be computed with the query:

$$E_{adom} = F_{type(\widehat{C_1})}(C_1) \cup ... \cup F_{type(\widehat{C_m})}(C_m) \cup \{a_1\} \cup ... \cup \{a_p\}$$

where $C_1, ..., C_m$ are pattern classes in PB and $a_1, ..., a_p$ is the list of elements occurring in q.

Complex values construction. At this stage, we prove by induction that for each type $\tau \in \mathcal{T}_e \cup \mathcal{PT}$, there exists an algebra query G_τ that construct the set of values I of type τ such that $adom(I) \subseteq adom(q, I)$.

Base case: When τ is a type for atomic constants, for pattern identifiers or for type names (i.e., $\tau \in \mathcal{AT} \cup \mathcal{OID} \cup \mathcal{L}$), we can use E_{adom}.

Inductive case: Three cases occur[1]:

1. τ is $< A_1 : \tau_1, ..., A_n : \tau_n >$: then G_τ is $tup\_create_{A_1,...,A_n}(G_{\tau_1}, ..., G_{\tau_n})$;
2. τ is $\{\tau_1\}$: then G_τ is $nest(G_{\tau_1})$;
3. τ is $pt_{name} :< s : \tau_s, d : \tau_d, m : \tau_m, f : \tau_f, v : \tau_v >$, with $\tau_v \equiv \mathcal{V}$: then G_τ is $tup\_create_{pt_{name},s,d,m,t,v}(G_\mathcal{L}, G_{\tau_s}, G_{\tau_d}, G_{\tau_m}, G_{\tau_f}, G_{\tau_n})$.

Construction of the algebraic query. The last step consists into an inductive construction of the queries E_ψ for subformula ψ of φ. We assume without loss of generality that the logical connectives $\vee$ and $\forall$ do not occur in φ. We also assume that atomic formulas relying on relation and pattern class names, used as predicates, in φ do not contain constants or repeated variables.

As we have already pointed out (see the beginning of Section 6.1), according to the $\mathcal{EPM}$ model, patterns can be viewed as complex objects, except for their *pid* component. However, by applying an initial preprocessing phase, which takes polynomial time, they can be reduced to complex objects by eliminating their pid.

Therefore, the construction of the algebraic query corresponding to subformulas appearing in calculus queries over patterns can be reduced to the analogous procedure for complex objects presented in [AB95], where traditional relational cases are treated in standard way as shown in [AHV95].

Special considerations are due only for the membership predicate.[2]

Let q be the query $\{e|\phi(e)\}$ and let ψ be $x \in y$ a subformula of ϕ. Suppose that x is of type τ, then y is of type $\{\tau\}$. The set of value of type $\{\tau\}$ within the active domain is returned by the expression $G_{\{\tau\}}$. Thus, the query

$$\sigma_{A_x \in A_y}(tup\_create_{A_x,A_y}(G_\tau, G_{\{\tau\}}))$$

returns the desired result. With this construction, E_φ returns a set of tuples with a single attribute A_x. Thus, the query q is equivalent to $tup\_destroy(E_\varphi)$. □

[1] Actually, since patterns are in fact tuples, we may device simple two inductive cases. However, for the sake of clarity, we explicitly refer to the case in which τ is a type in $\mathcal{PT}$.

[2] Recall that extensional containment predicates are not allowed in the strictly safe version of the pattern calculus proposed, as well as for the strictly safe calculus for complex objects proposed in [AB95].